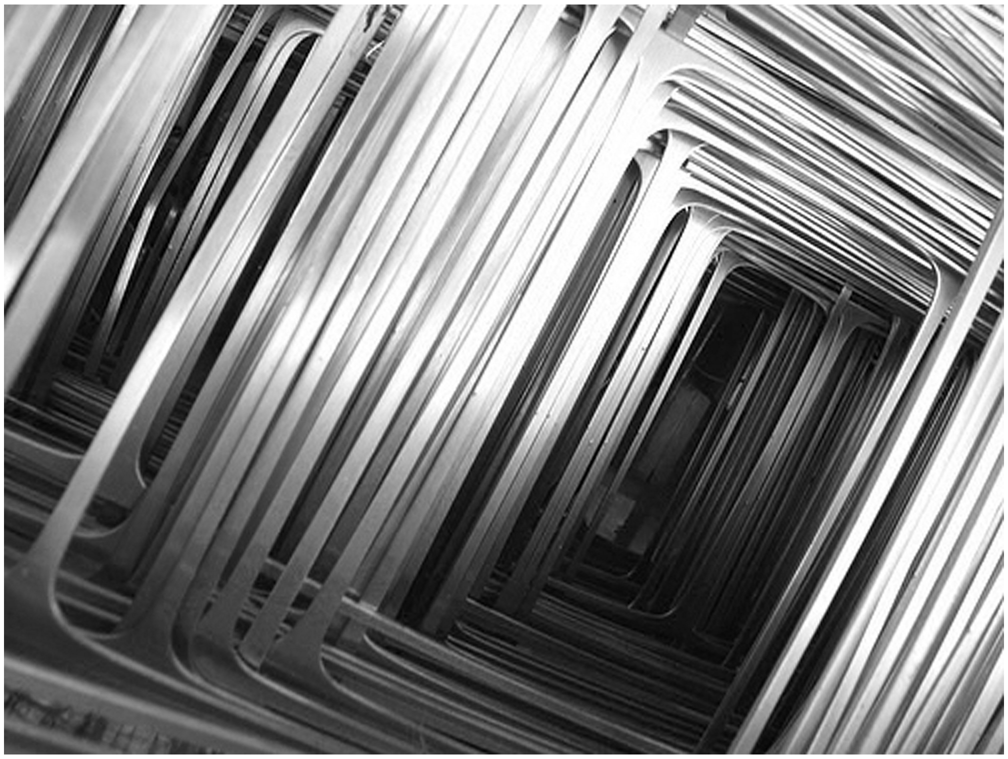
UNIVERSITY OF GOTHENBURG



Creating an interpreted dissector for Wireshark

Master of Science Thesis in Computer Science

Tobias Wärre

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Creating and interpreted dissector for Wireshark

Tobias Wärre

© Tobias Wärre, February 2010.

Examiner: Aarne Ranta

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Junk yard sexy (metal abyss) by Tanakawho on Flickr.com, available under an attribution and non commercial Creative Commons license, grey scale conversion by the author.

Department of Computer Science and Engineering
Göteborg, Sweden February 2010

Abstract

Gateway GPRS Support Node (GGSN) is a telecommunication router, which is used for mobile data communication. Ericsson's GGSN is a very protocol intensive GGSN, as it is able to communicate with many other telecommunication network elements in a wide range of protocols. At a glance, Ericsson's GGSN is a rack, which consists of a number of Flexible PIC Concentrators (FPCs), which in turn hosts a number of Physical Interface Cards (PICs). Different PICs has different roles in the Ericsson GGSN architecture, and as such they communicate with each other in a number of different proprietary protocols; protocols that are undergoing constant change.

Wireshark is an open source software protocol analyser application, which is used extensively for network troubleshooting, analysis and protocol development. Wireshark supports a variety of standard protocols, but does not (per definition) support proprietary protocols (a variety of standard protocols, but by no means all).

Hence, for a designer/programmer of Ericsson GGSN, it would be very helpful to use Wireshark for troubleshooting, analysis and development also of the none-Wireshark-supported protocols. The usual procedure to add proprietary protocols is to patch Wireshark with the new code. That is a time consuming activity both for setting up the environment and coding the protocol. The idea is to provide a protocol dissector that will take a text file as input and dissect protocols from that.

Sammanfattning

En Gateway GPRS Support Node (GGSN) är en telekommunikationsväxel för mobil dataöverföring. Den är väldigt protokollintensiv då den kan kommunicera med en mängd andra kommunikationsapparater med hjälp av ett stort antal protokoll. En GGSN innehåller flera specialiserade kort med olika syften och uppgifter som kommunicerar med olika proprietära protokoll. Dessa protokoll ändras ofta under utvecklingen.
Wireshark är ett program som lyssnar på och analyserar nätverkstrafik. Programmet har stöd för ett flertal öppna protokoll men har, delvis på grund av dess licens (GPL) dåligt stöd för proprietära protokoll. Tidigare har man på Ericsson gjort en så kallad dissekerare som man har kompilerat till en modul, men allt eftersom protokollen de använder ändras måste även modulen ändras. Ett stort problem är att utvecklingsmiljön för Wireshark är komplicerad och tar lång tid att göra i ordning och detsamma gäller för underhållet av modulerna. Idén med detta arbete är att skapa en modul som läser in en textfil som är en definition av protokollet som skall avkodas i nätverkspaket och därmed slippa utveckla modulerna och problem med licenser.

Preface

I would like to thank my wife Annika, who have given me support and pushed me on when I wrote this report. I would like to thank my examiner Aarne Ranta for all the support during the time of the thesis. I would also like to thank Anna Sandström for connecting me with the people who came up with the idea for this thesis, Henric Bergenwall and Kim Martinsen. Henric was also my tutor at Ericsson. And last I would like to thank all the men and women on the Wireshark development mailing list, while not directly contributing; they definitely helped keeping my hope up of finishing this thesis.

Index

1 Introduction

At Ericsson's department for GGSN, there is a need for a network analyser program, and they fill that need with Wireshark[1]. The GGSN uses a lot of proprietary protocols that changes regularly. Wireshark is chosen since it is and open source program, since the source can be modified. So far, they have patched Wireshark with the proprietary protocol dissectors that they have written themselves.

The problem with these protocols are that they are proprietary and as such should not be given out to the public or be included in Wireshark because of its license. The protocols also changes fairly often requiring changes in the protocol dissector. These changes are time consuming to implement and also have special requirements on the development environment where it is being implemented. These restrictions also apply if the protocol dissectors are compiled into a plug-in since they have the same dependencies and it will also require the same compiler version for the plug-in as was used for Wireshark.

To solve this problem a special dissector has been developed, which reads a text file and use the outline in that text file to present the information from the captured network traffic. This will have the effect that it is much faster to change the dissector when a protocol changes, since there is only need to change the contents of the text file and it has no extra dependencies that has to be installed or checked for correct version.
This dissector have been created using a tool called BNFC[2], which reads a file where you can specify keywords, symbols, literals and syntactic structure, i.e. a language, and it will generate lexer, parser and skeleton code for further expansion.

