Spring 2013

Data carving parser generation

Benjamin Nathaniel Kelley
James Madison University

Data Carving Parser Generation

Benjamin Nathaniel Kelley

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Computer Science

May 2013

Table of Contents

List of Tables

List of Figures

Abstract

As our day to day interaction with technology continues to grow, so does the amount of data created through this interaction. The science of digital forensics grew out of the need for specialists to recover, analyze, and interpret this data. When events or actions, either by accident or with criminal intent create, delete or manipulate data, it is the role of a digital forensics analyst to acquire this data and draw conclusions about the discovered facts about who or what is responsible for the event. This thesis identifies a gap in the research between data analysis and interpretation. Current research and tool development has been focusing on data acquisition techniques and file carving. Data acquisition is the process of recovering a forensically sound copy of the evidence, such as a bit-by-bit copy of a hard drive or an image of the contents of a computer system's RAM. File carving is the process of searching for and extracting files from the acquired data. Few tools provide a means of quick and easy file validation and data extraction once they have been recovered, and the tools that do are either limited in their ability or very complex and require a lot of overhead and a steep learning curve to use effectively. The tool created through this research fills this gap. The tool utilizes a file description language that can textually describe the layout and on-disk format of a file type's data. This language is very intuitive and easy to read and understand by humans. By using a description as input, the tool builds a syntax tree which can be used to parse and extract various fields of interest from any file matching the provided description. This allows for the quick analysis and interpretation of any file type, even those with uncommon or proprietary formats, as long as a valid description is provided.

1. Introduction

As technology advances, so does our daily interaction with it. For many of us, it is hard to think of going a day without the use of some form of computer, smart phone, PDA, or any other kind of electronic device. As our use and dependency on technology grow, so does the amount of data created by them. The field of digital forensics has evolved based on the increasing need to recover and analyze data when events such as hard drive crashes, virus infections, or actions taken by users cause damage or provide insight to a criminal investigation.

Technology has become integrated and necessary in our day-to-day lives. It is highly likely most crimes have a digital element. Terrorists are using the Internet to communicate, recruit new members, or spread propaganda. The threat of network-based attacks are an impending concern for governments, companies, and individuals alike; the Internet Crime Complaint Center (IC^3) has seen a drastic increase in the number of reported Internet based crimes over the past decade [1]. Computers are being utilized by a variety of offenders for various goals. They are, for example, being used by some criminals to communicate, plan crimes, and lure targets or by those wanting to store and share contraband material such as child pornography.

This interaction with technology creates evidence and this evidence can take many forms. Files are downloaded, created, edited, transferred and deleted. Many computer systems keep logs of user, system, or network activity that contain evidence of illicit activity. A computer system's RAM can also hold evidence of a criminal's activity: what processes were running, files were open, or other operating system metadata.

Traditional forensics has long upheld Locard's Exchange Principle [2]. This principle states that any interaction between two items results in a transfer of evidence: between a culprit and victim, weapon, or even the crime scene itself. This rule extends to the digital realm as well. Attackers in a network intrusion often leave trails of their activity through system logs, file system or operating system metadata, as well as many other sources. The role of a digital forensics analyst is to collect and analyze this data and, importantly, be able to attest to its accuracy and soundness [3].

Digital forensics and the importance of digital evidence acquisition has existed for many years, but only recently has been identified as a scientific discipline. In 2008 the American Academy of Forensic Science (AAFS) created a new department that focuses on the development of Digital and Multimedia Sciences (DMS). Although the creation of the DMS established digital forensics as a scientific discipline, academics and practitioners in the field still face many challenges. The DMS is working towards the creation of digital forensic standards and practices that is still in its infancy [3]. In 2009 the National Academy of Sciences (NAS) stated [4]:

> Over the past 10 years, this process has become more routine and subject to the rigors and expectations of other fields of forensics science. Three holdover challenges remain: (1) the digital evidence community does not have an agreed certification program or list of qualifications for digital forensic examiners; (2) some agencies still treat the examination of digital evidence as an investigative rather than a forensic activity; and (3) there is wide variability in and uncertainty about the education, experience, and training of those practicing this disciple.

Members of the NAS and the AAFS along with other United States and international agencies and organizations are working together to develop a standardized set of guidelines, qualifications, and practices to establish digital forensics as a recognized and reputable field of hard science. There are also many law enforcement officers and academic researchers working to develop tools and methods for the collection, analysis and interpretation of digital evidence.

Evidence from digital devices can be collected in numerous ways; for example, through physical acquisition utilizing specialized hardware tools, or by logical acquisition through the device's file system. Research to improve and develop new methods to acquire forensically sound copies of recovered data has been an area of interest and have been rapidly developed over the past few decades. One topic of data recovery that has seen numerous advancements in recent years is file carving [5]. File carving is the process of locating and extracting files from a larger source, such as an image of a hard drive or memory.

Equally important to obtaining data are the specific techniques and methods employed to analyze the recovered data throughout an investigation. Regardless of how the data is recovered, until it is analyzed and presented in a useable form, no information can be obtained or interpreted from it. The goal of analysis is to take acquired evidence and produce a timeline of events, draw correlations, and ideally insights into whom or what was responsible for the event of interest. Both academic and commercial research and tool development has been conducted, which has led to great advancements in this area. It is the purpose of this research to contribute to this specific area of digital forensics by proposing an innovative method of metadata extraction from recovered files.

Upon reviewing the current body of knowledge regarding file carving and the freely and commercially available tools to assist with this task, it is clear a gap exists in the research.

Much of the research being conducted deals with methods of carving data from memory dumps, hard drive images, or other sources of acquired data. Many of the available tools simply rely on implementations of these various methods. Although there have been great improvements over the past few years in carving techniques [5], few tools focus on interpreting the data within the files once they have been carved. Some commercially available tools, such as EnCase [6] or X-Ways [7], support a minimal amount of interpreting, parsing, and presenting the meta-data and data within the files, but both are limited to files the application supports or great effort from the examiner is required to parse specific fields of data.

This gap in the research makes clear a need to develop more effective solutions to the problem of how to analyze and interpret data once it has been recovered through carving techniques. A tool should be developed to resolve this issue because time and effort are wasted when an examiner needs to manually interpret data or write program(s) to interpret data from an unsupported file type. Such a tool should possess the following characteristics:

1. Readable and structured input and output.
2. Customizable to fit the needs of any investigation.
3. Automatically parses meta-data from files.

This thesis outlines the development of such a tool. To allow for customization, the tool relies on an existing file description language. The language allows an examiner to textually describe the layout of a binary file type. This description is human readable and easily understood. This existing description language will be used as input to a tool that parses and interprets a binary file for the file type the examiner provides. The tool will allow the examiner to validate a file and quickly extract any fields of data of interest from the file. Using the file description language has the advantage of quickly parsing for any file type needed, even uncommon or proprietary files types, as long as a valid description is provided.

The remainder of this thesis discusses background research, tool development, and results of this thesis. Chapter 2 reports in further detail related work regarding digital forensics and data carving. It also discusses the currently available tools that accomplish tasks similar to the tasks this tool does. The limitations of these tools are also identified. Chapter 3 provides a detailed discussion of the file description language used by this tool. It covers all the features of the language supported by the initial prototype of the tool. Chapter 3 also discusses the features left out and details the reasons for their omission. Chapter 4 presents the development of the tool and provides detailed discussions of the phases of development and the implementation of the program. Chapter 5 discusses the capabilities of the tool prototype, provides examples of its execution, and presents future work possibilities and extensions to future versions of the tool.

2. Background

One particular area of interest in recovery is data carving. Data carving, sometimes referred to as file carving or simply carving, is usually defined as the process of finding and extracting useful data from a data source [5] [8]. Files recovered from hard drive images, memory dumps, or other sources of data can be a source of evidence in an investigation. This has made carving an area of great interest within the digital forensics field and new methods, techniques, algorithms, and tools are being actively developed. Data carving and recovery techniques can broadly be divided into two large categories: hardware and software. Hardware recovery is the development of hardware devices to extract data from a source of interest. Software recovery refers to the development of pieces of software that can help interpret and make sense of data after it has been recovered [5].

2.1 History of Data Carving

The history of data recovery and carving started with 0traditional techniques that relied on the file system's structures. In many file systems, when a file is deleted the data is actually still present, and links between associated blocks are active until the data is overwritten.

The evolution of data carving tools started with Start of File (SOF)/End of File (EOF) carving. SOF/EOF carvers use a simple method of scanning the data image for common file type headers and possibly common file footers, then extracting data in between. Although the method is simplistic, these carvers were (and sometimes still are) fairly effective and yielded good results [9].

However, fragmentation poses a large problem for SOF/EOF carvers. Fragmentation occurs when a file's data is not stored in contiguous sectors of the storage medium, and is instead split into two or more fragments. Although many file systems try to keep fragmentation to a minimum whenever possible, and modern operating systems have defragmentation services which run regularly, it is still impossible to avoid completely. Garfinkel discusses three conditions where files must be written in at least two fragments [10].

1. There are no sectors in a contiguous manner large enough to hold the file data.
2. A file is appended with additional data, and not enough empty sectors follow the original data. Although some systems will relocate the file, most will simply add the appended data elsewhere.
3. Certain file systems may not support writing very large files in more than a certain number of contiguous sectors.

Statistical surveys show fragmentation occurs more often in certain file types. For example, Garfinkel showed that, on average, Outlook files are the most commonly fragmented files, with 58% of files being split into at least two parts across the file system. Other file types identified as commonly being fragmented were 15% of .exe, 16% of .dll, and 17% of .doc files [10].

Current research and development pertaining to data carving is largely centered on methods of finding possible data from large data sources. As the field advances, more and more data carving methods and tools are being invented. Garfinkel and Metz have proposed eleven carving taxonomy categories [8].

1. Carving

2. Block-Based Carving
3. Statistical Carving
4. Header/Footer Carving
5. Header/Maximum Size Carving
6. Header/Embedded Length Carving
7. File Structure Based Carving
8. Semantic Carving
9. Carving with Validation
10. Fragment Recovery Carving
11. Repackaging Carving

The first category, Carving, is the general term applied to the process of searching for and extracting a file's data from a larger source of raw data. The other categories each refer to the specific technique the tool or method executes to achieve its goal.

Block-Based Carving tools scan the input in a block-by-block manner and attempt to determine if the current block is part of a file [8]. The carving tool Revit implements this technique, for example. Revit scans each block of input data against its configuration of file definitions [11].

Statistical Carving algorithms scan input clusters and use a set of characteristics and statistics related to them to determine if the cluster is a part of an output file. Veenman's work offers examples of certain features and characteristics that can be used to help statistically locate portions of files; the repetition or frequent use of the same symbol, such as angle braces in HTML files, or a data's entropy can be used as a statistical feature in these carving tools [12].

Header/Footer based carving methods is the category created for the techniques explained earlier in this section. Start of File and End of file carving methods, where the tool or examiner looks for well-known file headers or footers, fall into this category of carving tools.

The Header/Maximum Size Carving technique is similar in methodology to the Header/Footer algorithms. The carving tool or examiner scans the input for well-known file headers. Once located, the carver simply extracts the maximum file size specified. Although this is a crude technique and often data not associated with the file is also extracted, it is still an effective option for certain file types that are not affected if they contain extraneous data following a valid file [8].

The Header/Embedded length method of carving is an effective method for certain types of files, specifically files that contain a size, or means of determining its size, embedded within the header or data of the file. Tools using this method scan the input data searching for file header patterns. Once a file header is found, the size of the file is determined from values in the header or file data and only that amount of data is extracted.

File Structure Based Carving methods involve using more specific information about the structure and format of a file type's data in addition to the header and footer to help determine if a sector belongs to a file. Semantic Carving and Deep Carving have been used as names for this method [8]. Metz and Mora explain that the characteristics of a file's structure can be used to locate sectors belonging to it. In JPEG files, for example, each section after the header begins with the byte 0xff. Each section also contains an entry for its size, which can be used to extract sections of the file accurately[11].

Semantic Carving should not to be confused with the older name given to File Structure Based Carving methods. Semantic Carving methods involve examining the contents of a candidate file's data and using linguistic, grammatical, syntactic and semantic analysis to determine which sections are actually parts of the file. The Forensic Wiki offers an example of locating an English HTML file, with blocks of French text located within it. Using a semantic carver, the tool might determine that the French sections do not belong to the file and are extraneous data extracted from another file or a deleted file [8].

Fragment Recovery Carving is a category for any method that attempts to locate, extract and reconstruct fragmented files. Tools can incorporate other file carving strategies to locate fragmented pieces of a file.

Repackaging Carving involves locating and extracting parts of a larger file type and adds new headers, footers, or other necessary structures so the data can be examined using standard tools. Garfinkel's ZIP Carver locates and extracts single ZIP archive entries and places them into a new central directory so the files can be accessed with any available zip explorer or utility [8].

2.2 Volatility

The Volatility Framework is a collection of open source Python-based tools released under the GNU General Public License. Volatility is designed to help with collecting and extracting artifacts from a system's volatile memory (RAM). Volatility has a large range of supported memory formats and OS distributions. Currently Volatility supports all major 32 and 64 bit Windows versions (XP up to 7) and support for Linux kernels 2.6.11 – 3.5.x were recently added. OSX and Android support are currently being developed for future releases of the framework as well.

The Volatility Framework is a tool for analyzing a system’s kernel memory space. Given a memory dump, a Windows hibernation file, virtual machine snapshot, or other supported copy of RAM, the framework can help in the analysis by parsing out relevant information. Volatility can quickly give the user a summary of the state of the system at the time of the memory snapshot: what and how many processes were running by each user, open files, logged in users, and hundreds of other possibly valuable pieces of metadata stored in the kernel memory.

As helpful as the Volatility Framework is, it is limited to assisting in understanding and analyzing the contents of a system's RAM. The work in this thesis is similar in focusing on the parsing and extraction of meta-data, but differs in the source of data interpreted. Where Volatility focuses on the interpretation of kernel memory, this work focuses on interpreting any piece of binary data as long as a valid file description is provided [13].

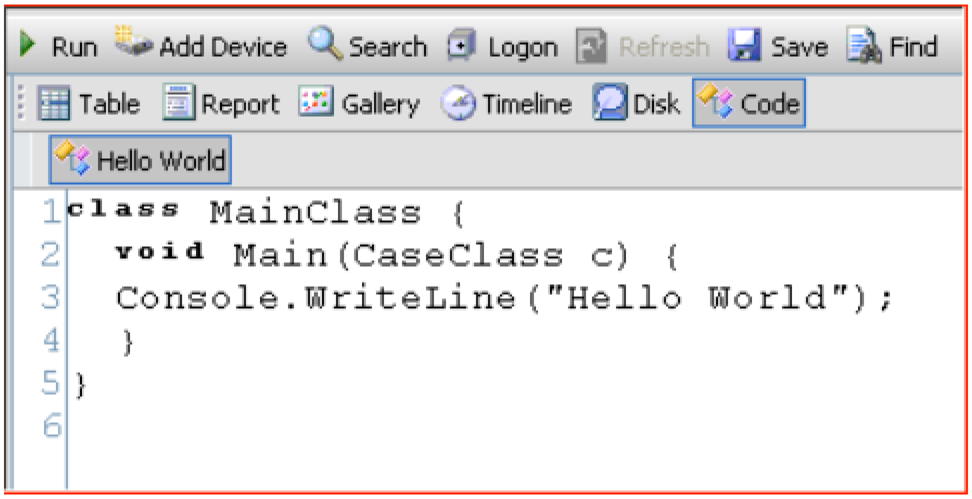
2.3 EnCase and EnScript

Guidance Software’s EnCase is arguably the market’s current leading forensic tool kit [14]. EnCase has become a standard and well trusted tool for criminal, private, and corporate investigations. EnCase is a full-featured tool suite that assists practitioners through every stage of an investigation. The software has tools to assist in forensically sound data acquisition from disk and RAM on a wide range of supported devices and technologies. It also offers assistance in data carving and file recovery from acquired disk images and tools for analyzing the recovered data. EnCase can generate timelines and reports based on recovered logs, file system metadata, and other sources of recovered evidence [6].