1.1 Purpose

The purpose of the thesis work is to create an interpreted dissector that uses a protocol definition written in a text file. It shall be easy to write the dissector. It shall be possible to declare fields of various sizes and types. Said fields shall have the property that their values can be used in expressions. It shall be possible to make expressions, both arithmetic and boolean. It shall be possible to make decisions depending on the outcome of an expression, so that part of the dissector code shall or shall not be run. It shall be possible to define a protocol. It shall be possible to define structures which said protocol can use and reuse.

[1] Formerly known as Ethereal.
[2]Backus-Naur-Form Converter

1.2 Limitations

There are boundaries that will limit what will be implemented. There are 20 weeks of time for the research and implementation for one person. The dissector is going to be programmed against Wireshark, whose API is documented, but there will also be new files included into the build of Wireshark. The programming language of the task is C sine Wireshark is written in that language. The build system uses the make[3] tool which uses an amount of files called makefile to define the structure of the build processes that are needed to compile Wireshark. These makefiles and their structure are not mentioned in the Wireshark documentation in any greater detail.

[3] In the windows world, this tool is called nmake.

2 Common network principles

Network traffic is handled in layers where each layer only knows about itself and minor about the layer above and under them. The OSI-model describes the network layer structure. It has 7 layers, where the top three layers often get bundled together as one, making it 5 layers. [1]

The layers are mostly a theoretical construction since few protocols are strictly in one layer; TCP for example resides both in the application layer and transport layer but is mostly sorted into the transport layer since that is its main purpose. One of the main reasons for constructing this model is to give a reasonable framework to discuss network protocols and issues, another is to reduce complexity in favour of efficiency and flexibility.

This is a brief description from bottom to top of the five layer version.

2.1 Physical layer

The physical layer is the lowest layer in the stack. This is where the physical characteristics are defined. It deals with transmit medium (radio waves, twisted pair cables, fibre optics et c), signal modulation (IEEE 802.11, voltage et c) and other physical attributes. The physical layer has the responsibility of transmitting the individual bits over the medium.

2.2 Link layer

The second lowest layer is the link layer. This is the lowest logical layer and can also be partly implemented in hardware. It handles the transmission of frames and varies depending on different kinds of links. A packet may be transmitted using different link layers before reaching its destination. Examples of link layers are Ethernet and PPP.

2.3 Network layer

The middle layer is the network layer, which is responsible for moving the datagram from one host to another to finally reach the destination using some definition of addresses and networks. Examples of protocols are IP and SPX.

2.4 Transport layer

Next is the transport layer which handles messages from application layer to application layer over the network layer. Examples of transport layer protocols are TCP, UDP and IPX. TCP is responsible for making sure that packets are delivered and presented to the application layer in the correct order, it also has mechanisms for throttling in case of network congestion. UDP is much simpler and leaves such matters to the application layer. IPX is a transport layer which uses SPX as network layer and has the same role as TCP has.

2.5 Application layer

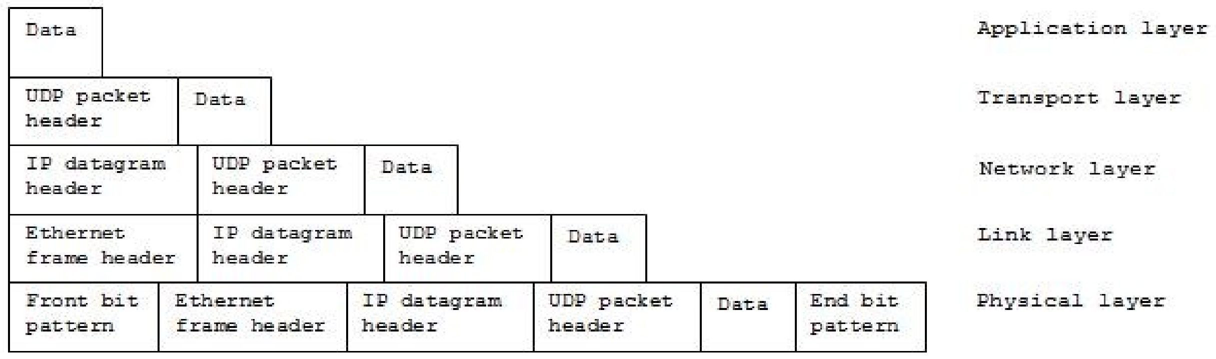
The application layer is the top-most layer and is where applications communicate with each other. Examples of application layer protocols are HTTP which formats web sites and DNS which handles the human readable internet addresses.

2.6 Packet layout

The packet is layered like the protocol stack describes it. The application layer tells the transport layer to transport the data. The transport layer then encloses the data into a packet and adds a header to that. Then it handles the packet down to the network layer which also adds a header to it. This repeats down to the physical layer which sends the packet away.

This has some drawbacks as well as some benefits. The main drawback of is the overhead that all headers produce. The relative overhead diminishes as the data in the packets grow larger. The benefit is that each layer has its own information and only has to deal with information regarding the immediately surrounding layers.

Wireshark uses this hierarchy of layers to display the contents of a captured packet so that the structure of the packet and its purpose gets clearer.



3 Common compiler principles

Compiling source code is done in several steps to break down the problem into smaller chunks and also to gain some additional benefits as the code can be modularised and replaced depending on which programming language or hardware that is used. This is done pretty much in the same way as for the network protocol stack, so that the complexity can be reduced through component modularization. The description herein is made up of three steps, front end, intermediate step and back end, where all parts are independent of what transpires in any other part except for what the input/output interfaces between them produces.

3.1 Front end

The front end of a compiler deals with all language specific items like keywords and syntax. After or under the parse step the compiler will transform the program into an intermediate code representation for further processing by the latter stages. [2]

3.1.1 Lexer

If there is a pre-processing step as text replacement or macro expansion they are done in the lexer before any other processing. This is most common in the languages C and C++ where labels can be assigned values or expressions, which is done as text replacement and will work as if they were written into the code directly. The advantage of this is that macros can be defined, magic numbers will no longer have to be magic but will (or perhaps may) be given names explaining their meaning. The negative side about macros is that they have no name space of their own. This means that if some variable in the macro has the same name as in the function that uses the macro, the variable in the macro will be the same as in the function, which may yield unexpected and unwanted results.

```
new_pos = old_pos + delta_pos
list of tokens:
(id, 1) (op, 2) (id, 3) (op, 4) (id, 5)
```

How a line of code corresponds to a string of lexer tokens.

The lexer then interprets textual input and makes it into tokens which it sends them to the parser. The first action it does is stripping the code of unnecessary white spaces and comments. It continues by identifying lexemes, this is done by matching to keywords and transforming them into tokens. The keyword is either fixed, as the keywords ‘if’ and ‘else’ or is matched into a pattern if it is not a keyword. For every new identifier, the lexer will create a new entry in the symbol table. All this is done using matching to patterns, if it matches a keyword, the pattern is simply the letters of the keyword and otherwise a pattern may be matched by many different strings as in the case of identifiers[4]. In this stage, the structure of the program is unknown, but that will be taken care of later in the parser.