The most important feature, in regards to this thesis, is EnCase’s automation and extensibility features. The EnCase suite includes a scripting language called EnScript. EnScript began as a scripting language enabling users to automate other built-in features in EnCase, but quickly evolved into a complex higher-level language. The EnCase program has a built-in EnScript editor and compiler, allowing the user to write, edit, compile and package their EnScripts to share with other EnCase users.

Since its inception EnScript has grown substantially and has outgrown its original use of automating EnCase features. EnScript can access the local file system, execute other Windows programs, and the Enterprise edition provides remote access [15]. A full discussion on the possibilities of EnScript programs is well outside the scope of this paper because of its ever growing list of features and abilities. However, the remainder of this section provides an introduction to the basics of the language and examples of using EnScript to perform metadata extraction from recovered files.

The EnScript programming language is syntactically very similar to the C or C++ languages. Like C++, each EnScript must have a main class which contains a main method; other classes and methods can be created as needed by the examiner. Figure 1 shows how a simple EnScript program is structured.



```
class MainClass {
  void Main(CaseClass c) {
  Console.WriteLine("Hello World");
  }
}
```

Figure 1 Structure of a simple EnScript [15]

These EnScripts can be compiled to a proprietary format called EnPack. Since EnPacks are executed by EnCase, they can be transferred among investigators and executed on any platform that EnCase supports.

The EnScript language can be used to open a file, navigate through it, and extract various fields of data that might be of interest. Figure 2 below shows an excerpt from an EnScript program that parses Microsoft XP System Restore Logs [16].

```
...
class MainClass {
  void Main(CaseClass c) {
    // INPUT VALIDATION, Is there any evidence to process?
    SystemClass::ClearConsole();
    if (!c || !c.EntryRoot().FirstChild()) {
      SystemClass::Message(0, "No input defined!", "No entries in case!
Add an evidence file and then run script again.");
      return;
    }
    // INPUT VALIDATION END


    // SEARCH OBJECT CREATE AND INITIALIZE
    SearchClass search();

search.AddKeyword("\\x12\\xEF\\xCD\\xAB....\\x15\\x00\\x00\\x00....",
KeywordClass::GREP);
    search.Create();
    //

    // MAIN RECURSION LOOP
    forall (EntryClass e in c.EntryRoot()) {

      // The following logic checks if name begins with "change.log"
and parent folder name begins with "RP" and its parent name begins with
"_restore{"
      if (e.Name().Find("change.log")  == 0 &&
          e.Parent().Name().Find("RP") == 0 &&
          e.Parent().Parent().Name().Find("_restore{") == 0) {

// Create EntryFileClass object
        EntryFileClass file();

// Open the change.log file with the file object just created
        if (file.Open(e)) {

        // This invokes Find function and checks whether there
        are any search hits
          if (search.Find(file) > 0) {

            // Print information about folder RPxx to console
            Console.WriteLine("\n Created : " +
                              e.Created().GetString() + "
                              Folder: " +
                              e.Parent().FullPath());

            file.SetCodePage(CodePageClass::UNICODE); // <-- do not
forget to do this, the file object needs to be set to read unicode

            // ITERATE THROUGH SEARCH HITS HERE
            foreach (SearchClass::HitClass hit in search.GetHits()) {

              // Seek to 64 bytes from hit offset
              file.Seek(hit.Offset() + 64);

              // Define path and name variables as strings
              String fullpath,
                     newfilename;

              // Read path
              file.ReadString(fullpath);

              // Skip 8 bytes after path
              file.Skip(8);

              // Read name
              file.ReadString(newfilename);

              // Print information to console
              Console.WriteLine(newfilename + "      " + fullpath);
            }
            // END ITERATION LOOP
          }
          file.Close(); // Close the change.log file
        }
      }
    }// END MAIN RECURSION LOOP
  }
}
```

Figure 2 Excerpt of EnScript program for parsing Microsoft XP System Restore Change Logs [16].

The example script in Figure 2 performs three tasks:

1. Checks that the current case has evidence
2. Loops over the evidence looking for restore logs
3. Loops over each log and parses out the log entries.

The first goal is accomplished by:

```
SystemClass::ClearConsole();
    if (!c || !c.EntryRoot().FirstChild()) {
      SystemClass::Message(0, "No input defined!", "No
entries in case! Add an evidence file and then run script
again.");
      return;
    }
```

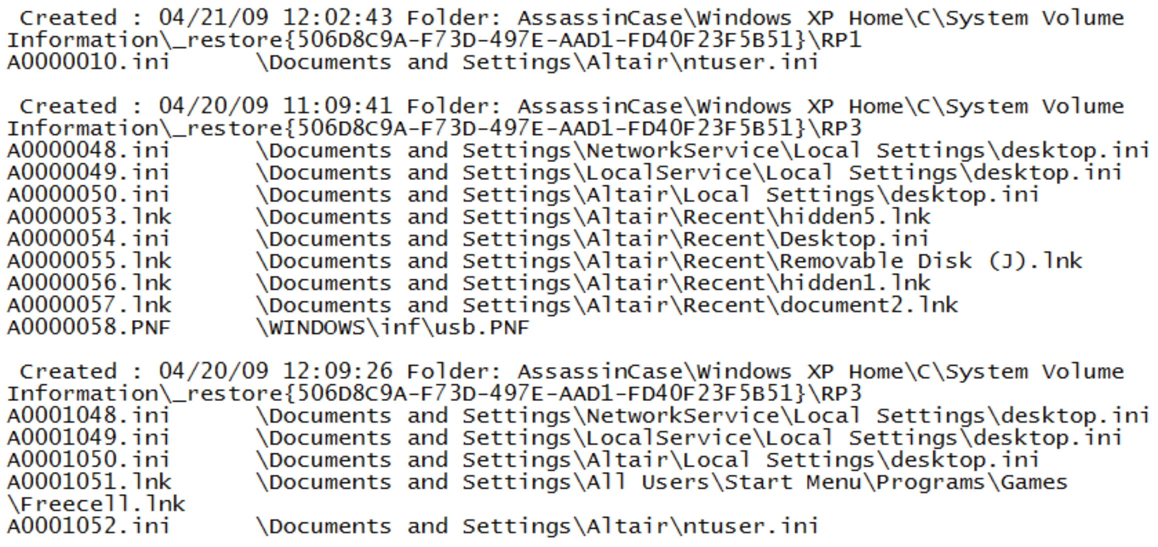
The script first clears the console. Then, if the current case opened in EnCase does not contain any evidence files, an error message is printed and the script aborted. Evidence is checked by the line if `(!c || !c.EntryRoot().FirstChild())`. The variable `c` is the opened case and the logic checks if it exists and that its first child in its list of evidence exists as well.

Before the program enters the two nested loops that look for log files and parse out the contents, a search object is made.

```
// SEARCH OBJECT CREATE AND INITIALIZE
    SearchClass search();

search.AddKeyword("\\x12\\xEF\\xCD\\xAB....\\x15\\x00\\x00\
\x00....", KeywordClass::GREP);
 search.Create();
```

As with many higher level languages, EnScript provides a standard library of classes and functions and `SearchClass` is among these. A `SearchClass` object is created and initialized by adding the keywords for the search. The hex string 0x12EFCDAB is at the start of each entry in the change logs files and is used as the keyword for the `SearchClass`. This will be used in the later loops to search the logs for entries to parse.

If the file is a change log file, the search initialized earlier is executed on them, `if (search.Find(file) > 0).` A message about the file is printed to the console and the inner loop is entered. The inner iteration loop loops over the search results: `foreach (SearchClass::HitClass hit in search.GetHits()).` Reading, writing, and seeking within files in EnScript is syntactically very similar to C or C++; file.seek() is used to move 64 bytes from the start of the log entry and file. `ReadString` is used to read in the full path name. Similarly, the filename is found 8 bytes later and read in. For each log entry, the full path name and the file name are read in and printed to the console. Figure 3 shows an example output of the script.



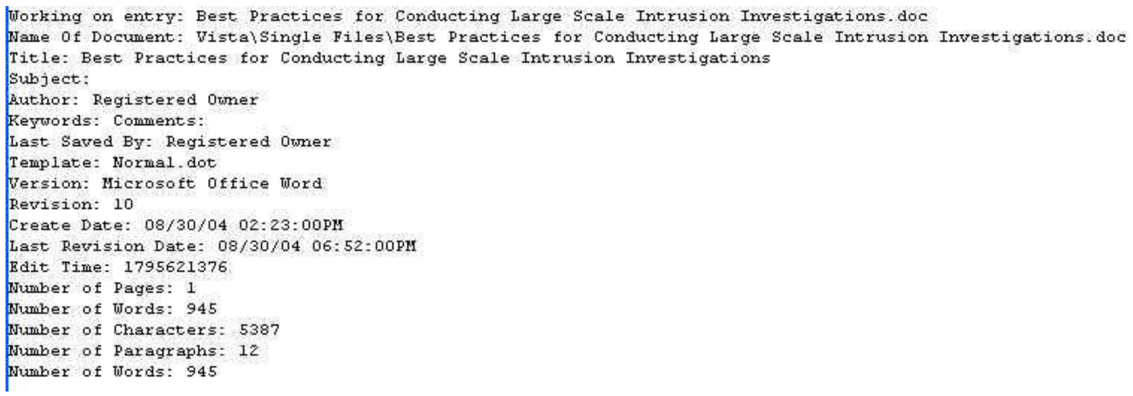
```
 Created : 04/21/09 12:02:43 Folder: AssassinCase\Windows XP Home\C\System Volume
Information\_restore{506D8C9A-F73D-497E-AAD1-FD40F23F5B51}\RP1
A0000010.ini      \Documents and Settings\Altair\ntuser.ini

 Created : 04/20/09 11:09:41 Folder: AssassinCase\Windows XP Home\C\System Volume
Information\_restore{506D8C9A-F73D-497E-AAD1-FD40F23F5B51}\RP3
A0000048.ini      \Documents and Settings\NetworkService\Local Settings\desktop.ini
A0000049.ini      \Documents and Settings\LocalService\Local Settings\desktop.ini
A0000050.ini      \Documents and Settings\Altair\Local Settings\desktop.ini
A0000053.lnk      \Documents and Settings\Altair\Recent\hidden5.lnk
A0000054.ini      \Documents and Settings\Altair\Recent\Desktop.ini
A0000055.lnk      \Documents and Settings\Altair\Recent\Removable Disk (J).lnk
A0000056.lnk      \Documents and Settings\Altair\Recent\hidden1.lnk
A0000057.lnk      \Documents and Settings\Altair\Recent\document2.lnk
A0000058.PNF      \WINDOWS\inf\usb.PNF

 Created : 04/20/09 12:09:26 Folder: AssassinCase\Windows XP Home\C\System Volume
Information\_restore{506D8C9A-F73D-497E-AAD1-FD40F23F5B51}\RP3
A0001048.ini      \Documents and Settings\NetworkService\Local Settings\desktop.ini
A0001049.ini      \Documents and Settings\LocalService\Local Settings\desktop.ini
A0001050.ini      \Documents and Settings\Altair\Local Settings\desktop.ini
A0001051.lnk      \Documents and Settings\All Users\Start Menu\Programs\Games
\Freecell.lnk
A0001052.ini      \Documents and Settings\Altair\ntuser.ini
```

Figure 3 Example output of the XP System Restore Log parser [16].

The script enters the two nested loops that the author refers to as MAIN RECURSION

LOOP and ITERATION LOOP. The main recursion loop loops over each piece of evidence in the open case, then checks if the file is an XP System Restore Log file by checking its filename and parent directory names.

Much like the goal of the tool discussed in this thesis, EnScript can be used to extract and present fields of metadata from files as well. For example, Muller provides on his website an EnPack for a script written to read, extract, and display the metadata fields within a Microsoft Document file [17]. The source code for the script is not provided, but Figure 4 shows a screen shot of its execution.



```
Working on entry: Best Practices for Conducting Large Scale Intrusion Investigations.doc
Name Of Document: Vista\Single Files\Best Practices for Conducting Large Scale Intrusion Investigations.doc
Title: Best Practices for Conducting Large Scale Intrusion Investigations
Subject:
Author: Registered Owner
Keywords: Comments:
Last Saved By: Registered Owner
Template: Normal.dot
Version: Microsoft Office Word
Revision: 10
Create Date: 08/30/04 02:23:00PM
Last Revision Date: 08/30/04 06:52:00PM
Edit Time: 1795621376
Number of Pages: 1
Number of Words: 945
Number of Characters: 5387
Number of Paragraphs: 12
Number of Words: 945
```

Figure 4 Output from Microsoft Office file metadata parser written in EnScript [17].

Although useful, the source code of EnScript programs can quickly become hard to read and understand, especially for large or complex file types. Another disadvantage of using EnScript for the purpose of meta-data extraction is its steep learning curve. The EnScript language has many built-in classes and functions that require programmers to have much prior knowledge and experience to program more advanced tasks.

Due to its reliance on C++ style syntax, the language does not lend itself very well to the description of a file's format and data structures. Overhead is required to open, navigate, and read a file and this extra code can distract the reader from learning the file's structure.

The final limitation observed for using EnScript for metadata extraction is that the examiner must write an entirely new program for each file type of interest. This requires the examiner to not only learn and understand the structure of the file, but also to spend the added time and effort to write the code to extract the data. Unless integration with EnCase's case management or other features is required, the examiner would most likely benefit from writing standard C or C++ code as EnScript adds an additional level of complexity without offering any additional features in navigating and extracting data from a file.

2.4 X-Ways and WinHex

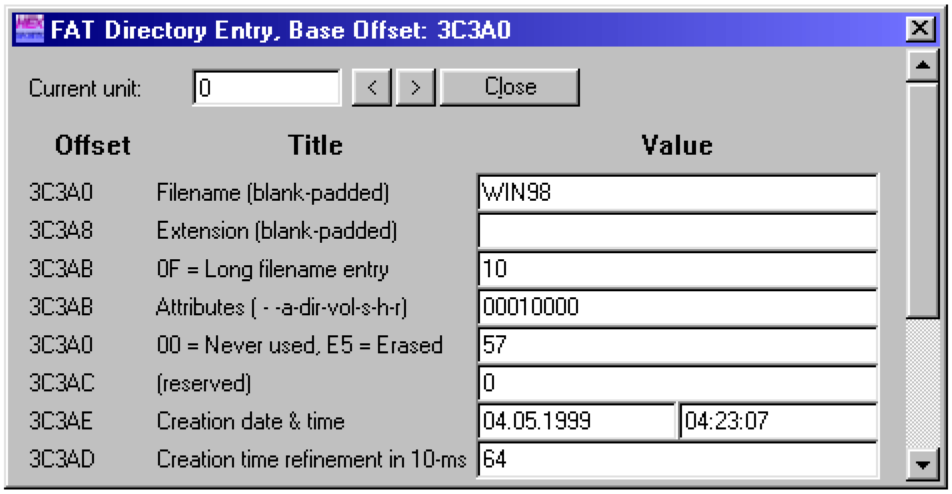
The WinHex forensic tool kit began as a universal hexadecimal editor for the Windows platform, but quickly expanded far beyond this scope. WinHex provides features to image disks, inspect data, attempt to locate and recover data either deleted or lost, and edit and create new data. As WinHex began to become much more than a simple HEX editor, X-Ways AG released its flagship forensic application X-Ways Forensics. X-Ways Forensics is a forensic toolkit based on WinHex. Much like other forensic workstations, X-Ways can assist an investigator with a wide range of tasks: case management, audit logging, file carving and data recovery. X-Ways provides an extension API called X-Tensions, which allows users to add additional features or automate existing features already in the toolkit.

One of X-Way's data interpretation features is of particular interest to this work. WinHex comes with built-in support for interpreting 20 common file types such as jpegs, zip archives, mp3s, and system data structures like Ext and FAT file system directory entries. This template feature is used to interpret and display binary data in dialog boxes for viewing, editing, or creating data. The templates can be used to examine data from either a file, an image of a disk sector, or virtual memory. The templates are saved in plaintext files and contain a textual description of the file or data structure for which the template is written.