3.1.2 Symbol table

The symbol table is where all symbols are stored for later reference. All defined keywords and other symbols for a language have their own entry from the beginning whereas entries for functions and variables are created during the lexer phase.

Id	Symbol
1	new_pos
2	=
3	old_pos
4	+
5	delta_pos

This is a symbol table based on the last example.

3.1.3 Parser

The parser takes the list of tokens from the lexer and makes checks on the syntax and then transforms the list into a syntax tree. It is also here that mistakes will be generally visible. If someone wrote '=' instead of '+' in , the parser would detect it because that the token for an assignment would not fit into the place where the addition would be. This is possible because when creating the tree, it will sort the different tokens into its positions, and when one token does not fit into the position it should have, there is an error.

Sometimes the parser may transform the tree a few times to get the desired result. Some compilers parse trees from different lexers into the same form so that a more general back-end can generate code. This means that the back-end only have to be written once for each hardware platform.

```
List of tokens:
(id, 1) (op, 2) (id, 3) (op, 4) (id, 5)

Parse tree of tokens:
        (op, 2)
          / \
   (id, 1)  (op, 4)
              / \
       (id, 3)  (id, 5)
```

The transformation from the list of tokens from the lexer to the tree from the parser.

3.2 Middle ground

Some compilers have a middle or intermediate step that will make an abstraction between the front end and the back end. For `N*M` front ends and back ends there is only need to write `N` front ends and `M` back ends. Though, practically, there are some restrictions to this idea, even if the ideal middle language has long been sought after. The GCC uses an intermediate tree structure which works best with C like languages. It is also in this part of he compiler that we find machine independent optimisations. [3]

[4] The usual types of patterns are regular expressions or context free grammars where the former is less powerful but uses less power to compute and the latter is very powerful and is mostly used for the structure of a language.

3.2.1 Intermediate code generation

Most modern compilers will generate intermediate code of some kind. The intermediate code can be a language specifically made for such a purpose, it can be data structures for internal use, or it can be code generated in another programming language and use the compiler for that language as a back end for generating machine code. The early C++ compiler was, for example, a front end that compiled C++ code into C code and then let a regular C compiler be the back end. This is not a rare occurrence since there are a lot of C compilers for a wide variety of hardware platforms.

It is in this step where the semantic analysis takes place.
Programming languages that have static typing usually make type checking variables passed into functions, assignments and other places where the type is of concern for accurate computing. Some languages[5] have the ability to give the programmer the choice to change one type to some other type through casting, i.e. telling the compiler what type a parameter has. Other languages require transformations through specific functions, such as round() or ceiling() for floating point type numbers to integer type numbers[6]. Other languages uses dynamic typing which usually is a complex exercise where the runtime environment discovers what type a variable or parameter has[7].

The semantic analyser also checks if variables, functions et c has been declared or defined so that they are usable. If a variable that does not exist is used, then the programmer has either made a typo or forgot to declare the variable. In languages without specific declaration of variables, there may be additional errors if the programmer misspells a variable name[8].

3.3 Back end

The back end optimizes code for the hardware and generates the machine code for it. If the hardware has parallel capabilities there might be steps to optimize further for that, the same is true for pipe lining and cache handling. This part of the task is not considered for this task since the protocol "code" will be interpreted. It will be done by the compiler for the dissector code when it compiles into the Wireshark library.

3.3.1 Machine code generation and optimization

In this stage, the compiler takes the intermediate representation together with the symbol table to produce the final machine code. This should be done efficiently, both the back end and the target

[5] C/C++, Java, et c

[6] Functional languages as SML and Haskell

[7] Smalltalk, Ruby

[8] Prolog is such a language, since the functions may be called "backwards". A new parameter can be sent into a function to obtain a result and may then be used later on, but if the function would behave differently if the parameter already exists there could be a difficult task to track down that specific typo.

program has to be resource effective. Not only should more efficient instructions be used, there is also a question about how to allocate the registers[9] in the CPU and administer loading and saving to slower memories. These problems are mathematically difficult, but there are good approximation techniques to be used so that the end result will be good but probably not optimal. [3] [4]

[9]A register is the fastest piece of memory in the CPU, where values are available instantly for processing.

4 Analysis and design

4.1 *Current system of development of dissectors*

To develop a dissector for Wireshark, one has to make it either as a plug-in or compile it into Wireshark's dissector library. Plug-ins may have to be remade between versions of Wireshark because of behavioural changes or changes in the plug-in interface. Code that is submitted into the dissector library is maintained by the Wireshark developers but would be considered free due to the GNU Public License and the written code could be used or analysed by anyone who wanted to.

In the case of patching, which essentially is the same as writing an ordinary dissector, one would need to re-compile Wireshark, which includes setting up a development environment. The source would not be spread widely and the protocol would be kept secret. Of course, this would have to be done every time there was a change in the protocol or when an upgrade of Wireshark is deemed necessary!

When it comes to plug-ins, they have to be compiled by the same compiler as the main program, which means that if you do not have the same compiler and version of it, you may have to re-compile the entire program to make it work properly. The other obstacles are that the Wireshark plug-in architecture is not stable and may change between releases and that not all operating systems that Wireshark runs on can support plug-ins.[5] The advantage is that it is easy to install and it is a lot faster to compile and build.

Since Wireshark is developed against multiple operating systems and hardware platforms care would also have to be taken so that the code will to be general enough or be adjusted to accommodate the different systems. This means that different versions of C, C compilers, byte-ordering et c has to be considered when the code is written.

4.2 *Future development of dissectors*

Instead of recompiling the program or making new plug-ins when a protocol changes, one would simply have to change the contents in a text file, which the program then would read and analyse to know how a protocol is specified. This would make it easier to maintain private or proprietary protocols and information since the protocol definition file itself would not have to be under the GPL. It would also make debugging and changing a protocol and easier task since the procedure only needs that the protocol definitions file to be edited and Wireshark restarted for the changes to take effect.

4.3 Research

The research began by looking into XML since it is widely known and used for similar purposes[10]. The tool BNFC was mentioned by the examiner so it was also included into the research. The last and not insignificant step into the research was to research the Wireshark read me files, especially README.developer which contains a lot of tips and tricks for developing for Wireshark. A parallel step to the research was to set up the development environment[11].

4.3.1 XML

XML was chosen as a candidate since it is a widely known language. Wireshark uses the library libxml2 which is an open source XML parser. It is used, among other , for describing the different attribute-value-pairs that is used by the Diameter protocol and some different applications on top of it and also to describe similar structures for the Wimax protocol.

```
<PowerButton>
      <Shape diameter=”3.14”>Circle</Shape>
      <Lighted/>
</PowerButton>
```

Example of XML tags

XML is a context-free language and is designed to express structures in text format. It is widely used for web-applications and also numerous other applications. XML itself is expressed by using start tags and end tags, where the end tag is identical to the start tag except for the slash (/) in the beginning of the end tag. Between a pair of tags there can be either information or one or several pairs of tags. There is also a short version of a tag pair if it does not contain any information, it will only be one tag ending with the slash sign.

```
if ( <expression>) {
      <declaration> ;
      <declaration> ;
}
else {
      <declaration> ;
}
```

Example of code that can be produced by BNFC.

There exists several schema tools for verifying that an XML document has the right grammar where the most common is DTD[12]. In the DTD there will be a specification of the grammar that a document has to follow. To tell what DTD to use, there is a possibility to choose DTD by adding a line in the document file. The specification is made in a CFG[13] which have a notation that reminds of regular expressions but has the power of context free grammar.

[10] It is used for the Diameter AVP (Attribute Value Pair) decoding where the AVPs' values and contents are described
[11] See Appendix D
[12] Document-Type Definition
[13] Context Free Grammar

4.3.2 BNFC

BNFC [6] is an open source research program that takes a text file with a specification of a language, in form of labelled BNF grammar, i.e. labelled context free grammar, and can create output in a variety of languages such as Haskell, C, Java et c. BNFC generates the front-end of a compiler from the rules that are specified from the file it takes as input and generates a lexer, a parser, a pretty printer and skeleton files for further expansion by the programmer[14].

The language has to be specified in labelled Backus-Naur Form, which is a context free grammar where all different production rules have different labels.

BNFC has some pre-defined tokens in form of basic types. Some examples of these are `String`, `Ident` and `Integer`. If these basic types are not enough, more tokens can be made using regular expressions. [7]

```
token TokenName <regular expression> ;
```

```
token Hexadecimal '0' ('x'|'X') (digit | ["abcdef"] | ["ABCDEF"])+ ;
```

The rule for making rules in BNFC is:

```
Label . Category ::= Production ;
```

The label has to be unique while the category groups rule labels together. The '::=' corresponds to the production arrow in ordinary CFGs and to the right of it are the productions where tokens or categories can be declared forming production rules. To end a production rule you have to add a semi-colon, this makes it possible to spread an advanced rule over several lines in the specification.

```
Add. Exp  ::= Integer "+" Exp ;
Sub. Exp  ::= Integer "-" Exp ;
Int. Exp  ::= Integer ;

Exp -> Num '+' Exp |
       Num '-' Exp
Num -> 0 | 1 | 2 .. 9 | Num Num
```

Top: Labeled BNF production rules for BNFC
Bottom: CFG production rules

In there are the labels Add, Sub and Num. These are grouped together in Exp to make expressions. Even if there only is one production rule it has to specify both label and category, though in that case, they may have the same name. There are possibilities of making list specifying intermediate symbols or terminating symbols for each element in the list with the keywords separator and terminator respectively. Lists can also be made by hand using recursive calls to the category if there is a need to have a special behaviour. One such example is the above Exp category which uses the recursive call to define behaviour instead of listing items.

[14] BNFC produces code in C, C++, Java or Haskell depending on input parameters. Haskell is the default choice.

A more advanced example is an if-statement. Here we introduce several categories into the production and also lists of categories and definitions of keywords and parenthesis.

```
terminator ";" Decl ;
If.   IfStm ::= "if" "(" Exp ")" " {" [Decl] "}" ElStm ;
Nil.  ElStm ::=                                        ;
Elif. ElStm ::= "else" IfStm                           ;
Else. ElStm ::= "else" "{" [Decl] "}"                  ;
```

If-statement in labeled BNF

shows code that can be accepted by the grammar in . First there is the IfStm which takes an expression and any number of declarations, where each declaration is ended with a semicolon.

Then it goes on to the category ElStm where it decides on the production with the label Else since it uses the keyword else and does not start over at IfStm.

4.3.3 Lua

Wireshark has built-in support for Lua, which is a general purpose scripting language that uses external bindings and functionality to execute the script and affect the surrounding environment. [8] Lua works embedded in the host program and functions in Lua can be called from the host program and vice versa. Lua supports the different programming paradigms procedural, object oriented, functional and data driven programming.

4.3.4 Choice of language

The dissector has to be written in some language and as a part of the research XML, a tool called BNFC as well as the Lua facility that Wireshark has was looked into.

Lua was disregarded as an option fairly soon because it uses bindings written in Wireshark which is licensed under the GPL. This could make any code in Lua that uses those bindings to be bound to the GPL also. This would make it undesirable as the language to use in this assignment since one of the main reasons was that the protocols are proprietary.[9]

The main advantage of XML is that there already exist several XML-tools, several of who is free to use and have licenses that are compliant to the GPL. Another advantage of XML is that it is general and you can define how you want to use it yourself. To make XML behave as wanted, there would have to be a DTD or roughly equivalent XML schema specified to check if the XML-tags were well formed and correct according to a syntax scheme.

XML is a very verbose language, which, for the purpose, can be considered to be a problem. The structure of XML requires a lot of tags that makes writing and reading obfuscated, the latter reasons which are quite important for files edited by hand. It is also very different from the language C, which is a common language and also (lexically as syntactically) the base for many other languages such as C++ and Java.

The tool BNFC takes a file containing a specification of the language in labelled BNF and generates code that creates an abstract syntax tree, a pretty printer for the abstract syntax tree and also files with skeleton code. The specification file would be equivalent with the XML schema. Up to this point, these two approaches seem about equivalent. However, BNFC gives a far greater ability to define the lexical and syntactical structure of the language and the skeleton code is also already made to traverse the abstract syntax tree and that facility had to be done by hand for the XML option.

When defining the lexical and syntactic rules in BNFC, the space needed is much less than if the same rules would be written in Flex or YACC, down to about 1-2% in comparison.[10] Since the amount of code is so much smaller, the risks for programming faults are much less prominent.

The choice fell on building a new language using BNFC. This because that the labour would be approximately the same for the end language and easier for each iteration the language underwent, and the written code would be easier to read and modify. Also, the ability to iterate the design of the language several times without having to rewrite functions for traversing the syntax tree also greatly influenced the decision in BNFC's favour. This is made easy with BNFC since the functions are pre-generated and all that needs to be added is the wanted functionality.

The language was made C-like using declarations, if-statements and structures in a fashion that is similar to C. The design makes it easy to read and understand and translation from C structures is easy where additional logic, as dynamic arrays and conditional statements, can be applied when and where it is needed by the programmer.

4.3.5 Method of development

The project started by creating a real dissector using the guidelines in the file README.developer[15] that is bundled with the source code. The dissector was basic and only written to understand how to create a regular protocol dissector in C. The second step was to

[15] This and other documentation files are included when in the downloaded Wireshark source code.

create a small language[16] that would be easy to implement, so that it could be determined that it was feasible to create a better and richer language that would be usable.

The largest challenge with this step was to modify the build so that the dynamic language would be built and included into the Wireshark dissector library. There were two reasons for this; first, the build process is quite complicated, where each and every folder has its own makefile[17], often for several environments and some had common files for inclusion into all environment-specific makefiles. All makefiles has dependencies to other files and it was not uncommon that to find a variable used in a makefile there had to be a search for three or four makefiles in parent directories to find these. It has become easier since then because it created a better understanding of the building process of C source files.