Figure 5 contains the description for a FAT file system directory entry included with WinHex and X-Ways and how the associated template dialog box will appear in the program, respectively.

```
template "FAT Directory Entry"

// Template by Stefan Fleischmann
// X-Ways Software Technology AG

// To be applied to a sector of a FAT16 or FAT32 drive
// that contains a directory.  Not suitable for LFN
// (long filename) directory entries.

description "Normal/short entry format"
applies_to disk
multiple

begin
        char[8] "Filename (blank-padded)"
        char[3] "Extension (blank-padded)"
        hex 1           "0F = LFN entry"
        move            -1
        binary  "Attributes ( - -a-dir-vol-s-h-r)"
        goto            0
        hex 1           "00 = Never used, E5 = Erased"
        move            11
        read-only byte "(reserved)"
        move            1
        DOSDateTime     "Creation date & time"
        move            -5
        byte            "Cr.  time refinement in 10-ms units"
        move            2
        DOSDateTime     "Access date (no time!)"
        move            2
        DOSDateTime     "Update date & time"
        move            -6
        uint16  "(FAT 32) High word of cluster #"
        move            4
        uint16  "16-bit cluster #"
        uint32  "File size (zero for a directory)"
end
```

Figure 5 FAT Directory Entry Template Code [7]



FAT Directory Entry, Base Offset: 3C3A0

Current unit: 0

Offset	Title	Value
3C3A0	Filename (blank-padded)	WIN98
3C3A8	Extension (blank-padded)	
3C3AB	0F = Long filename entry	10
3C3AB	Attributes (- -a-dir-vol-s-h-r)	00010000
3C3A0	00 = Never used, E5 = Erased	57
3C3AC	(reserved)	0
3C3AE	Creation date & time	04.05.1999 04:23:07
3C3AD	Creation time refinement in 10-ms	64

Figure 6 FAT Directory Entry Template dialog box in X-Ways program [18]

As Figure 6 shows, the template is applied to the data opened in the WinHex viewer and the values are parsed and presented to the user in a neat and organized manner. The template dialog box provides the ability to view, edit and create new entries.

This template feature provides a system for extending support for additional file definitions. Users can create their own template files with a definition for any data structure they may be interested in viewing. X-Ways hosts a repository of user-generated template files for additional file types and data structures on their website. X-Ways encourages users to submit their own template files to be shared and hosted within the repository. Figure 7 shows an example template file for the description of the BMP file type written by Khomenko Volodymr [19].

```
...
begin
        section "BMP File Header"
                read-only char[2]      "BMP_ID"               // 00
                uint32  "File size"                           // 02
                uint32  "Reserved"                            // 06
                uint32  "ImageDataOffset"                     // 0A
        endsection

        section "BMP Info Header"
                uint32  "HeaderSize"                          // 0E
                uint32  "Width"                               // 12
                uint32  "Height"                              // 16
                uint16  "Planes"                              // 1A
                uint16  "BPP"                                 // 1C
                uint32  "CompessionMethod"                    // 1E
                uint32  "ImageSize"                           // 22
                uint32  "XPixelsPerMeter"                     // 26
                uint32  "YPixelsPerMeter"                     // 2A
                uint32  "PaletteSize"                         // 2E
                uint32  "ColorsImportant"                     // 32
        endsection

        section "Palette(If PaletteSize=0 then no palette)"
                numbering 0

                {
                        byte "B[~]"
                        byte "G[~]"
                        byte "R[~]"
                        byte "A[~]"

                } [PaletteSize]
        endsection
end
```

Figure 7 Template description for a BMP file [19]

Figure 8 shows the specifications of the BMP file type from Microsoft [20].

```
typedef struct tagBITMAPFILEHEADER
{
  WORD  bfType;
  DWORD bfSize;
  WORD  bfReserved1;
  WORD  bfReserved2;
  DWORD bfOffBits;
} BITMAPFILEHEADER,
*PBITMAPFILEHEADER;
```

```
typedef struct tagBITMAPINFOHEADER
{
  DWORD biSize;
  LONG  biWidth;
  LONG  biHeight;
  WORD  biPlanes;
  WORD  biBitCount;
  DWORD biCompression;
  DWORD biSizeImage;
  LONG  biXPelsPerMeter;
  LONG  biYPelsPerMeter;
  DWORD biClrUsed;
  DWORD biClrImportant;
} BITMAPINFOHEADER,
*PBITMAPINFOHEADER;
```

Figure 8 Bitmap file header and info header specifications [20].

As Figure 7 and Figure 8 show, the WinHex template file for the BMP file type are true to the file specification and it is just as easy to read and comprehend the type, size and meaning of each field from the WinHex template as from the BMP file specification.

Figure 5 and Figure 7 give an example of the level of complexity and breadth of the description language used in the template files. The language used in the template is barebones and basic. It supports all common data types: 16 and 32-bit integers and floating points, booleans, hex values, characters and strings. Lines such as

```
uint32      "Width"
```

perform 3 actions: reads 4 bytes of data from the current position, stores the value in a variable named Width as an unsigned 32-bit integer, and move the current position into the data 4 bytes forward. The language provides the ability to move backwards and forwards any number of bytes through the data using the move keyword.

```
move            -1
move            1
```

The preceding lines of code move the position in the data backwards and forwards one byte respectively. This allows the user to move past uninteresting data fields, or for certain fields to be read and interpreted in different ways and stored in different variables. There is also the `goto` keyword, which can be used to seek to various offsets within the data structure. A `goto` can be used by providing a variable which has been read before such as:

```
uint16          "Example"
goto            "Example"
```

This will move the position pointer to the offset read into the Example variable, relative to the start of the data structure.

The language has a minimal amount of logic and control flow keywords. It supports conditionals and simple loops. The following code excerpt from X-Way's template for ZIP archives demonstrates both.

```
…
section "Compressed file local headers"
    numbering 0
    {
        section "File header"
            hex 4 Value

            IfEqual Value 0x504B0304
                move -4
            Else
                ExitLoop
            EndIf

            hex 4           "ZIP local file header signature"
            hex 2           "Version needed to extract"
            hex 2           "General purpose bit flag"
            hex 2           "Compression method"
            DOSDateTime     "Last mod file date/time"
            hex 4           "CRC 32"
            uint32          "Compressed size"
            uint32          "Uncompressed size"
            uint16          "Filename length"
            uint16          "Extra field length"
            string          "Filename length"  "File name"
            hex             "Extra field length"      "Extra field"
            move            "Compressed size"  // to end of file data
        endsection
    }[100]
Endsection
…
```
Figure 9 Except from ZIP archive X-Way template [21]

The instructions between the curly brackets are executed 100 times, as denoted by the [100] following the closing bracket, unless the loop is exited prematurely by the `Else` statement if the value read into Value does not equal 0x504B0304. This template excerpt is comparable to the specified structure of a ZIP archive file's local file header. Figure 10 shows the actual specification of a ZIP file header for each contained file [22]. Both the specification and the template file are similarly structured and easy to read. The template file only adds a small amount of complexity to the description for control flow and loop execution; for simple control flow such as this, the added complexity is not very distracting from the overall structure of the data.

```
File header:

      local file header signature     4 bytes
      version needed to extract       2 bytes
      general purpose bit flag        2 bytes
      compression method              2 bytes
      last mod file time              2 bytes
      last mod file date              2 bytes
      crc-32                          4 bytes
      compressed size                 4 bytes
      uncompressed size               4 bytes
      file name length                2 bytes
      extra field length              2 bytes

      file name (variable size)
      extra field (variable size)
```

Figure 10 ZIP local file header specification [22].

The language does have its limitations. It is more akin to a simple scripting language than a file description language; each line is a command to either read bytes of data, compare data, or move the current cursor position. Complex templates require lots of logic and control flow; the overall structure of the data can get lost in the details.

The language also does not offer any level of abstraction. No compound data structures can be declared and used to make the template more human readable and accessible.

The final limitation of the language is its linear format. It becomes hard to read and comprehend the full scope and layout of the data structure. With only two options for control flow and no support for lists or arrays, some template descriptions can become unnecessarily complex for the human reader.

3. File Description Language

One main objective of this project is to address the problems and short comings of the languages used by other available tools. Other tools like EnCase or X-Ways have languages that lose focus on the data being interpreted. They either lack power, making simple descriptions of files convoluted and hard to read, or contain too many features and the big picture of the structure being described is lost. Some languages, like EnCase's EnScript, were not originally designed with this function in mind; they were designed for tool automation or extension, and the feature of parsing a data structure was added later and forced into the syntax and styling of the language.

The tool developed through this research took advantage of a file description language being developed in part by Dr. Florian Buchholz at James Madison University. The language was designed to describe the layout of binary data within a file. As such, it is low-level, but remains intuitive. The language focuses solely on describing the structure of a binary file, and is not bloated with extraneous features outside of this scope. The syntax and grammar of the language creates a textual description of data that is human readable and easy to comprehend. Because of its flexibility and ease of use, the language offers an excellent medium to share and discuss file structures with others.

Support is provided for the following:

- All basic data types
- Declaring and creating higher level constructs for basic data types
- Lists
- Pointers (global and local)

- Logic and common flow control

The individual elements of the language are discussed in further detail in the following sections.

3.1 **Basic** Types

There are four basic types of data in the description language:

- boolean - a single bit of data
- uint - unsigned integer value of either 1, 2, 4, or 8 bytes in size (unit1, unit2, unit4,uint8)
- sint - signed integer value of length 1, 2, 4, or 8 bytes in size (sint1, sint2, sint4, sint8)
- char - 1 or 2 byte character value (char1, char2)

The following line of code shows how these four basic types are assigned to variables.

```
V1 := sint4;
```

This line creates a variable named V1 assigning the type of a 4 byte signed integer. Simple file types can be fully described using only these basic types as Figure 11 below shows.

```
SIMPLEFILE := {
    Field1          := sint4;
    Field2          := uint2;
    Flag            := boolean;
    SecondFlag      := boolean;
    Field5          := char1;
}
```

Figure 11 Simple file description using only basic types

The preceding example is a complete description of a very simple file type that holds two integer values, two booleans and one character value in that order. Files with large lists of integers or strings of characters would be excruciatingly arduous to both compose and to comprehend. Since files this simple rarely exist, the description language can accommodate the need for more complex data structures.

3.2 Declarations

As preceding discussion on basic types showed in its simple example, a declaration can assign a type to a name. There are expanded declarations which that for the inclusion of descriptions, a name, permissible values and common values for the data type. For example, Figure 12 shows how the declaration of Field1 in the earlier file description could be expanded.

```
SIMPLEFILE := {

    Field1 := {
        sint4;
        description := "A description of the data field.";
        name := "a more explicit name for the field";
        permissible := [1,2,3];
        common := [1,3];
    }

}
```
Figure 12 Expanded variable declaration

An expanded declaration contains the field's type, descriptions, name, permissible and common values surrounded by curly brackets. The field type is the only required entry.

The description and name entries in an expanded declaration are strings allowing for a more accurate and descriptive name for the data and a human readable description of what the data represents.

The permissible entry allows the author to list all possible values that are legitimate possibilities for the field. Similarly, the common entry offers a way to express common values likely to be found in this field.

3.3 Structs

The description language, like many modern programming languages, offers a method to create complex structures of data using the basic types and declarations as building blocks. Similar to many C style languages, a struct in the description language is a method of grouping together various declarations into one logical unit. In this description language they are declared as shown in Figure 13.

```
Identifier := {
    declaration_1;
    declaration_2;
    …
    declaration_n;
}
```

Figure 13 Definition of a struct

Figure 13 above demonstrates how a struct is merely a collection of declarations, either simple, extended, or a mixture of the two. After a struct has been declared, it can be used in the declaration of other structs as well, as seen below.

```
struct1 := {
    v1 := uint2
    v2 := sint1
}

struct2 := {
    v1 := struct1;
    v2 := boolean;
}
```

Figure 14 Using structs in declarations

Individual elements of a struct can later be addressed using dot style notation.

```
Struct1.v1
```

3.4 Lists

A common feature in many file types and data structures is the repetition of data types in long running lists. These lists in the description language are similar to arrays in many programming languages.

Continuing with the simple file type example, the Field2 declaration could be expanded to be a list of several uint2 types:

```
SIMPLEFILE := {
     …
     Field2 := uint2[5];
     …
}
```

Figure 15 Declaration of a list

As with arrays in most modern programming languages, after a list has been declared, individual elements can be addressed using indexing. Given the declaration above, `Field2[2]`accesses the second uint2 value in the Field2 list.

Since a list of higher level structures is a common occurrence in more complex file types, lists can also be used when the type of a declaration is not a basic type, but rather a user-defined struct or any complex type such as lists, conditionals, or char or bit fields.

3.5 Conditionals

Another common paradigm in file types is the ability for field values to affect others. For instance, a one bit boolean field can be used as a flag and if true the next field in the file will be a char2 list otherwise the field will be a uint1. An example such as this is not an uncommon occurrence, and the description language offers a means to express this.

Conditionals in the language allow for the assigned type in a declaration to depend on the value of another declared field.

```
SIMPLEFILE := {
    Field1 := uint1;
    Field2 := case(Field1) {
        0:  uint1;
        1:  uint2;
        2: anotherDeclaredStruct;
        else: char2;
    }
}
```
Figure 16 Conditional declaration

In this example, the value in the Field1 variable determines the declared type of the Field2 variable. The list of cases can be as long and diverse as the author needs and can be followed by an optional else case, which will be the default type if the value does not match any of the cases.

3.6 Bitfields and Charfields

Many file types contain continuous lists of individual single bits. A collection of flag bits in a file header is an excellent example. Although this pattern could be expressed in the description as a list of boolean types, the language offers a more concise approach using the bitfield type. Bitfields are a sequence of bits of a given size with the ability to provide an optional name and description fields for each bit. Their declaration follows either of the two syntaxes:

```
      identifier := bitfield[size];
      identifier := bitfield(size, name_0, description_0, ...,
                      name_(size-1), description_(size-1));
```

Where size is the number of desired bits. In the second declaration, for each bit in the bitfield, a name and a description must be provided as strings. Figure 17 provides an example of a bitfield declaration.

```
SIMPLEFILE := {
    Bfield1 := bitfield[5];
    Bfield2 := bitfield(5, flag1, "a description",
                           flag2, "flag 2's description.",
                           flag3, "flag 3's description.",
                           flag4, "flag 4's description.",
                           flag5, "flag 5's description." );

}
```

Figure 17 Declarartion of bitfields

Likewise, a list of character types grouped together as a string is also a common feature in many file types. As with bitfields, this feature could be described in the language using lists of char1 or char2 types, but again, this is not as readable and easily understood by the user. The language therefore offers another type to help in the declaration of strings: a charfield. A charfield is a sequence of char types of a given size and an optional encoding. Charfields can be declared in a number of ways:

```
identifier := charfield(type, size, encoding);
identifier := charfield(type, size);
identifier := charfield(type, character, encoding);
identifier := charfield(type, character);
//where type must be char1 or char2
//instead of specifying a size, the latter specify a terminating
character, which must be of type type.

The following short syntax is permissible:

identifier := char1[size]; // ASCII charfield of given size
identifier := char2[size]; // Unicode charfield of given size
```

3.7 Language Features Left Out of Initial Research

The following features of the language were chosen not to be supported for this initial research. Even without their support included in the parser, the features discussed above provide a wide set of possible descriptions, and many common file types can be adequately described using only them. It is necessary to at least mention the skipped language features which will hopefully make it into future versions of the parser.