The other large obstacle was the debugging. During the development of the parser, simple print functions sufficed since the structure of the program was simple. There was enough versatility to test different angles of a problem without any impact on other code, so the code was rather safe before it was integrated into Wireshark. When the code was integrated into Wireshark, printing the results and debugging was not a good enough solution since Wireshark do not print to an ordinary console[18]. There was an effort to try and integrate the Wireshark code into a Visual Studio project so that the project could be stepped through from the start of the Wireshark process. This integration was needed since most of the execution in that part of the code happened during the start up of the process. This failed however and there was some effort to write debugging information into files instead of showing it on a console. This solution worked well with the simple test language. There is also a way to print text in the dissector window, but this is not available until the packet dissection.

4.4 Development

4.4.1 Using the makefile system

A program written in C needs to be compiled and linked together. This can be a large process and would be overwhelming for most projects if there wasn't any tool to do this automatically. The most common tool for building C-programs in UNIX-like systems[19] is make. Its equivalent in the Microsoft world is called nmake and follows almost the same rules as its UNIX cousin does, which makes it necessary to have separate files configuration files for these two environments.

[16] See Appendix A
[17] A makefile specifies the building process and is used by a tool called make in UNIX and nmake in windows as configuration. As what files should be compiled and how, what libraries should be linked, where additional resource files are etc.
[18] Wireshark actually has an error console, but it was not know at that moment of development.
[19] All flavors will from here be called UNIX.

All the rules for the compiling and linking of the program shall be defined in the make configuration file, suitably called makefile. In this file you can specify which files that are going to be compiled or linked into the executable and which files are going to be compiled or linked into the libraries, which will be called from the executable or other libraries.

4.4.2 Wireshark documentation

The documentation when developing for Wireshark consists mainly of documentation in code and some REAMDE files. The documentation in the code can be extracted by using the tool Doxygen, which picks out comments which are marked in a special way and converts it into html pages with direct links between the different sections so that definitions of types and functions are easily found.
The README files contains some tips and tricks within some field, like the README.plugins which contains information more specifically of how to write plug-ins, while README.developer is more general for developing.

4.4.3 Other miscellaneous obstacles

The PATH environment variable in Windows has no officially “best before” length. However, to be able to build with Microsoft Visual Studio, the PATH had to be limited in length, else some mysterious errors would occur that did not show any reference to the PATH length. The solution was found during extensive search in the Internet.

The glib (GNU library) which has its own type definitions and is used quite extensively throughout Wireshark. The great advantage of using the glib type system is that it can guarantee a certain type to have a certain size on all hard wares, a feature that is very convenient in a cross platform program.

4.5 Identifying needed specifications

When the language was created there was a need to know what was needed for the purpose, how the syntax should be created, what keywords to use and if anything should copied from other languages. Since the language shall describe the structure of information in network packets, it was decided that it was needed to be able to declare fields, sizes and types, since information comes in a variety of them. There is a number of constructs in the language that was created. Every construct that might be needed and the reason for having or not having them in the language have been listed. The constructs were inspired by the Wireshark protocol dissectors and the layout of C data structures.

4.5.1 Protocol

This is the main entity, where all the declarations for a protocol are made. In the beginning there was some consideration about having the possibility to define several protocols. However, it was decided against later because of time constraints. The main reason to why there is a need to specify a protocol is that there are protocol specific information that has to be declared, as sub layer interface and names and filters. It is also handy in the case that the extension will support more than one protocol in the future. There is also constructs to later make it possible to add extra properties to the protocol if that would be necessary.

4.5.2 Declarations

The most basic construct is the declaration, which from the beginning was a size and a name and has later on been filled with extra information. It is needed to show what information that the protocol is containing and at which position. It was also considered to add extra properties for describing a checksum, different kinds of addresses and so forth so computation could be done automatically. There is a special declaration to print a small text snippet.

4.5.3 Field sizes

The accepted sizes of fields are 8, 16, 32 and 64 bits. These are the most commonly used sizes of information elements. However, a need for the ability to use whatever size the user wanted was seen, among other uses for flags. The idea was to declare a size in bits using a number between 1 and 64 or even 128 depending on how well the implementation went. This proved difficult to do and would have taken a lot of time to complete even if the field would be included in another, larger field. The original idea was to have the field standing in its own right and causing additional complexity using bit sized alignment problems instead of the usual byte alignment problem.

4.5.4 Arrays

The arrays can be either static in size or dynamic depending on an expression. Any declaration can have the array extension. Together with structures, this makes it possible to easily define large masses of structures.

4.5.5 Loops

There was an idea about including a looping functionality so that data could be iterated through. But since this would add a lot of additional complexity and the need for loops were considered to be very low this idea never took hold in the design. Most of the problem was solved anyway with arrays with dynamical lengths. This is the only language construct in this list which does not have a definition, implemented or not, in the language definition.

4.5.6 Length

For loops to work there would also have to be some kind of length specification of how long some collected fields or structures or even the remainder of the protocol. The other alternative would be a for-each-like construct, but it was replaced by the dynamic array.

4.5.7 If-else

Since a protocol can include a lot of options there should be a way to choose a right path through the declarations. An if statement takes a boolean expression and then chooses a way depending on the outcome.

4.5.8 Switch

The switch-statement is handy when dealing with a large amount of possibilities depending on the outcome of an expression, since it structures the code into convenient chunks that are easily recognized. The implemented switch statements would be not more than some if else statements in a row, making this feature unnecessary except for syntactic sugar purposes.

4.5.9 Expressions

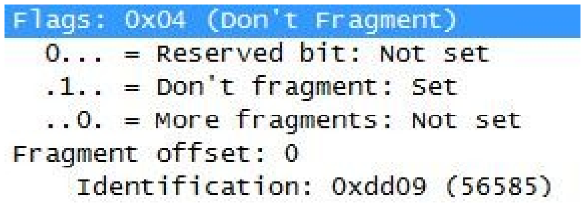
Since there was a need to have a way to define sizes in a dynamic fashion and also be able to branch expressions were needed. It was decided to use boolean expressions and arithmetic expressions as two different kinds of expressions so that there would be some extra safety in programming and making the code easier to read. Boolean expressions can be created by comparing arithmetic expressions.

4.5.10 Structure

The structure declaration was modelled on how the protocol was declared but is somewhat lighter and can be used if there are structures of information in the protocol that repeats itself or can be lifted out for clarity. A structure can contain any declarations as a protocol.

4.5.11 Flags

For a while there was an attempt to create a structure to declare flags for a field but it had to be discarded it because of time constraints. It is still possible to do with if-statements and the display declaration but the end result is not as pretty.



```
Flags: 0x04 (Don't Fragment)
  0... = Reserved bit: Not set
  .1.. = Don't fragment: Set
  ..0. = More fragments: Not set
Fragment offset: 0
    Identification: 0xdd09 (56585)
```

Example of flags from the IP-protocol

4.5.12 Protocol type

There is a possibility in Wireshark to change the displayed title of the protocol so that there is a possibility to show that it is a special kind protocol packet[20]. This possibility was removed since the implementation was ambiguous and the possibility for very long titles was ominously large.

4.5.13 Hand off

I wanted an ability to finish the dissection and hand the over to another user defined protocol. This construct would use a string which would be the name of that protocol. This was also cancelled because of time constraints and the limitation of one protocol in the protocol definitions file.

[20] A FTP packet could be displayed as *FTP signalling* or *FTP payload*, using the options of adding the word signalling or the word payload.

5 *Defining a protocol and its structures*

The first things to be defined for protocols and structures are their respective general properties. The common properties for protocols and structures are the long name, the short name and the abbreviation. Protocols also need the extra property called sub dissector which describes which lower level the protocol relies on, e.g. a TCP port and a port number. The definition of structure also requires a name for the structure that will be used when the structure will be used in a declaration. The abbreviation is used for the filtering function in Wireshark. No protocol or structure may be recursive in nature; it is possible to do but will lead to unexpected results.

```
protocol {
      name = "Protocol Foo";
      short = "Prot Foo";
      abbrev = "prot_foo";
      sub_dissector = "tcp.port" 12345;
}
{…}
```
The header of the protocol Foo.

```
structure structure_bar {
      name = "Structure Bar";
      short = "Struct Bar";
      abbrev = "struct_bar";
}
{…}
```
The header of the structure Bar.

5.1 *Fields and declarations*

Fields can be used as a constant in the meaning that the field contains a value which it fetches from the payload that is being dissected and that value will never be changed for the scope of the field. This structure demands that the field is declared before it is used as a value in an expression. It is not possible to access a field in another structure or protocol.

A declaration is made in the form

```
      Length Type Id Description Filter Offset;
```

`Length` is declared as `8bit`, `16bit`, `32bit` or `64bit`. It is used for the size of an individual part. If the field is an array, each element of the array has that size.

`Type` is the type of the field. There is a choice between `char`, `float` or `int`. For char and float there are limitations in lengths. `Char` can only have the lengths `8bit` or `16bit`. The `8bit` length will use an ASCII-decoder while the `16bit` will use a fake Unicode-decoder due to lack of support for Unicode in Wireshark. `Float` can only use the lengths `32bit` and `64bit` and will interpret the data as floating point numbers as specified by the IEEE. `Int` can use all sizes but require extra parameters. These parameters have to do with the display properties of an integer. There are the possibilities to display as octal, hexadecimal or decimal (`OCT`, `HEX` resp. `DEC`). The latter with the options signed and unsigned. There are two more parameters, `HEX_TO_DEC` and

`DEC_TO_HEX` who are a mixture of decimal and hexadecimal, where there is also a need to specify the `signed/unsigned` for the decimal display.

`Id` is the identifier for the field and is the name that will be used if the field's value will be used later on. The Id can also be extended by `[]` to declare it as an array. It is possible to set a fixed number for the array size but it can also be a field value or an expression, e.g. `identifier[3]` or `identifier[length/8]`.

`Description` is a double-quoted string that is used when that field is displayed in Wireshark.

`Filter` is also a double-quoted string, but requires lower-case letters, dashes and underscores only. It is used by Wireshark to filter packets by value or existence.

`Offset` is optional and designed as on, off and a default value as on. Use the boolean values `true` or `false` to activate other than the default behaviour where true is on and false is off.

The use of a structure is declared:

```
      structure StructureName Id;
```

`structure` is the keyword and shall remain as is.

`StructureName` is the name that was given when defining the structure. The structure can be defined before or after the use of the name.

`Id` has the same definition as above.

To display a string
you can use the command display followed by a string in double quotes. The string cannot be formatted but will be shown exactly as it is written.

```
      display "String to be displayed."
```

If-statements
In the parenthesis there shall always be a boolean expression at the top. The expression can be a comparison of two arithmetic expressions. All declarations resides between the curly brackets. The statement does not have to have any else cases, they are entirely optional, the else if case is possible to enter several times and the last else case can only be done once at the end of an if-statement. It is possible to do nested if-statements.

5.2 Expressions

Operators in expressions that has the same precedence in expressions are left associative, which means that they start evaluating from the left side if two operators have equal precedence.

If there are two operators with the same precedence working on the same value, the left part will be executed first. If it is preferred that another part should be calculated first then it can be put inside parenthesis. Parenthesis also limits precedence.

```
if ( length/12 < 8 )
{
    <some declarations>
}
else if ( length/12 >= 10 )
{
    <some other declarations>
}
else
{
    <yet some other declarations>
}
```

BinExpr is an arithmetic expression using binary operators. The ones specified are addition, subtraction, multiplication, division, bitwise Or and bitwise And[21] `(+,-,*,/ ,|, &)`. Multiplication and division share the same precedence and their precedence is higher than the other four operators', who share the same precedence.

BoolExpr is a boolean expression where the answers are either true or false. The different operators are Not, And, Or and Implication (`not`, `&&`, `||`, `->`). These have the same precedence.

5.3 The scope of declarations

Declarations are only visible in the scope of their structure / protocol and called structures will not be able to access those field values as shown in .

```
Structure s1{
      /* Field is not accessible in this scope
         neither before or after the declaration
         of s2 */
      Structure s2;

}
Structure s2{
      //Field is accessible in this scope
      Field;
      Structure s3;
}
Structure s3{
      //Field is not accessible in this scope.
}
```

How the scope of declarations work.

[21] Bitwise And and bitwise Or are operators that compares each bit in a field to the corresponding bit in another field. For the And operator, the bit in the result will be set if and only if the corresponding bits in the two compared fields are set. For the Or operator, the bit will be set if at least one of the two other bits are set.

6 Interpreter flow

Some notations, items in `courier new` depicts names of functions or data structures which is described.