3.7.1 Lists of Unknown Size

Lists can also be declared with an unknown size. This can occur in certain file types which have a list of data or some data structure of arbitrary size not explicitly shown anywhere else in the file. Declarations of lists of unknown size are similar to standard lists with the size omitted.

```
        identifier := type[];
```

3.7.2 Virtual Fields

A virtual list is an abstract concept that allows querying of list members, which are computed over an actual structure of the data type. For this, the actual data type needs to be described first (using a struct), and then a formula for computing a given member of the list needs to be given. Within the name space of the formula the identifier i can be used to reference the ith element.

```
        identifier := vlist(size, ctype, formula);
        identifier := vlist(ctype, formula);
```

3.7.3 Pointers

Pointers are similar in practice to pointers in C; they are numeric values that hold the location of a piece of data or structure. Pointers can be declared in one of two ways, either global or local. Global pointers require the absolute offset from the beginning of the file for the pointer, while local pointers are given the location of another identifier to which the offset is applied.

```
        identifier := globalp(size, type, address);
        identifier := localp(size, type, offset);
        identifier := localp(size, type, identifier, offset);
```

4. Development

This chapter discusses the Binary Analysis Tool (BAT) developed for this thesis. The BAT takes as input a file description and uses this to build a syntax tree representing the structure of the binary file type described. The tool then parses a binary file and interprets the data according to structure of the syntax tree. The data parsed out of the binary file is presented to the user in an organized manner for easier analysis. Figure 18 shows an overview of the BAT.

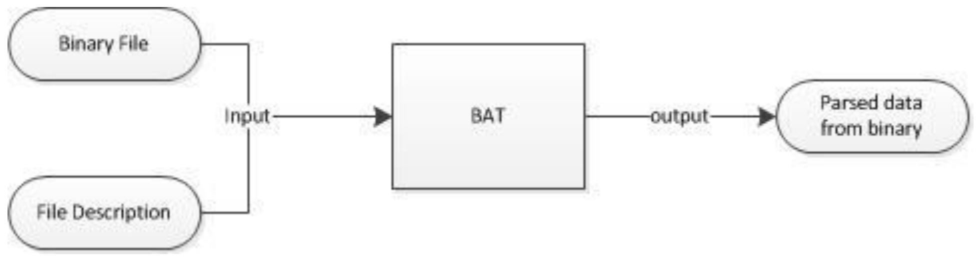


Figure 18 Overview of the BAT.

The tool performs three distinct tasks during execution:

1. Break the file description received as input into a stream of individual tokens
2. Construct a syntax tree based on the token stream.
3. Parse the binary file and present the data to the user in a readable format.

The tool is divided into three components: the tokenizer, the parser, and the binary parser. Each module is responsible for one of three tasks. The tokenizer component is responsible for breaking the file description into tokens which can be used by the parser to help build the syntax tree. The parser module's role is to interpret the token stream produced by the tokenizer using the grammar of the file description language and builds the syntax tree. The final component, the binary parser, uses the syntax tree created by the parser to read and interpret the data within a binary file. This data is presented to the user in an organized and readable manner. Figure 19 shows an overview of the execution process of the BAT's individual modules.

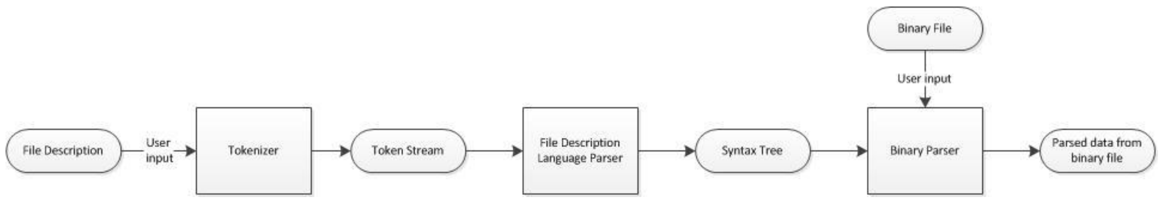


Figure 19 Overview of the BAT execution process

The following sections of this chapter discuss in further detail each phase of the BAT's development. Section 4.1 discuss the tools and development environment chosen to implement the tool. Section 4.2 discusses the Lexical Analysis phase and the implementation of the tokenizer. Section 4.3 discusses the development of the file description language parser and the construction of the syntax tree. Section 4.4 discusses the implementation of the binary parser module that uses the syntax tree to parse a binary file.

4.1 Development Environment and Tool Research

The first step was to research and decide upon the development platform, environment, and tools (if any) that would be used to create this program. Given the importance of lexical analysis and parsing the tools Flex and Bison [23] were considered. The Flex and Bison tools are free GNU tools for the production of lexical analyzers and parsers respectively. They are decedents of the original LEX and YACC tools developed in 1975 [24].

During research, an open source Python library named PLY [25] was found. PLY, which stands for Python LEX and YACC, is an implementation of the LEX and YACC tools in Python. The PLY library slightly deviates from the LEX and YACC standards, but keeps the general development constructs and layout the same. The changes take advantage of Python-specific improvements, such as returning tuples or objects from the tokenizer. Because of the similarities of the tools PLY is straightforward to learn if one has prior knowledge of LEX and YACC as well as Python.

Due to existing knowledge of the LEX and YACC frameworks and personal preference for Python, I decided to use Python with PLY as my development language. The Python interpreter is becoming standard on all Linux and MAC OSX distributions and easily accessible on Windows machines. This choice makes the BAT more platform independent than using FLEX and Bison. FLEX and Bison are C-based tools and would need to be compiled and distributed for each targeted platform.

4.2 Lexical Analysis

The first stage of developing any parser, interpreter, translator, or compiler tool is called lexical analysis or scanning. The following sections discuss what the role of a lexical analysis is, how its function is accomplished, and how the lexical analyzer is implemented in this work.

4.2.1 Lexical Analysis Background, Terms, Definitions and Tools

Aho et al. [26] describe the role of a lexical analyzer as reading a character stream from the input, grouping the characters together into lexemes, and producing a stream of tokens from these lexemes as output to the parser. The authors also offer the following terms and definitions, which I will adhere to for my discussion:

- **Token** – a name representing a lexeme matched by a pattern, may also be paired with an associated attribute. For example, the token for an identifier might be paired with the actual string of the identifier name.

- **Pattern** – an informal description of the set of lexemes corresponding to a given token.
- **Lexeme** – a sequence of characters that matches a particular token based on a pattern. It is possible for a token to have several lexemes.

Table 1 demonstrates the relation between these three terms with examples from the file description language.

Token	Pattern	Lexeme
Case	Characters c, a, s, e	Case
Id	A letter followed by zero or more letters, numbers, or underscores.	Field1, file_header, Xy2Z
hexadecimal_number	Characters 0, x, followed by one or more digits 0-9 or letters A-F	0xFF, 0x10, 0x0A

Table 1 Token, Pattern, and Lexeme examples

A lexical analyzer can be developed by hand for any language with a list of language tokens and pattern rules for these tokens. Scanners and lexers generally work by scanning input one character at a time and checking it against a list of patterns. However, over the past few decades many tools have been developed to help in the creation of lexical analyzers. As discussed in the previous section, Lex (and its descendent Flex) were among the first developed and are still widely used today. There are many other tools available for lexical analysis, many of which, including the PLY library used by the BAT, are based upon the design of Lex. Most of these tools use regular expressions as the means of defining the patterns to match input against, and Lex, Flex, and PLY are no exceptions [23].

Although Lex and Flex are based on C-style syntax and PLY is based on Python, the structure of a lexical analyzer specification using the tools are similar. A program written with these tools is generally divided into three sections: declarations, transition rules, and auxiliary functions. The declarations section contains any constants, variables and most importantly, any regular expression definitions needed for the lexer. The transition rules section contains a list of new regular expressions or predefined expressions and actions to be carried out when the pattern is matched. Finally, the auxiliary section is a place for any arbitrary code or helper functions the author might need to assist with the actions taken in the transition section.

4.2.2 Language Tokens

To begin work on the lexical analyzer, I first broke down the file description language into the individual token types required. As the file description language conforms to most programming language archetypes, I was capable of following the general token outline described by Aho et al. for the creation of my token list [26].

1. A token for each keyword or reserved word.
2. Tokens for each operator.
3. A token to represent all identifiers
4. At least one token for constants.

5. Tokens for each punctuation symbol.

Using these rules, I constructed my token list and the patterns that would be used to match individual lexemes to them. Appendix A shows the entire token list the first version of the parser supports. The table contains the symbol, keyword, or regular expression used to match each token in the list.

The table of tokens shows that there is one token for each of the language's reserved words; this includes the basic types, case, else, bitfield, charfield, description, name, permissible, and common. There is also one token for each of the language's operators and one for each punctuation symbol. The pattern for all of these tokens is the reserved word itself or the symbol.

The remaining tokens, BIN_NUMBER, OCT_NUMBER, DEC_NUMBER, HEX_NUMBER, STRING, COMMENT, and ID all require a more complex regular expression for its matching pattern.

Tokens for the unsupported features such as the globalp and localp keywords are also included in the tokenizer implementation. This added no substantial work effort to the implementation of the lexer and was done so that future work on the tool will only need to focus on the parsing module of the tool. In the current version of the parser, if one of these unsupported tokens is used the parser will throw an unexpected token error and the parser will stop. It is unnecessary for the parser to continue because the syntax tree produced will no longer be an accurate representation of the binary file without support of these features.

4.2.3 Lexical Analyzer Implementation

Using the token list discussed in the previous section, a lexical analyzer for the file description language was developed. The full source code for the lexical analyzer written in python using the PLY library can be found in Appendix B.

Using PLY to assist in the creation of a lexical analyzer is similar to creating lexers with Lex or Flex; there are, however, some structural and syntactical differences. In PLY, the first task that must be done when creating a lexical analyzer for a grammar is to create a list of reserved words within the language. As seen in the full source code and Figure 20, the list *reserved* is a list of all the keywords in the file description language. The last four reserved words in the list (main, struct, little_endian, and big_endian) were added to the language during the development of this thesis. Their purposes will be discussed in further detail later in this chapter.

```
reserved = (
       'BITFIELD', 'CHARFIELD', 'GLOBALP', 'LOCALP',
       'CASE', 'ELSE', 'DESCRIPTION',
       'NAME', 'PERMISSIBLE', 'COMMON', 'UINT1', 'UINT2', 'UINT4', 'UINT8',
       'SINT1', 'SINT2', 'SINT4', 'SINT8', 'CHAR1', 'CHAR2', 'BOOLEAN',
       'STRUCT', 'MAIN', 'LITTLE_ENDIAN', 'BIG_ENDIAN'
        )
```

Figure 20 list of reserved words

A second list also needs to be created to hold all the tokens of the grammar. As seen in the code excerpt in Figure 21, the *tokens* list is the reserved words list plus the additional tokens for the language.

```
tokens = reserved + (
       # Literals
       'BIN_NUMBER', 'OCT_NUMBER', 'DEC_NUMBER', 'HEX_NUMBER', 'STRING',
       # Operators
       'PLUS', 'MINUS', 'ASTERISK', 'FWDSLASH', 'PERCENT',
       'PIPE', 'ASSIGN', 'COLON', 'SEMI', 'COMMA', 'DOT',

       # Delimeters ( ) [ ] { }
       'LPARAN', 'RPARAN',
       'LBRACKET', 'RBRACKET',
       'LBRACE', 'RBRACE',

       'ID', 'NEWLINE',
       'COMMENT'
       )
```

Figure 21 delcaration of list of tokens

Once a list of tokens has been defined, the rules and regular expressions for matching these new tokens are defined in the next section of the PLY program. Any token added to the *tokens* list needs to be specified with a regular expression rule to match. Each rule is defined through declarations with the special prefix `t_`, which indicates it defines a token.

This can be done in two ways with PLY. The expression can be specified as a string for simpler tokens, such as the following example:

```
t_WORD       = r'[a-z]+'
```

If the matching is more complex or the programmer needs the tokenizer to take additional actions once matched, a new method can be defined.

```
def t_NUMBER(t):
            r'\d+'
            t.value = int(t.value)
            return t
```

The above example, taken from the PLY documentation [27], shows the matching rule for any decimal number. PLY reads all input in as strings so the observed value needs to be converted into a Python integer before returning the token.

Each token specified in this manner takes one argument: an instance of `LexToken`.

`LexToken` is a PLY object with the following attributes:

• `t.type`	The token type as a string.
• `t.value`	The observed value from the scanned input.
• `t.lineno`	The line number the token was found.
• `t.lexpos`	The position of the token, relative to the start of the file.

The default `t.type` will be the token type being specified; continuing with the `t_NUMBER` example, the `t.type` value will be set to the token type NUMBER by default. The values of the `LexToken` can be manipulated as necessary, including the token type, before being returned.

Figure 22 shows an excerpt of the declaration of many of the simple language tokens. The symbols of the language (i.e. '+','-', '|', etc.) are all simple regular expressions and no additional steps are needed by the tokenizer so the first method of defining the appropriate expressions was used. During execution, PLY scans the input file and when one of the declared regular expressions is matched, it will return the associated token. For example, in Figure 22 the t_PLUS token is declared to be the regular expression '\+'. If PLY finds the symbol + during execution, the t_PLUS token will be returned.

```
# Operators
t_PLUS          = r'\+'
t_MINUS         = r'-'
t_ASTERISK      = r'\*'
t_FWDSLASH      = r'/'
t_PERCENT       = r'%'
t_PIPE          = r'\|'

t_ASSIGN        = r':='
t_COLON         = r':'
t_SEMI          = r';'
t_COMMA         = r','
t_DOT           = r'\.'
```

```
t_LPARAN     = r'\('
t_RPARAN     = r'\)'
t_LBRACKET   = r'\['
t_RBRACKET   = r']'
t_LBRACE     = r'{'
t_RBRACE     = r'}'
```

Figure 22 declaration of token regular expressions

Like FLEX, PLY adheres to two rules to avoid ambiguity when matching tokens. First, the longest string possible is matched. Second, when there is a tie, the first specified rule is used [23]. For example, the regular expressions "0[1-9]+" and "[0-9]+" will match the token 07. In this case, PLY will return which ever rule is specified first.

Several of the tokens were slightly more complex and required the second approach of matching. For example, the method *t_ID* shown in Figure 23 is used to match all identifiers. The first line of any token function in PLY must be the regular expression used to match the token. The *t_ID* function matches the regular expression [a-zA-Z_][a-zA-Z_0-9]*, which will match any word starting with an upper or lower case letter or an underscore followed by any number of letters, numbers, or underscores. Before the method returns it needs to check if the matched word is in the reserved list and if so returns that token value instead of an ID.

```
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    if t.value.upper() in reserved:
        t.type = t.value.upper()
    return t
```

Figure 23 t_ID *function for matching identifiers or keywords*

The final method *t_error* is a requirement of PLY that will execute if the tokenizer encounters a character that cannot be matched to any of the rules above. This lexical analyzer prints an error message alerting the user of the illegal character and the line number it was found on and skips it all together. Improvements to this error handling function can be made in the future versions of the lexical analyzer.

```
def t_error(t):
    print("Illegal character %s" % repr(t.value[0]))
    t.lexer.skip(1)
```

Figure 24 error handling function required by PLY.

4.3 File Description Language Parser

The second phase in of the BAT is syntax analysis or parsing. The following sections discuss the role of the parser, how it accomplishes this role and how the parser for this tool is implemented.

4.3.1 Grammar

The first step of developing the file description parser was to finalize the grammar of the file description language. Although tools such as YACC, BISON and PLY have support for GLR parsers, from my experience it is easier to work with LALR(1) parsers. GLR grammars are ambiguous by default and require extra work from the parser to determine what is actually being expressed in the input. When a GLR parser encounters a conflict due to the ambiguous grammar, it splits the job and continues along both possible parse paths in parallel. The added complexity to resolve these conflicts can make the parser slow. Unless all possible conflicts are known beforehand and resolved appropriately, unexpected errors can be common in GLR parser [23]. GLR parsers can, however, support a much larger variety of grammars.

The grammar for the description language is not ambiguous enough that a GLR parser is needed; a LALR(1) parser will correctly handle the language. By default PLY will attempt to build a LALR(1) parser unless specified otherwise, so no special action is needed.

The grammar productions used for the first version of the parser can be found in Figure 25. As stated earlier in this chapter, four reserved words were added to the language during the course of this work. The **little_endian** and **big_endian** keywords were added to denote the endianess of the file. Specifying the endianess is optional, but must be the first field of the description if included. If no endianess is specified, the binary parser will assume the file is little endian. The **main** keyword was added to denote which struct should be interpreted as the start of the binary structure. There must be one and only one main struct and it must be the first struct defined in a description. The **struct** keyword was added to further differentiate the struct and declaration productions of struct.

description	->	*endianess main_struct struct_list*
endianess	->	**LITTLE_ENDIAN**
		BIG_ENDIAN
		empty
main_struct	->	**MAIN** *struct*
struct_list	->	*struct_list struct*
		struct
struct	->	**STRUCT ID := {** *decl_list* **}**
decl_list	->	*decl_list decl*
		decl
decl	->	**ID :=** *type* **;**
		ID := { *extendedDecl* **}**
extendedDecl	->	*type* **;** *extendedOP_list*
extendedOp_list	->	*extendendOp_list extendedOp* **;**
		extendedOp **;**
extendedOp		-> **DESCRIPTION := STRING**
		NAME := STRING
		PERMISSIBLE := STRING

		COMMON := STRING
type	->	**UINT1**
		UINT2
		UINT4
		UINT8
		SINT1
		SINT2
		SINT4
		SINT8
		CHAR1
		CHAR2
		BOOLEAN
		ID
		list
		charfield
		bitfield
		conditional
list	->	*type* **[** *number* **]**
		type **[** *reference* **]**
charfield	->	**CHARFIELD (CHAR1 ,** *number* **, STRING)**
		CHARFIELD (CHAR2 , *number* **, STRING)**
		CHARFIELD (CHAR1 , *number* **)**
		CHARFIELD (CHAR2 , *number* **)**
		CHARFIELD (CHAR1 , ID , STRING)
		CHARFIELD (CHAR2 , ID , STRING)
		CHARFIELD (CHAR1 , ID)
		CHARFIELD (CHAR2 , ID)
		CHAR1 [*number* **]**
		CHAR2 [*number* **]**
bitfield	->	**BITFIELD [** *number* **]**
		BITFIELD [*bit_list* **]**
bit_list	->	*bit_list bit_def*
		bit_def
bit_def	->	**ID , STRING**
conditional	->	**CASE (** *reference* **) {** *case_list* **}**
case_list	->	*case_list case*
		case
case	->	*case_value* **:** *type* **;**
		ELSE : *type* **;**
case_value	->	*number*
		STRING
number	->	**BIN_NUMBER**
		OCT_NUMBER
		DEC_NUMBER
		HEX_NUMBER

reference	->	**ID .** *reference*
		ID

Figure 25 File description language grammar

There are six shift/reduce conflicts in the above grammar. A shift/reduce conflict is caused when PLY has the option to either reduce the current stack of input tokens or to continue by shifting the next input token onto the parsing stack. The conflicts in this grammar come from the recursion in the *reference, case_list, bit_list, extOp_list, decl_list,* and *struct_list* productions.

By default, PLY will resolve shift/reduce conflicts by shifting the next token [25]. This solution will correctly resolve the conflicts in this grammar and no special actions or precautions are required.

Before development began, a few caveats to the language were introduced. Firstly, structs are to be treated with a global scope. This allows for easy referencing of declarations between structs, but does limit the user by forcing all structs to have unique names. Declarations defined within the same struct must to be named uniquely as well.

The second caveat pertains to declaration referencing. Declarations are treated in a top-down manner during parsing. This limitation restricts references to only the declarations that have been declared earlier in the file description. The following example would not be a valid description because of this caveat:

```
main struct file := {
        x := uint1[y];
        y := uint1;
}
```

4.3.2 Representing the Binary Structure

Using Python or other languages with support for object oriented-programming provides a useful feature when constructing a parser. As the YACC module of PLY uses the earlier defined lexical analyzer to tokenize the input and parse the language based on the grammar, the parser can construct a tree of node objects to represent the structures being defined.

The following node classes were defined for the construction of this syntax tree:

- StructNode
- DeclareNode
- TypeNode
- CharfieldNode
- BitfieldNode
- ListNode
- BasicTypeNode
- ConditionalNode

Figure 26 contains the definition of the StructNode. Instance of this node type contains two attributes: a name, and a list of DeclareNodes. The list is initialized empty in the node's constructor and as the parser parses the struct's declarations they are appended. During instantiation, each struct node adds itself to the symbol table. The symbol table is a global dictionary used to store each defined struct using its name as the key. The symbol table is used again after the entire description has been read to update any pointer references to the correct structNode.

```
class StructNode(Node):

    def __init__(self, name):
        self.name = name
        self.declarations = []


        SymbolTable[name] = self
```

Figure 26 StructNode definition

The definition for a DeclareNode is shown in Figure 27. A DeclareNode is constructed for each declaration read by the parser. These nodes contain 6 attributes: name, longname, description, permissible, common, and type. The name attribute is the identifier associated with the declaration. Longname and description store the string for the name and description options if present in the declaration. Likewise, the permissible and common attributes holds the list of values if they are provided in the declaration. The final attribute of the node, type, hold a pointer to an instance of a TypeNode.

```
class DeclareNode(Node):
    def __init__(self, name, t):
        self.name = name
        self.longname = ""
        self.description = ""
        self.permissible = []
        self.common = []
        self.type = TypeNode(t)
```

Figure 27 DeclareNode definition

The TypeNode definition can be found in Figure 28. A TypeNode only contains one attribute: type. The type attribute is a pointer to one of several different possibly node types. The constructor for a TypeNode is sent one parameter by the parser; the parameter *t* will either be a fully constructed CharfieldNode, BitfieldNode, ListNode, ConditionalNode, or a string. First *t* is checked if it is among the list of basic type strings and, if so, a new BasicTypeNode is created. If *t* is not a basic type, it is checked if it is an instance of one of node objects and set appropriately. If none of these cases return true for *t*, then *t* must be a user defined struct. However, there is the chance this struct has yet to be defined as the parser is reading the input. The TypeNode puts itself into the global dictionary *patchTable* by using itself as the key and the struct name as the value. After the entire file description has been read by the parser, this will be updated with a pointer to the appropriate node type; this will be discussed in further detail later in this chapter.

```
class TypeNode(Node):
    def __init__(self, t):
        if t in basicTypes:
            self.type = BasicTypeNode(t)
        else:
            if ( isinstance(t, ListNode) or
                 isinstance(t, CharfieldNode) or
                 isinstance(t, BitfieldNode) or
                 isinstance(t, ConditionalNode) ):
                self.type = t

            else:
                #a struct. Might not be defined yet. type will be reset after
                #full description is read.
                self.type = t
                patchTable[self] = t
```

Figure 28 TypeNode definition

Figure 29 shows the definition of the CharfieldNode class. The CharfieldNode has 3 attributes: type, size and termChar. The type attribute is a pointer to a BasicTypeNode and will always be either a char1 or char2. The size of the CharfieldNode is set by the parser if the declaration of the charfield provided one. The termChar attribute is likewise set by the parser if one is provided in the declaration.

```
class CharfieldNode(Node):
    def __init__(self, t):
        self.type = BasicTypeNode(t)
        self.size = 0
        self.encoding = ""
        self.termChar = ""
```

Figure 29 CharfiledNode definiton

The BitfieldNode class definition can be found in Figure 30. The BitfieldNode is simple and only contains 2 attribute: size and bits. The size attribute is set by the parser as the number of bits in the bitfield. The bits attribute is a strings used to store the names of the individual bits if included in the file description.

```
class BitfieldNode(Node):
    def __init__(self, size):
        self.size = size
        self.bits = []
```

Figure 30 Bitfield definition

The definition for the BasicTypeNode can be found in Figure 31. The BasicTypeNode contains one attribute: type. The type attribute is a string, which will always be equal to one of the basic types. The BasicTypeNode also defines the function `getSizeInBits()`. This function returns size of the basic type in bits.

```
class BasicTypeNode(Node):

    def __init__(self, t):
        self.type = t

    def getSizeInBits(self):
        if self.type in ["uint1", "sint1", "char1"]:
            return 1 * 8
        elif self.type in ["uint2","sint2","char2"]:
            return 2 * 8
        elif self.type in ["uint4", "sint4"]:
            return 4 * 8
        elif self.type in ["uint8", "sint8"]:
            return 8 * 8
        elif self.type == "bool":
            return 1
        else:
            return 0
```

Figure 31 BasicTypeNode definition

The ConditionalNode definition can be found in Figure 32. ConditionalNodes contain only one attribute: conditions. The conditions attribute is a dictionary, which is initialized empty, but is filled with the possible conditions as the parser continues. The conditions are added to the dictionary by using the case value as the key and a TypeNode as the value.

```
class ConditionalNode(Node):
    def __init__(self):
        self.conditions = {}
```

Figure 32 ConditionNode definition

For example, if the following file description excerpt is being parsed, each case will be added to the conditions dictionary. The key 1 will be added to the dictionary with the value of a new typeNode constructed with the value 'uint1'. Likewise, the keys 4 and 'else' will be added with the new typeNodes constructed with the values 'uint4' and 'unit8' respectively.

```
x := case(y) {
    1: uint1;
    4: uint4;
    else: uint8;
}
```

4.3.3 Tables

There are two global tables referenced throughout the parser. The *symbolTable* and the *patchTable*. Both are instances of the built-in dictionary Python data type.

The *symbolTable* is used to store structNodes after they have been defined. The struct's name is used as the key and the instance of the structNode is stored as the value.

The *patchTable* is used to store typeNodes if they were defined referencing a user defined struct. There is the chance the struct has not been parsed at the time of the typeNode's creation. To address this issue, the typeNode is stored in the table as the key to ensure uniqueness and the referenced struct's name as the value. After the parser has completed parsing all the input, the *patchTable* is iterated over. For each typeNode it contains the pointer to the referenced structNode is updated with the correct value found in the *symbolTable*.

4.3.4 Parser Implementation

Using the grammar and the node structures discussed in the previous sections, a parser for the language was developed. The full source code for the parser using the PLY library can be found in Appendix C.

One of the original and most widely used tools for parser generation is YACC (Yet Another Compiler Compiler). YACC, and its more recent descendant BISON, contain a series of rule-action pairs. A rule is a series of symbols, tokens, or other rules.

Using PLY to assist in the creation of a parser is conceptually similar to writing a YACC or BISON definition. A PLY program's rules are defined through functions. Similarly to token definition in PLY's implementation of LEX, each grammar rule in the language needs to be defined through a function with the special prefix *p_*. Each function's docstring must contain the appropriate context-free grammar rule [25]. The following example shows the definition of the main_struct rule:

```
def p_main_struct(p):
    'main_struct : MAIN struct'
```

The docstring must contain the rule's name and the context-free grammar specification separated by a colon. The specification can contain any series of token types, literal symbols or other defined grammar rules. The main_struct rule's docstring, `'main_struct : MAIN struct'` specifies that a main_struct consists of the keyword token MAIN and a struct. The grammar specification for a struct must be defined elsewhere in the PLY program.

Each rule in PLY receives one parameter *p*. This parameter is a list of the values returned by each element of the rule. The following example taken from the PLY documentation demonstrates the mapping of the grammar symbols to *p* [25]:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

If the element is a token type, the value of the grammar element will be same as the *t.value* set by the tokenizer. For other non-terminal grammar elements, the value will be determined by the value placed in *p[0]* of the subsequent grammar rules. In the main_struct rule, for example, *p[1]* will contain the value 'MAIN' and *p[2]* will contain the return value of the struct rule defined later in the PLY program.

The body of a grammar rule function can contain any Python code the user would like. Typically, the body will contain instructions to perform any necessary syntax or semantic checking and to construct new node instances to be added to the syntax tree the parser is building.

As the parser is executing, each grammar rule associated with a node type is responsible for collecting the necessary information and constructing a new instance of the node. Each grammar rule returns the new instance of the node by setting the value of *p[0]*.

The following excerpt shows how the *struct* grammar rule constructs and returns a new structNode instance.

```
def p_struct(p):
    'struct : STRUCT ID ASSIGN LBRACE decl_list RBRACE'
    global currentDeclareList
    p[0] = StructNode(p[2])
    p[0].declarations = currentDeclareList
    currentDeclareList = []
```

The method creates a new struct node in *p[0]* by providing the value of the ID token to the constructor. The new structNode's declarations list is set to the current value of the global variable *currentDeclareList*. Like YACC, PLY uses the bottom-up parsing method [25], meaning that the *decl_list* non-terminal will have already been parsed by the time of this rule's execution. The *currentDeclareList* is used by the *decl_list* production to store pointers to each of the declareNodes created while parsing the current struct. Finally the function creates a new *currentDeclareList* for the next struct to be parsed.

As the following code shows, the *currentDeclareList* is used in the *decl_list* grammar rule to store the list of declarations of the struct currently being parsed:

```
def p_decl_list(p):
    '''decl_list : decl_list decl
                 |  decl  '''
    if len(p) == 3:
        currentDeclareList.append(p[2])
    else:
        currentDeclareList.append(p[1])
```

As mentioned in the previous section, if a declaration's type is a struct, there is the chance the struct has not been defined in the input at parse time. To address this issue, all type nodes with a struct type are placed into a dictionary called *patchTable* when the typeNode is created. The nodes use themselves as the dictionary key to ensure uniqueness and set the value to the name of the struct the type node should point to.

After the entire input has been read, the *treePatch* function is called to update any type nodes that are missing a pointer because the type was a structNode that may not have been defined at the time of its declaration. The function can be seen below:

```
def treePatch():
    for node in patchTable.keys():
        node.type = SymbolTable[patchTable[node]]
```

For each type node in the *patchTable*'s keys, the node's type field is updated with a pointer to the correct struct by looking it up in the symbol table using the struct's name stored in the *patchTable*.

4.3.5 Example Syntax Tree

This section illustrates the syntax tree produced by the parser by discussing a simple example. Figure 33 shows a description of a simple binary file type. Figure 34 shows a graphical representation of the resulting syntax tree after the parser has completed parsing the file description. The *file* and *footer* structNodes each contain a list of pointers for all of their respective declareNodes. The ConditionalNode contains a pointer to a typeNode for each possible case value defined in the description.

```
main struct file := {
        v1 := sint4;
        x := uint8[2];
        y := footer;
        cfield := charfield( char1, 12);
        c1 := case(y.f1) {
                1: uint1;
                2: uint2;
                3: uint4;
                else: uint8;
        };
}

struct footer := {
        f1 := uint1;
        f2 := uint2;
}
```

Figure 33 Simple file description used to demonstrate the syntax tree

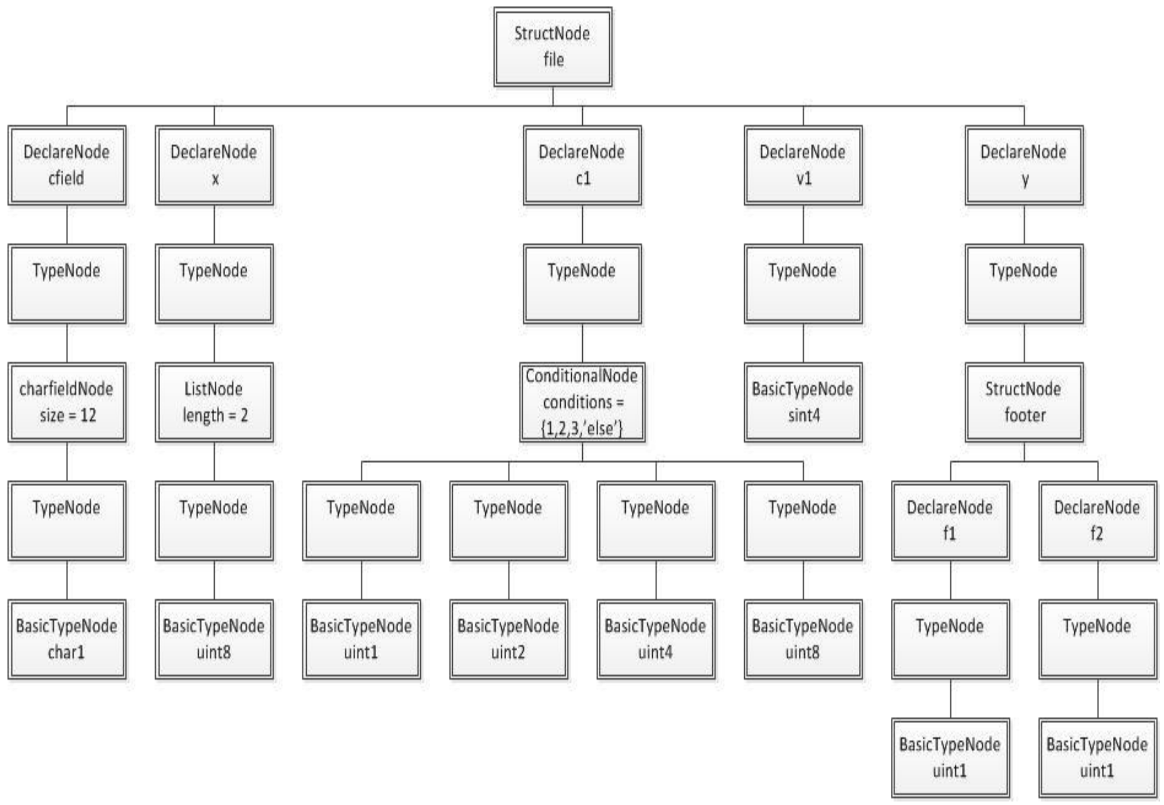


Figure 34 Syntax tree of the simple file type

4.4 Binary Parsing

Once the parser has completed its full pass of the input, the tree that correctly represents the file type described will be completed. During the parsing phase enough data is collected and can be used to create each node and correctly structure the tree so that a binary file can accurately be parsed while traversing the tree. The BinaryParser class was implemented to accomplish this task. This section discusses the implementation and function of the binary parser.