6.1 Initial sequence

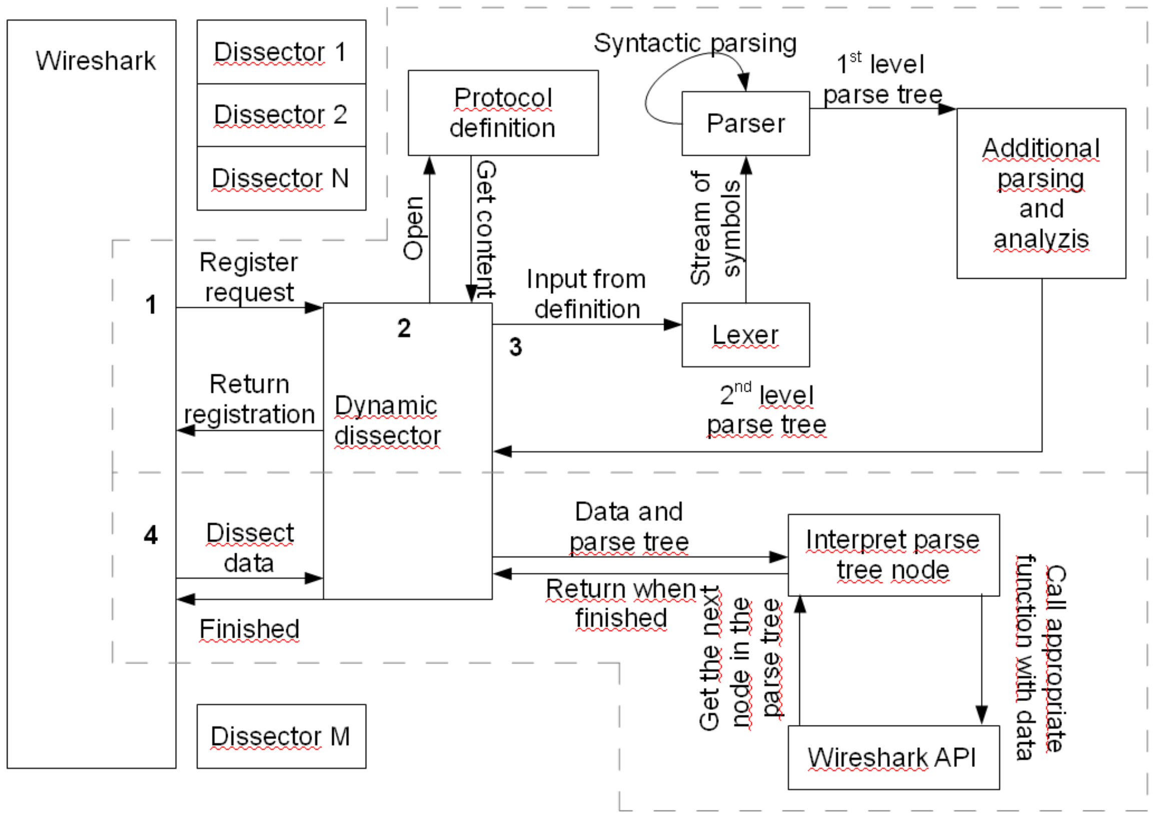
When the dissector is being registered, it opens up the protocol file and reads it. It will send the text through the lexer and parser and if the code is correct, it will create a parse-tree. These steps are done in the code generated by BNFC.

In the next step, the interpreter will enter the hand-coded region. It will create more intermediate data structures to hold information from the parse-tree as well as extra information that are needed on beforehand, either as hard requirements from the Wireshark API or to speed up later processing by calculating what constants there may be such as lengths and sizes from expressions. The interpreter will then continue to create other parse-structures and will do checks for duplicate declarations of fields or if there are fields that have not been declared yet that are used in expressions.

The interpreter will also do type checking. If an expression only has constants and no variables, the interpreter will calculate them on the spot since constants only have to be calculated once, this will save work later on since the same dissector can be used for several packets in the same capture in Wireshark.

6.2 Wireshark registration

After these steps the Wireshark interface takes over and traverses the structures created earlier in the interpreter. It creates new structures with information that has to be registered in Wireshark for later uses. Information includes sub layers in the protocol stack, which patterns to use in these, information about fields for later retrieval when presentation of the fields in question are to be done. When reaching a structure tag, the dissector will create a sub tree, so that the information in the structure is collapsible in the GUI in Wireshark. The information about the sub tree is also being registered. All this information got their handles which Wireshark later uses in conjunction with calls to the Wireshark API for presenting the information in the fields.
When the dissector later is called to dissect some data it once again traverses the parse-tree to interpret the structure of the protocol, it saves values for the variables, evaluates expressions where needed and evaluates if-else-statements and presents structures.



When the Wireshark process is started, it calls every dissectors' register function and among these is `proto_register_dynamic_protocol`.

A description of the flow of the dynamic dissector.
The top part covers the registration process and the lower part covers the dissection.

The register function `proto_register_dynamic_protocol` opens the file with the protocol definition, dynamic.prt, which is located at the same place as the executable.

It then calls the entry point function for the parser, `pDefinitions`, with the file pointer as an argument. The auto-generated lexer and parser will create an abstract syntax tree, the 1st level parse tree in the figure.

In the second pass, all information that has to be available in structures that are going to be registered in Wireshark will be created and populated and when the last pass is over, the structures are registered[22] and the 2nd level parse tree is created.

The end of the registration is actually a second call to the dynamic dissector, but since it also is a part of the registration phase, it is treated as one phase. In the call to `proto_reg_handoff_dynamic_protocol`, the dissector function will be registered together with the handle of the protocol, which was received in the call to `proto_register_dynamic_protocol`. It will also register the protocol from the underlying layer and a pattern for identifying what protocol dissector that should be called.

In the second pass, a call to the function `collect_everything`, all the information that has not yet been parsed will be so. Some information will have to be converted, such as integers and floating point numbers, and in the case of the dynamic dissector, all new tokens, hexadecimal, octal and negative integers. All tokens that are built into BNFC are already converted into the right unit, while the new tokens are converted "by hand". The function returns a data structure containing a pointer to each top structure in the protocol definition file.

6.2.1 Top most routines

```
typedef struct {
      protocol** protocols;
      guint no_protocols;
      structure** structures;
      guint no_structures;
} everything;
```

The function `collect_everything` calls the two functions `visitStructure` and `visitProtocol` via the function `visitDefinitions`. It is the last of the three functions which populates the arrays in the data structure `everything`. The function `visitDefinitions` also calls the function `visitListTypes` that collects all the contents in the protocols and structures, and add that to the respective structure or protocol structure. The `visitDefinitions` function and `visitListTypes` are recursive and traverses the linked lists which are provided in the 1st lever parse tree.

The `structure` and the `protocol` data structures are quite similar since the structure construct have a subset of the responsibilities as the protocol has. These are allocated in `visitStructure` and `visitProtocol` respectively. They are used throughout the entire dissector code for getting and setting the values of fields and creation of the dissector tree. In all information bearing data structures beneath everything, the enumeration kind is used to be able to differ the different structures between each other.

[22]The most important structure is the header field information, which contains information about name, abbreviation, size, data type and display type for a single field.

```
typedef struct {
      defs                kind;
      base_properties*    b_prop;
      ret_type*           a_prop;
      ret_type*           subdiss;
      list_types*         l_types;
      gchar**             var_names;
      guint               no_vars;
      var_value*          variables;
      guint               tot_size;
      dissector_handle_t  handle;
} protocol;
```

The `b_prop` is a structure which contains the common properties between a protocol and a structure. `a_prop` is a list with additional properties, which are not yet used, but the list is populated and ready to use if needed. The `sub_diss` field contains the sub dissector information that is registered so that Wireshark know when to call the dissector for a specific protocol. The `list_types* l_types` will be described later. The

```
typedef struct {
      defs               kind;
      base_properties*   b_prop;
      ret_type*          a_prop;
      list_types*        l_types;
      gchar*             id_string;
      gchar**            var_names;
      guint              no_vars;
      var_value*         variables;
      guint              tot_size;
} structure;
```

structure uses `id_string` for unique identification of each structure. The array `var_names` contains all variable names and is also used to look up which entry a variable has in in the array `variables`, when a variable's value is to be set or fetched. The `no_vars` integer is the size of the variable arrays, which is dynamically allocated. The `variables` array contains data structures which contains the value and what type it has. The integer `tot_size` is the maximum amount of declarations that can be executed from the protocol or structure, that is, it is the total number of declarations in each. The `dissector_handle_t` is the handle of the protocol that Wireshark uses.