The binary parser is defined as follows:

```
class BinaryParser:
    def __init__(self, root):
        self.root = root
```

The constructor method of the parser takes as its single parameter a pointer to the root node of the earlier constructed tree. The binary parser class also defines the following functions:

- `def parse(self, filename):`
- `def parseStruct(self, structNode):`
- `def parseDeclaration(self, declNode):`
- `def parseType(self, tNode):`
- `def parseList(self, lNode):`
- `def parseBasicType(self, btNode):`
- `def parseCharfield(self, cfNode):`
- `def parseBitfield(self, bfNode):`
- `def parseConditional(self, cNode):`

The **parse** takes one parameter: the filename of the binary which is to be parsed. As the code excerpt below shows, the function is relatively simple. It first opens the file as read-only and in binary mode. The function then starts the parsing traversal of the tree by passing the root node to the **parseStruct** function.

```
def parse(self, filename):
        self.binaryFile = open(filename, "rb")
        self.parseStruct(self.root)
```

The **parseStruct** function is passed an instance of a struct node as a parameter and enters a loop over the list of declaration nodes contained inside the struct node. For each declaration node in the struct, a pointer to node is passed along to the **parseDeclaration** functions. The following code shows how this is done:

```
def parseStruct(self, sNode):
        print(sNode.name + " :\n")

        for decl in sNode.declarations:
            self.parseDeclaration(decl)
```

Each declaration node passed to the **parseDeclaration** function simply reports the declaration's name and continues the tree traversal by sending a pointer to the declaration node's type node to the **parseType** function.

```
def parseDeclaration(self, dNode):
        print(dNode.name)
        self.parseType(dNode.type)
```

The **parseType** function receives as a parameter a pointer to a typeNode. The function performs checks to determine which type the node points to; it's type will either be a BasicTypeNode, StructNode, ListNode, CharfieldNode, BitfieldNode, or a ConditionalNode. Depending on the outcome of these checks, the traversal of the tree continues by passing the pointer to the correct parsing function, as the following excerpt shows:

```
def parseType(self, tNode):
        if isinstance(tNode.type, BasicTypeNode):
            self.parseBasicType(tNode.type)

        elif isinstance(tNode.type, StructNode):
            self.parseStruct(tNode.type)

        elif isinstance(tNode.type, ListNode):
            self.parseList(tNode.type)

        elif isinstance(tNode.type, CharfieldNode):
            self.parseCharfield(tNode.type)

       elif isinstance(tNode.type, BitfieldNode):
            self.parseBitfield(tNode.type)

        elif isinstance(tNode.type, ConditionalNode):
            self.parseConditional(tNode.type)
```

The **parseList** function is passed as a parameter an instance of a ListNode. The function enters a loop that will execute *lnode.length* times. Each pass of the loop will perform similar checks as the **parseType** function to determine which parsing function to call to continue parsing the binary.

```
def parseList(self, lNode):
        string = ""
        for i in range(lNode.length):
            string += "\n[" + str(i) + "] :\t"

            if isinstance(lNode.type, BasicTypeNode):
                string += self.parseBasicType(lNode.type)

            elif isinstance(lNode.type, StructNode):
                string += self.parseStruct(lNode.type)

            elif isinstance(lNode.type, CharfieldNode):
                string += self.parseCharfield(lNode.type)

            elif isinstance(lNode.type, BitfieldNode):
                string += self.parseBitfield(lNode.type)

            elif isinstance(lNode.type, ConditionalNode):
                string += self.parseConditional(lNode.type)
```

ConditionalNodes are parsed by the **parseConditional** function. To parse a conditional node, the value read into the declaration being referenced must first be looked up. The conditionalNode's *reference* attribute is a list of declaration names as strings. These strings are used as keys into the *globalDecTable* to look up the value parsed from the binary. The value is then checked to see if it is a member of the node's list of conditions. If so, parsing continues by passing the typeNode associated with the case value to the **parseType** function. If the value is not found in the conditions list and no else case was defined, an error is reported to the user.

```
def parseConditional(self, cNode):
        #get value stored
        value = self.globalDecTable
        for ref in cNode.reference:
            value = value[ref];
```

```
#check if value is in conditions list
if value in cNode.conditions:
    self.parseType(cNode.conditions[value])
elif 'else' in cNode.conditions:
    self.parseType(cNode.conditions['else'])
else:
    print("error: case not found.")
```

As the tree is traversed by these parsing methods, eventually the tree will reach one of the three types of leaf nodes: BasicTypeNode, CharfieldNode, and BitFieldNode. These three parsing methods are the only functions which actually perform any reading of the binary file.

Reading values of more than one byte introduces the issue of endianness. As stated earlier, the keywords **litte_endian** and **big_endian** were added to the file description language to denote the file's endianness. When reading values greater than one byte the parser will interpret the bytes according to the stated endianness, or if no endianness was provided, little endian is assumed.

```
def parseBasicType(self, btNode):
        print(btNode.type)
        value = self.binaryFile.read(int(btNode.getSizeInBits() / 8))

      if btNode.type in ["uint1","uint2","uint4","uint8",

                         "sint1","sint2","sint4","sint8"]:

          if   btNode.type == "uint1":
                if self.BigEndian:
                    value = struct.unpack('>B', value)[0]
                else:
                    value = struct.unpack('<B', value)[0]

          elif btNode.type == "sint1":
                if self.BigEndian:
                     value = struct.unpack('>b', value)[0]
                 else:
                    value = struct.unpack('<b', value)[0]
```

As seen in the partial excerpt of the **`parseBasicType`** function above, the program performs a read operation on the binary file and reads one, two, four, or eight bytes from the binary file depending on the size of the type being parsed. As discussed earlier, the BasicTypeNode defines a function named getSize() that returns the size of the type in bits. The read operation in Python, however, operates on the byte level, thus the size is divided by eight before being sent as a parameter to the read function.

This introduces a problem when attempting to read boolean values, which are only one bit in size. At this time, the binaryParser class does not support the reading of single bits of data. Python is not alone in operating on the byte level; very few programs are capable of operating at the bit level of granularity when reading binary data. Furthermore, it is rare for a file type to contain single boolean values in its specification; boolean flags are typically grouped together and expressed on the byte level even if the extra bits are unused. Therefore, it is usually best to express these flags using bitfields in the file description language.

After the value has been read, it is unpacked using the **struct** module. The **struct** module is a part of the Python standard library and is used to unpack bytes read from a file according to a specified endianness and interprets them as a specified Python type. The **struct** module is used by the parser to unpack the values to either Python integers or strings. The value read in is printed to the console before the function returns.

The `parseCharfield` function performs a series of read operations on the binary file. It performs either *cfNode.size* reads if the charfield was specified with a length, or it continues to read and interpret the data as characters until it reads in the terminating character defined in the Charfield declaration. The following excerpt demonstrates how the string is read, unpacked, and decoded.

```
def parseCharfield(self, cfNode):
        string = ""
        if cfNode.size >= 0:
            for i in range(cfNode.size):
         if cfNode.type.type == "char1":
             if cfNode.encoding != "":
                 string += value.decode(cfNode.encoding)
             else:
                 string += value.decode('utf-8')
         else:
             if self.BigEndian:
                 value = struct.unpack('>H', value)[0]
             else:
                 value = struct.unpack('<H', value)[0]
                   string += "\\u" + str(hex(value))[2:]
```

The final parsing method is the **parseBitfield** function. This function is passed a BitfieldNode as a parameter and performs a read operation on the binary file. The function reads in the size of the bitfield in bytes and prints the value of the bits in hexadecimal format.

```
def parseBitfield(self, bfNode):
        value = self.binaryFile.read(int(bfNode.size / 8))
        print( str(binascii.hexlify(value)))
```

5 Results

The previous chapter presented the design and implementation of the BAT's lexical analyzer, parser, and binary parser components. This chapter presents two of the test cases created to test and verify accuracy of the parsing tool.

5.1 Test cases

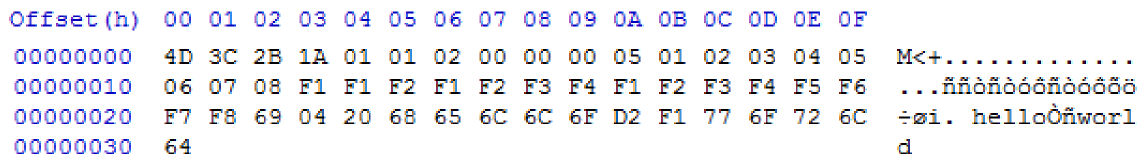
During development, several files were created for testing and debugging purposes. These test files were designed to test each feature of the language supported by the BAT. Although meaningless in the data it contains, the structure of the file and the associated description provided a broad test of the supported features of the description language.

The full description of the new file type can be found in Figure 35.

```
LITTLE_ENDIAN;
main struct file:= {
   magic_number := {
      uint4;
      description := "must start every file.";
      permissible := 0x1a2b3c4d;
   }

   _header := HEADER;
   _body   := BODY;
}

struct HEADER := {
   v1 := uint1;
   v2 := uint2;
   v3 := uint4;
   v4 := uint8;

   v5 := sint1;
   v6 := sint2;
   v7 := sint4;
   v8 := sint8;

   v9 := char1;
   v10 := char2[2];
}

struct BODY := {
   cf:= char1[_header.v3];
   _ext := case(_header.v1) {
      0: uint8;
      1: EXT;
      else: char1[10];
   };
}

struct EXT := {
   bfield1 := bitfield[0x8];
   bfield2 := bitfield(8, bit0, "a bit",
                          bit1, "another bit",
                          bit2, "a third bit",
                          bit3, "the last bit",
                          bit4, "padding",
                          bit5, "padding",
                          bit6, "padding",
                          bit7, "padding",);
   cfield := charfield(char1, 5, "ascii");
}
```

Figure 35 Test file description

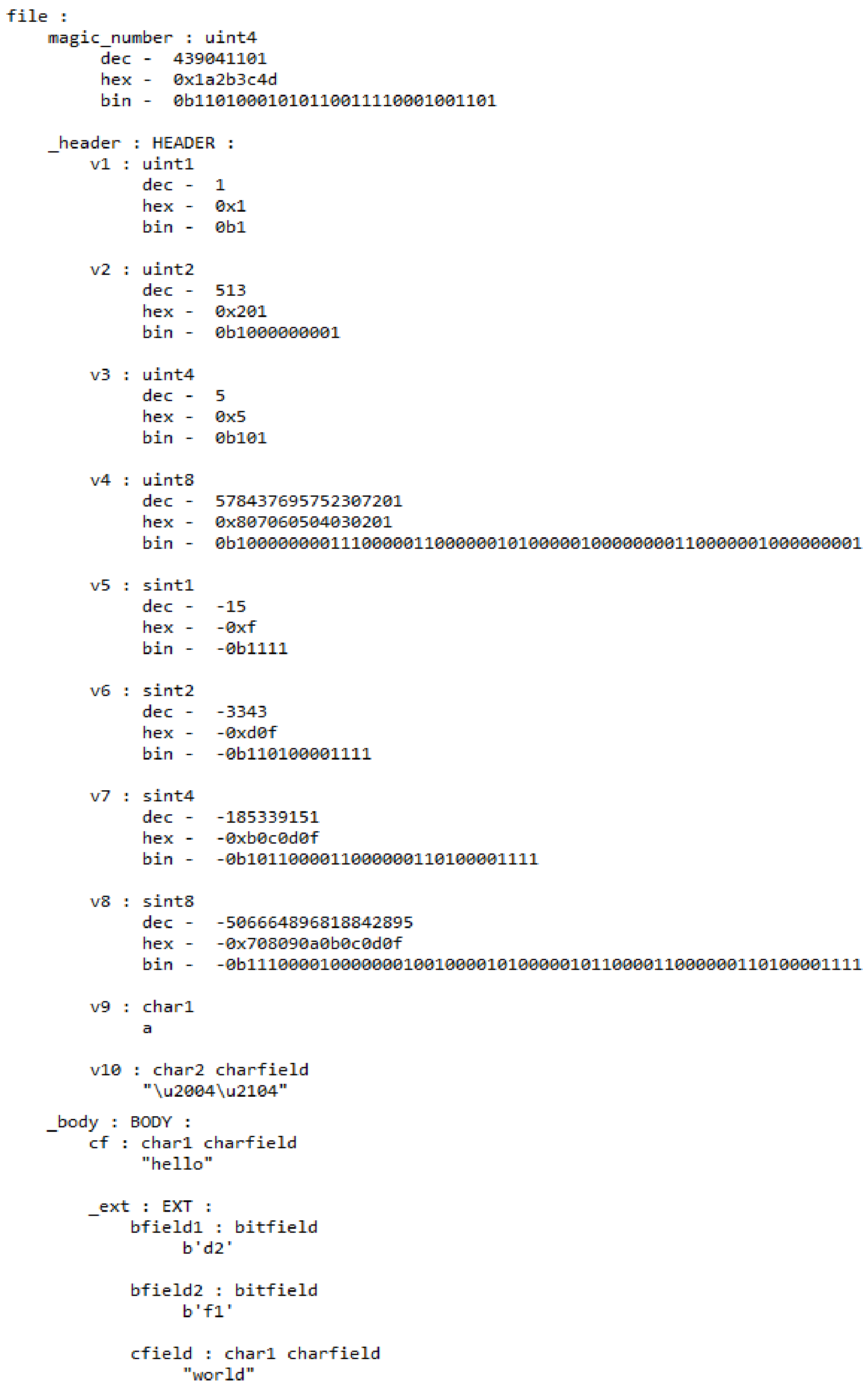
Test files that fit the above description were created to test the BAT. One of the test files is shown in Figure 36.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  4D 3C 2B 1A 01 01 02 00 00 00 05 01 02 03 04 05  M<+.............
00000010  06 07 08 F1 F1 F2 F1 F2 F3 F4 F1 F2 F3 F4 F5 F6  ...ññòñòóôñòóôõö
00000020  F7 F8 69 04 20 68 65 6C 6C 6F D2 F1 77 6F 72 6C  ÷øi. helloÒñworl
00000030  64                                               d
```

Figure 36 binary test file

Figure 37 shows the output from the BAT tool using the test file description and binary file. As the figure shows, the BAT accurately parses out each field and presents them to the user. The BAT's output displays character and charfields as the literal character or string read from the file; integer type fields are displayed with their decimal, hexadecimal, and binary representations. Bitfields are represented in hexadecimal format.



```
file :
    magic_number : uint4
         dec -  439041101
         hex -  0x1a2b3c4d
         bin -  0b11010001010110011110001001101

    _header : HEADER :
        v1 : uint1
             dec -  1
             hex -  0x1
             bin -  0b1

        v2 : uint2
             dec -  513
             hex -  0x201
             bin -  0b1000000001

        v3 : uint4
             dec -  5
             hex -  0x5
             bin -  0b101

        v4 : uint8
             dec -  578437695752307201
             hex -  0x807060504030201
             bin -  0b100000000111000001100000010100000100000000110000001000000001

        v5 : sint1
             dec -  -15
             hex -  -0xf
             bin -  -0b1111

        v6 : sint2
             dec -  -3343
             hex -  -0xd0f
             bin -  -0b110100001111

        v7 : sint4
             dec -  -185339151
             hex -  -0xb0c0d0f
             bin -  -0b1011000011000000110100001111

        v8 : sint8
             dec -  -506664896818842895
             hex -  -0x708090a0b0c0d0f
             bin -  -0b111000010000000100100001010000010110000110000000110100001111

        v9 : char1
             a

        v10 : char2 charfield
             "\u2004\u2104"
    _body : BODY :
        cf : char1 charfield
             "hello"

        _ext : EXT :
            bfield1 : bitfield
                 b'd2'

            bfield2 : bitfield
                 b'f1'

            cfield : char1 charfield
                 "world"
```

Figure 37 BAT output from test file description.

The custom file type discussed above tested a wide array of the features supported by the parser, but ultimately it is a meaningless data structure. Further tests were conducted on real world data structures. The following example using FAT32 directory entries provides a more realistic illustration of the capabilities of the BAT.

FAT32 directory entries are 32 byte data structures used to store the short name, timestamps, starting cluster number, and other meta-data stored by the file system for a file. Table 2 shows the official specification of a FAT32 directory entry provided by Microsoft [28].

Name	**Offset(byte)**	**Size(bytes)**	**Description**
DIR_Name	0	11	Short name.
DIR_Attr	11	1	File attributes: ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_LONG_NAME ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID The upper two bits of the attribute byte are reserved and should always be set to 0 when a file is created and never modified or looked at after that.
DIR_NTRes	12	1	Reserved for use by Windows NT. Set value to 0 when a file is created and never modify or look at it after that.
DIR_CrtTimeTenth	13	1	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second. The granularity of the seconds part of DIR_CrtTime is 2 seconds so this field is a count of tenths of a second and its valid value range is

			0-199 inclusive.
DIR_CrtTime	14	2	Time file was created.
DIR_CrtDate	16	2	Date file was created.
DIR_LstAccDate	18	2	Last access date. Note that there is no last access time, only a date. This is the date of last read or write. In the case of a write, this should be set to the same date as DIR_WrtDate.
DIR_FstClusHI	20	2	High word of this entry's first cluster number (always 0 for a FAT12 or FAT16 volume).
DIR_WrtTime	22	2	Time of last write. Note that file creation is considered a write.
DIR_WrtDate	24	2	Date of last write. Note that file creation is considered a write.
DIR_FstClusLO	26	2	Low word of this entry's first cluster number.
DIR_FileSize	28	4	32-bit DWORD holding this file's size in bytes.

Table 2 FAT32 directory entry specification [28].

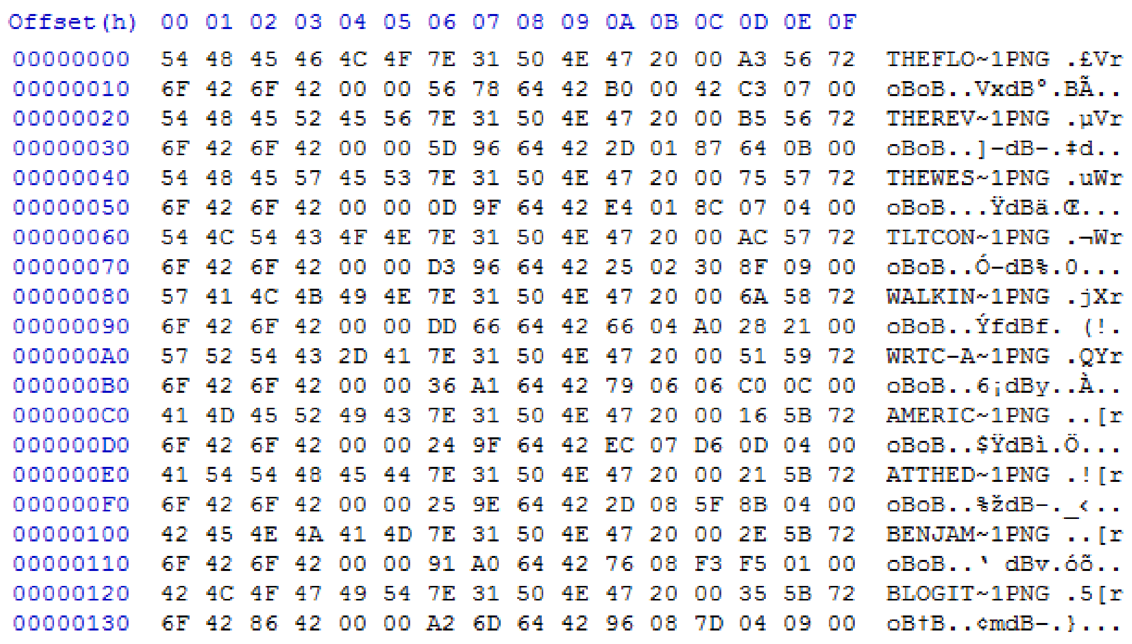
Using the specification provided by Microsoft, a description of a directory entry can be written using the file description language. Figure 38 shows a description of a file containing a list of ten FAT32 directory entries.

```
Main struct dirList := {
        directories := directoryEntry[10];
}

struct directoryEntry := {
        filename        := char1[8];
        extension       := char1[3];

        attributes      := bitfield(8,
                                    READ_ONLY, "File is read only.",
                                    HIDDEN, "File is hidden.",
                                    SYSTEM, "",
                                    VOLUME_ID, ""
                                    DIRECTORY, "Entry is a directory.",
                                    ARCHIVE, "Entry is an archive.",
                                    RES1, "Reserved, should be 0.",
                                    RES2, "reserved, should be 0.");
        reserved        := uint1;
        creationTimeMs := uint1;
        creationTime   := uint2;
        creationDate   := uint2;
        accesssDate    := uint2;

        StartingClusterHighWord := {
                uint2;
                name := "High word of the entry's starting cluster";
        }

        writeTime      := uint2;
        writeDate      := uint2;

        StartingClusterLowWord := {
                uint2;
                name := "Low word of the entry's starting cluster";

        filesize       := uint4;
}
```

Figure 38 Description of a list of FAT32 directory entries

The FAT specification indicates that the date and timestamps are all stored in 16 bit words using the MS-DOS format. Currently, neither the file description language nor the BAT have a method of specifying the format timestamps are stored. A uint2 is used to specify the time and date stamps. Until the interpretation of timestamps is supported by future versions of the parser, it will be the duty of the examiner to interpret the values stored in these fields.

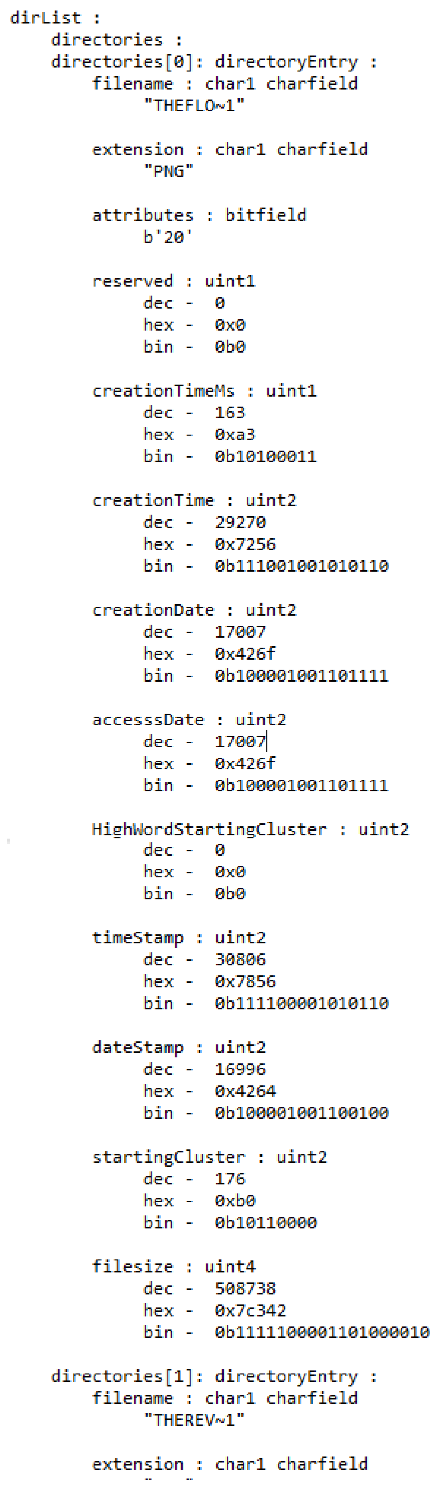
A binary file matching the description shown in Figure 38 was created. Figure 39 shows the contents of the binary file.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  54 48 45 46 4C 4F 7E 31 50 4E 47 20 00 A3 56 72  THEFLO~1PNG .£Vr
00000010  6F 42 6F 42 00 00 56 78 64 42 B0 00 42 C3 07 00  oBoB..VxdB°.BÃ..
00000020  54 48 45 52 45 56 7E 31 50 4E 47 20 00 B5 56 72  THEREV~1PNG .µVr
00000030  6F 42 6F 42 00 00 5D 96 64 42 2D 01 87 64 0B 00  oBoB..]–dB-.‡d..
00000040  54 48 45 57 45 53 7E 31 50 4E 47 20 00 75 57 72  THEWES~1PNG .uWr
00000050  6F 42 6F 42 00 00 0D 9F 64 42 E4 01 8C 07 04 00  oBoB...ŸdBä.Œ...
00000060  54 4C 54 43 4F 4E 7E 31 50 4E 47 20 00 AC 57 72  TLTCON~1PNG .¬Wr
00000070  6F 42 6F 42 00 00 D3 96 64 42 25 02 30 8F 09 00  oBoB..Ó–dB%.0...
00000080  57 41 4C 4B 49 4E 7E 31 50 4E 47 20 00 6A 58 72  WALKIN~1PNG .jXr
00000090  6F 42 6F 42 00 00 DD 66 64 42 66 04 A0 28 21 00  oBoB..ÝfdBf. (!.
000000A0  57 52 54 43 2D 41 7E 31 50 4E 47 20 00 51 59 72  WRTC-A~1PNG .QYr
000000B0  6F 42 6F 42 00 00 36 A1 64 42 79 06 06 C0 0C 00  oBoB..6¡dBy..À..
000000C0  41 4D 45 52 49 43 7E 31 50 4E 47 20 00 16 5B 72  AMERIC~1PNG ..[r
000000D0  6F 42 6F 42 00 00 24 9F 64 42 EC 07 D6 0D 04 00  oBoB..$ŸdBì.Ö...
000000E0  41 54 54 48 45 44 7E 31 50 4E 47 20 00 21 5B 72  ATTHED~1PNG .![r
000000F0  6F 42 6F 42 00 00 25 9E 64 42 2D 08 5F 8B 04 00  oBoB..%ždB-._‹..
00000100  42 45 4E 4A 41 4D 7E 31 50 4E 47 20 00 2E 5B 72  BENJAM~1PNG ..[r
00000110  6F 42 6F 42 00 00 91 A0 64 42 76 08 F3 F5 01 00  oBoB..‘ dBv.óõ..
00000120  42 4C 4F 47 49 54 7E 31 50 4E 47 20 00 35 5B 72  BLOGIT~1PNG .5[r
00000130  6F 42 86 42 00 00 A2 6D 64 42 96 08 7D 04 09 00  oB†B..¢mdB–.}...
```

Figure 39 Binary content of FAT32 directory entry list file.

The BAT was executed using the above description and binary file. Figure 40 shows an excerpt of the BAT's output.



```
dirList :
    directories :
    directories[0]: directoryEntry :
        filename : char1 charfield
             "THEFLO~1"

        extension : char1 charfield
             "PNG"

        attributes : bitfield
             b'20'

        reserved : uint1
             dec -  0
             hex -  0x0
             bin -  0b0

        creationTimeMs : uint1
             dec -  163
             hex -  0xa3
             bin -  0b10100011

        creationTime : uint2
             dec -  29270
             hex -  0x7256
             bin -  0b111001001010110

        creationDate : uint2
             dec -  17007
             hex -  0x426f
             bin -  0b100001001101111

        accesssDate : uint2
             dec -  17007
             hex -  0x426f
             bin -  0b100001001101111

        HighWordStartingCluster : uint2
             dec -  0
             hex -  0x0
             bin -  0b0

        timeStamp : uint2
             dec -  30806
             hex -  0x7856
             bin -  0b111100001010110

        dateStamp : uint2
             dec -  16996
             hex -  0x4264
             bin -  0b100001001100100

        startingCluster : uint2
             dec -  176
             hex -  0xb0
             bin -  0b10110000

        filesize : uint4
             dec -  508738
             hex -  0x7c342
             bin -  0b1111100001101000010

    directories[1]: directoryEntry :
        filename : char1 charfield
             "THEREV~1"

        extension : char1 charfield
```

Figure 40 BAT output from parsing FAT32 directory entries

6 Conclusion

The current state of file carving and data extraction research is largely focused on the recovery of files from hard drive images, memory dumps, or other sources of data. Research on data extraction from files to assist in the analysis of recovered files has been limited. The small number of currently available tools that can be used display binary file contents in a readable form, all have their limitations and drawbacks. The tools can be overly complex and require a substantial learning curve. They can be too simplistic and lack the ability to support complex file structures. This thesis identified a gap in the research and attempted to offer a tool to fill this gap.

6.1 Accomplishments

The BAT developed through this thesis offers a unique approach to data extraction by parsing and interpreting a binary file based on a textual description of the file type. Both time and effort are wasted when an examiner is required to develop custom programs to extract data from files. Not only is the examiner tasked with first gaining a deep understanding of the file type to be parsed, but then also with writing, debugging, and testing custom code to perform the data extraction. The BAT developed through this thesis removes this work load from the examiner.

The results from the initial version of the BAT demonstrate its effectiveness. The tool is easy to use and accurate at interpreting binary data. It only requires the user to provide a valid description of the binary to be parsed. The file description language offers an effective means of describing a binary format; it is easy to read, write, and understand due to the relatively small learning curve. The ease of use and effectiveness of this method of data extraction make this a worthwhile contribution to field of digital forensics. The BAT can be used to display a binary file effectively and accurately and assist the examiner in the analysis of the data contained within.

6.2 Limitations of This Work

The BAT developed during this thesis does have limitations. The parser does not support every feature of the file description language. By not supporting global and local pointers, lists of unknown size, and virtual fields, the parser is unable to parse and interpret more complex binary file types.

The current version of the parser also does not support the parsing of single boolean values. The occurrence of single boolean values within a binary file is low; boolean flags are usually found in groups and padded to the nearest byte. However, there is the chance that single boolean values could occur and the current parser will not be able to parse and interpret these binary files. It is the hope that future versions of the parser will support this feature.

Another limitation of the tool is the lack of code generation. The current version of the parser must read the file description and rebuild the syntax tree during every execution to parse a binary file.

6.3 Future Work

The results observed during this thesis are promising and the BAT has many possibilities for future work and extensibility. The most obvious choice for future work on the parser includes implementing the language features left unsupported by this initial research. Adding support for these extra language features would allow the parser to support the parsing of a larger variety of file types.

Another obvious topic for future work would be the creation of a file description repository. Hosting a repository of descriptions for common file type and data structures would provide a useful resource for examiners. This repository should also allow for user submissions, to provide an easy means of sharing descriptions among the digital forensics community.

Offering code generation instead of parsing the binary by traversing the syntax tree is also a topic for future work. Adding this feature would improve runtime speed by providing a separate program to perform the parsing of the binary file. The file description would only need to be read and parsed once instead of at the beginning of every execution and code would be generated from the resulting tree.

Improvements to the BAT's output and reporting functionality could also be made. The current version of the parser reports the values interpreted from the binary file by printing to standard out. For large files, this method can become quickly overwhelming for the user. Reports could be improved upon by generating an xml or database like structure which can allow for easier querying of individual fields or groups of fields that are of interests.

As discussed in the results from FAT32 directory entry test case, the file description language and the BAT have no means of interpreting timestamps. Timestamps can be common attributes, especially if parsing file system data structures. Future advancements could include adding features to the description language to denote the style and format of timestamps and implementing features in the BAT to handle timestamps and present them in a human readable format for easier analysis and interpretation.

Appendix A – Lexical Analyzer token list

TOKEN	**Symbol or Regular Expression**
t_PLUS	+
t_MINUS	-
t_ASTERISK	*
t_FWDSLASH	/
t_PERCENT	%
t_PIPE	|
t_ASSIGN	:=
t_COLON	:
t_SEMI	;
t_COMMA	,
t_DOT	.
t_LPARAN	(
t_RPARAN	)
t_LBRACKET	[
t_RBRACKET	]
t_LBRACE	{
t_RBRACE	}
t_BIN_NUMBER	b[0-1]+
t_OCT_NUMBER	[0][0-7]+
t_DEC_NUMBER	[1-9][0-9]*
t_HEX_NUMBER	0[xX][0-9a-fA-F]+
t_STRING	\“[^\”]+\”
t_COMMENT	//.*
t_ID	[a-zA-Z_][a-zA-Z_0-9]
t_MAIN	Main
t_BITFIELD	Bitfield
t_CHARFIELD	Charfield
t_GLOBALP	Globalp
t_LOCALP	Localp
t_CASE	Case
t_ELSE	Else
t_DESCRIPTION	description
t_NAME	name
t_PERMISSIBLE	permissible

t_COMMON	common
t_UINT1	unit1
t_UINT2	unit2
t_UINT4	unit4
t_UINT8	unit8
t_SINT1	sint1
t_SINT2	sint2
t_SINT4	sint4
t_SINT8	sint8
t_CHAR1	char2
t_CHAR2	char2
t_BOOLEAN	boolean

Table 3 File description language token list with associated matching patterns

Appendix B – Lexical Analyzer Implementation

```
import ply.lex as lex


reserved = (
       'MAIN', 'BITFIELD', 'CHARFIELD', 'GLOBALP', 'LOCALP', 'CASE', 'ELSE',
'DESCRIPTION',
       'NAME', 'PERMISSIBLE', 'COMMON', 'UINT1', 'UINT2', 'UINT4', 'UINT8',
'SINT1', 'SINT2',
       'SINT4', 'SINT8', 'CHAR1', 'CHAR2', 'BOOLEAN', 'STRUCT'
       )
tokens = reserved + (
       # Literals
       'BIN_NUMBER', 'OCT_NUMBER', 'DEC_NUMBER', 'HEX_NUMBER', 'STRING',
       # Operators
       'PLUS', 'MINUS', 'ASTERISK', 'FWDSLASH', 'PERCENT',
       'PIPE', 'ASSIGN', 'COLON', 'SEMI', 'COMMA', 'DOT',

       # Delimeters ( ) [ ] { }
       'LPARAN', 'RPARAN',
       'LBRACKET', 'RBRACKET',
       'LBRACE', 'RBRACE',

       'ID', 'NEWLINE',
       'COMMENT'
       )

# Operators
t_PLUS        = r'\+'
t_MINUS       = r'-'
t_ASTERISK    = r'\*'
t_FWDSLASH    = r'/'
t_PERCENT     = r'%'
t_PIPE        = r'\|'

t_ASSIGN      = r':='
t_COLON       = r':'
t_SEMI        = r';'
t_COMMA       = r','
t_DOT         = r'\.'

t_LPARAN      = r'\('
t_RPARAN      = r'\)'
t_LBRACKET    = r'\['
t_RBRACKET    = r']'
t_LBRACE      = r'{'
t_RBRACE      = r'}'

t_BIN_NUMBER = r'b[0-1]+'
t_OCT_NUMBER = r'[0][0-7]+'
t_HEX_NUMBER = r'0[xX][0-9a-fA-F]+'

def t_DEC_NUMBER(t):
       r'[1-9][0-9]*'
       t.value = int(t.value)
       return t

def t_STRING(t):
    r'\"[^\"]+\"'
    t.lexer.lineno += t.value.count("\n")
    return t
t_COMMENT    = r'//.*'

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    if t.value.upper() in reserved:
        t.type = t.value.upper()
    return t

t_ignore = ' \t'

def t_NEWLINE(t):
    r'\n'
    t.lexer.lineno += 1
    #return t

def t_error(t):
    print("Illegal character %s" % repr(t.value[0]))
    t.lexer.skip(1)

lexer = lex.lex()
```

Appendix C - Parser Implementation

```
import tokenizer
from tokenizer import tokens
import ply.yacc as yacc

from Nodes.StructNode import *
from Nodes.DeclareNode import *
from Nodes.ListNode import *
from Nodes.TypeNode import *
from Nodes.CharfieldNode import *
from Nodes.BitfieldNode import *
from Nodes.ConditionalNode import *

from SymbolTables.SymbolTables import *

currentDeclareList = []
currentID = ""

currentConditionals = {}

def p_description(p):
    'description : main_struct struct_list'
    p[0] = p[1]
    treePatch()

def p_main_struct(p):
    'main_struct : MAIN struct'
    p[0] = p[2]

def p_struct_list(p):
    '''struct_list : struct_list struct
                   | struct'''


def p_struct(p):
    'struct : STRUCT ID ASSIGN LBRACE decl_list RBRACE'
    global currentDeclareList
    p[0] = StructNode(p[2])
    p[0].declarations = currentDeclareList
    currentDeclareList = []

def p_decl_list(p):
    '''decl_list : decl_list decl
                 |  decl  '''
    global currentDeclareList
    if len(p) == 3:
        currentDeclareList.append(p[2])
    else:
        currentDeclareList.append(p[1])

def p_decl(p):
    '''decl : ID ASSIGN type SEMI
            | ID ASSIGN LBRACE extDecl RBRACE'''
    if len(p) == 5:
        p[0] = DeclareNode(p[1], p[3])
    else:
        p[0] = DeclareNode(p[1], p[4])

def p_extDecl(p):
    '''extDecl : type SEMI extOp_list
    '''
    p[0] = p[1]

def p_extOp_list(p):
    '''extOp_list : extOp_list extOp SEMI
                  | extOp SEMI'''

def p_extOp(p):
    '''extOp : DESCRIPTION ASSIGN STRING
             |  NAME ASSIGN STRING
             |  PERMISSIBLE ASSIGN STRING
             |  COMMON ASSIGN STRING'''

def p_type(p):
    '''type :  UINT1
              | UINT2
              | UINT4
              | UINT8
              | SINT1
              | SINT2
              | SINT4
              | SINT8
              | CHAR1
              | CHAR2
              | BOOLEAN
              | ID
              | list
              | charfield
              | bitfield
              | conditional'''
    p[0] = p[1]

def p_list(p):
    '''list : type LBRACKET number RBRACKET
            | type LBRACKET reference RBRACKET'''
    if isinstance(p[3], int):
        p[0] = ListNode(p[1], p[3])
    else:
        p[0] = ListNode(p[1], p[3])

def p_charfield(p):
    '''charfield : CHARFIELD LPARAN CHAR1 COMMA number COMMA STRING RPARAN
                 | CHARFIELD LPARAN CHAR2 COMMA number COMMA STRING RPARAN

                 | CHARFIELD LPARAN CHAR1 COMMA number RPARAN
                 | CHARFIELD LPARAN CHAR2 COMMA number RPARAN

                 | CHARFIELD LPARAN CHAR1 COMMA ID COMMA STRING RPARAN
```

```
                  | CHARFIELD LPARAN CHAR2 COMMA ID COMMA STRING RPARAN

                  | CHARFIELD LPARAN CHAR1 COMMA ID RPARAN
                  | CHARFIELD LPARAN CHAR2 COMMA ID RPARAN

                  | CHAR1 LBRACKET number RBRACKET
                  | CHAR1 LBRACKET reference RBRACKET
                  | CHAR2 LBRACKET number RBRACKET
                  | CHAR2 LBRACKET reference RBRACKET'''

    if p[1].upper() == "CHARFIELD":
        p[0] = CharfieldNode(p[3])
        if isinstance(p[5], int):
            p[0].size = p[5]
        else:
            p[0].termChar = p[5]
            p[0].size = -1
    else:
        p[0] = CharfieldNode(p[1])
        p[0].size = p[3]

def p_bitfield(p):
    ''' bitfield : BITFIELD LBRACKET number RBRACKET
                 | BITFIELD LPARAN number COMMA bit_list RPARAN
    '''
    p[0] = BitfieldNode(p[3])

def p_bit_list(p):
    ''' bit_list : bit_list bit_def COMMA
                 | bit_def COMMA
                 | bit_def
    '''

def p_bit_def(p):
    ''' bit_def : ID COMMA STRING
    '''

def p_conditional(p):
    '''conditional : CASE LPARAN reference RPARAN LBRACE case_list RBRACE
    '''
    global currentConditionals
    p[0] = ConditionalNode()
    p[0].reference = p[3]
    p[0].conditions = currentConditionals
    currentConditionals = {}

def p_case_list(p):
    '''case_list : case_list case
                 | case
                 | empty
    '''

def p_case(p):
    '''case : case_value COLON type SEMI
            | ELSE COLON type SEMI
    '''
    global currentConditionals
    currentConditionals[p[1]] = TypeNode(p[3])

def p_case_value(p):
    '''case_value : number
    '''
    p[0] = p[1]


def p_number(p):
    '''number : DEC_NUMBER
              | BIN_NUMBER
              | HEX_NUMBER
              | OCT_NUMBER
    '''
    p[0] = p[1]

def p_reference(p):
    '''reference : ID DOT reference
                 | ID
    '''
    if len(p) == 4:
        p[0] = []
        p[0].append(p[1])
        p[0].extend(p[3])
    else:
        p[0] = [p[1]]

def p_error(p):
    print("unexpected token: ", p)
    exit()

def p_empty(p):
    'empty :'
    pass

def treePatch():
    for node in patchTable.keys():
        node.type = SymbolTable[patchTable[node]]


fdp = yacc.yacc()
```

Bibliography

[1] Internet Crime Complaint Center, "2011 Internet Crime Report," 2011.

[2] E. Locard, Traite de criminalistique, Vol I., Lyon: Desvigne, 1931.

[3] E. Casey, "Digital Evidence and Computer Crime: Forensic Science, Computers and The Internet", San Diego: Academic Press, 2011.

[4] "Strengthening Forensic Science in the United States: A Path Forward," Committee on Identifying the Needs of the Forensic Sciences Community, National Research Council, February 2009. [Online]. Available: http://www.nap.edu/openbook.php?record_id=12589. [Accessed 2 February 2013].

[5] A. Pal and N. Memon, "The Evolution of File Carving," *IEEE SIGNAL PROCESSING MAGAZINE,* pp. 59-71, March 2009.

[6] Guidance Software, "EnCase Forensic v7," Guidance Software, [Online]. Available: http://www.guidancesoftware.com/encase-forensic.htm#tab=1. [Accessed 14 February 2013].

[7] X-Ways AG, "X-Ways Software Technology AG," [Online]. Available: http://www.x-ways.net/. [Accessed 8 Febraury 2013].

[8] S. Garfinkel, "File Carving," 25 September 2012. [Online]. Available: http://www.forensicswiki.org/wiki/File_Carving. [Accessed 2012 5 October].

[9] M. Cohen, "Advance Carving Techniques," *Digital Investigation,* vol. 4, pp. 119-128, 2007.

[10] S. Garfinkel, "Carving contiguous and fragmented files with fast object validation," *Digital Investigation,* pp. 2-12, 2007.

[11] J. Metz and R.-J. Mora, "Analysis of 2006 DFRWS Forensic Carving Challenge," 2006.

[12] C. J. Veenman, "Statistical Disk Cluster Classification for File Carving," in *Third International Symposium on Information Assurance and Security, 2007. IAS 2007.* , Manchester, 2007.

[13] The Volatility Project, "The Volatility Framework," [Online]. Available: http://code.google.com/p/volatility/. [Accessed 21 2012 November].

[14] Acquire Media, "Guidance Software Named a Leader in IDC MarketScape on Early Case Assessment," 2011. [Online]. Available: http://investors.guidancesoftware.com/releasedetail.cfm?releaseid=611297. [Accessed 4 April 2013].

[15] L. Muller, "EnScript Tutorial," [Online]. Available: http://www.lancemueller.com/blog/EnScript%20All%20tutorial%20Document.pdf. [Accessed 12 February 2013].

[16] Y. Khatri, "Enscript Tutorial 1 - Parse XP System Restore Logs," 2 MARCH 2012. [Online]. Available: http://www.swiftforensics.com/2012/03/enscript-tutorial-1-

parse-xp-system.html. [Accessed 22 February 2013].

[17] L. Muller, "Office Metadata EnScript," 10 July 2007. [Online]. Available: http://www.forensickb.com/2007/07/office-metadata.html. [Accessed 11 Feburary 2013].

[18] X-Ways AG, "Template Editing," [Online]. Available: http://www.x-ways.net/winhex/templates.html#User_Templates. [Accessed 11 February 2013].

[19] K. Volodymyr, "BMP Template," [Online]. Available: http://www.x-ways.net/winhex/templates/BMP.tpl. [Accessed 11 February 2013].

[20] Microsoft, "Bitmap Structures," [Online]. Available: http://msdn.microsoft.com/en-us/library/dd183392(v=vs.85).aspx. [Accessed 22 February 2013].

[21] X-Ways AG, "ZIP Template," [Online]. Available: http://www.x-ways.net/winhex/templates/ZIP.tpl. [Accessed 11 February 2012].

[22] PKWARE, Inc., "APPNOTE.TXT - .ZIP File Format Specification," 1 September 2012. [Online]. Available: http://www.pkware.com/documents/casestudies/APPNOTE.TXT. [Accessed 18 February 2013].

[23] J. Levine, "Flex and Bison", Sebastopol: O'Reilly, 2009.

[24] M. E. Lesk and E. Schmidt, "Lex − A Lexical Analyzer Generator," 21 July 1975. [Online]. Available: http://ken-cc.googlecode.com/svn/trunk/doc/lex.pdf. [Accessed

11 Feburary 2013].

[25] "PLY (Python Lex-Yacc)," [Online]. Available: http://www.dabeaz.com/ply/. [Accessed 20 August 2012].

[26] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques, & Tools", Boston: Pearson and Addison-Wesley, 2007.

[27] D. M. Beazley, "PLY (Python Lex-Yacc)," [Online]. Available: http://www.dabeaz.com/ply/ply.html. [Accessed 22 August 2012].

[28] Microsoft, "FAT32 File System Specification," 6 December 2000. [Online]. Available: http://msdn.microsoft.com/en-US/windows/hardware/gg463084. [Accessed 4 April 2013].

[29] P. Bobby, "Secure Artisan: My Road to Digital Forensics Execellence," [Online]. Available: http://secureartisan.wordpress.com/my-files/. [Accessed 15 February 2013].

[30] F. Crispino and M. M. Houck, "Encyclopedia of Forensic Sciences", Elsevier, 2013, pp. 278-281.