6.2.2 Body of declarations

```
typedef struct _list_types {
      defs                kind;
      gpointer            data;
      struct _list_types *next;
} list_types;
```

The `list_types` is a linked list structure, where each element in the list has a kind to describe the information, a pointer to the information and a pointer to the next element in the list. It is allocated and populated in `visitListDeclaration` and `visitListTypes`. It is used in the functions `construct_hf_array`, `proto_register_dynamic_protocol`, `create_dynamic_proto_tree`, `reset_list_types`, and `reset_pointers`. The kind field is one of `handoff`, `declaration`, `value_dep` and `ifelse`. `Handoff` is in the case that a handoff to another protocol is defined, but it is currently not used for anything within the dissector. The `value_dep` value is used for the switch-statement, which is also not currently supported. The values `declaration` and `ifelse` is used however and are the cornerstones of the dissection.

```
typedef struct _ifelse {
      defs              kind;
      guint             position;
      expression*       expr;
      list_types*       l_types;
      struct _ifelse*   next;
} ifelse;
```

The structure ifelse exists to hold the if- and else-statements. It is allocated in the functions visitIfStatement and visitElseStatement and used in construct_hf_array, create_dynamic_proto_tree, reset_list_types and reset_pointers. The position field is there mainly for describing the position in the debugging log-files, since the verbose error mode of the parser is a bit sketchy when it describes the error. However, the position is not that well defined and every list of declarations beneath an ifelse has their position starting at 0. After the ifelse, the counting continues as it did before. The ifelse also has an expression. If the expression evaluates to true, then the dissector should continue to dissect from the l_types in that ifelse instance and afterwards skip the rest of the ifelse list.

```
typedef struct _declaration {
      defs kind;
      guint position;
      guint64 bytes;
      guchar bits;
      expression *mult;
      gchar* id_string;
      guint id_num;
      gpointer data;
      display_type* d_type;
      gboolean inc_offset;
}declaration;
```

The declaration structure is used for saving all relevant data about a declaration. It is mainly information from this data structure that is transferred over to the Wireshark specific data structure hf_register_info, which is registered and used in the dissection when creating a new entry for a specific field. It is used for ordinary declarations as fields, arrays, structures and display statements. It is allocated and populated in the visitDeclaration function and used in the functions construct_hf_array, proto_register_dynamic_protocol and create_dynamic_proto_tree. There are two fields for length, one for bytes, and one for bits. This is because originally, there were plans to make two kinds of field and array statements. The first, and easiest, was to have fixed length fields and arrays, where the size of each part would be 8, 16, 32 and 64 bit long. The other variant would only take a number in bits in front of it and regardless of what number it was make it work. That was the real problem though, and it never took off. The mult expression is to determine the length of an array, if such was declared, using mult as a multiplier of the previously declared length. The id_string is the name of the field. The id_num is a reference number to which structure or protocol that the declaration is made in. The data pointer is used for the description and filter in the case of ordinary declarations. For declaring the use of a structure, the data field is only used for an identifier for the structure. The d_type is a description of how to display the information. A number can be display as decimal, signed or unsigned, hexadecimal or octal, while there also is an option to display a field as a

character (an array of characters makes a string). The `inc_offset` field only determines whether to increase the offset in the dissected buffer or to stay on the same point.
A display statement does only use the `kind` field, the `position` and the `id_string`, which in this case contains the message to be printed.
If a structure is declared, the dissector will create a sub tree with the underlying information, so that in the visual presentation, the information can be expanded or collapsed.

6.2.3 Expressions and values

There are two main functions to be used when evaluating an expression. The first is `visitBoolExpr`, which evaluates a boolean expression. The function `visitBoolExpr` is not called during the initial passes, but only when dissecting a packet. This because it cannot handle a case with a field value. The expression is saved though, and will be evaluated for each dissection. The `visitBoolExpr` function uses some helper functions to do the actual calculations. These functions are `boolean_and, boolean_or, boolean_not, boolean_impl` and `boolean_comp`. The last one takes two binary expression and a comparison flag and produce a boolean, the other ones take takes two booleans. All these function returns the value in a new instance of `const_value`.

The second function is `visitBinExpr` which evaluates an arithmetic expression. This function is called during the initial phases, unless it is a part of a boolean expression, and will produce a result if there is no field value in the expression, otherwise nothing will be produced and the expression will be saved for further evaluation. The function uses these helper functions for the actual computations: `add_const_value, multiply_const_value, subtract_const_value, divide_const_value, or_const_value` and `and_const_value`. The last two functions does bitwise and and bitwise or of the two values. All these takes the result of two binary expressions (two instances of `const_value`) and produces a new instance of `const_value` as a result.

```
typedef struct {
      defs     kind;
      guint32  type;
      gpointer value;
} const_value;
```

A result will yield an instance of the data structure `const_value`. It contains a `kind` field to indicate if it is a variable or a constant value. The `type` field is a collection of flags that indicates what type the value has so that it can be interpreted, and used, the right way. The value is stored in the other end of a pointer for the only reason the the author was unaware of the union construct, which can let several types share the same memory location. A written constant in the protocol definition will also be encapsulated into this structure.

There are two functions for retrieving the value from a `const_value` structure; `get_int_const_value` and `get_double_const_value`. The former will return any C integer value, including booleans and characters in 64 bit format, while the latter will only return floating point values in double format, which is also 64 bits. This means that, internally, the interpreter saves all values in 64 bit format.

6.3 Wireshark dissection

This is a call to the actual dissection of the payload data. The call is made to the function `dissect_dynamic_protocol`. It will traverse the 2nd level parse tree, which was collected in the structure `everything`, analyse the packet data and display the value in the packet data as the information in the parse tree dictates. It is here that the header field information is important as it contains all the information that is needed for the field to be displayed correctly.

When the last step has completed and the dissector has built the dissector tree, all variables' values will be freed by calling the function `reset_pointers`. Otherwise there could be collisions when the new values should be stored. This function in turn, will call the function `reset_list_types`, which will go through the 2nd level parse tree and free all results from expressions that have been calculated. If this call was omitted, all subsequently evaluated expressions would yield the same result, and the same path would be chosen through the parse tree every time even when the values changes.

7 Examples

7.1 Protocol example

```
//This is an example protocol with two structures.

protocol {
/* These names are required for different uses in the Wireshark presentation
of a protocol. */
    name = ”Example 1”;
    short = ”Ex 1”;
    abbrev = ”ex1”;
    sub_dissector = “udp.port” 1234;
}{
    32bit int HEX type "Type" "type" false;
    if(type & 0x0F == 1) {
        structure exstruct1 structure1;
    }
    else {
        Structure exstruct2 structure2;
    }
}

structure exstruct1 {
    name = “Example Structure 1”;
    short = “Ex Struct 1”;
    abbrev = “exstruct1”;
}{
    32bit int HEX type "Example Structure 1 Type" "exstruct1_type";
    if(type & 0x10 != 0) {
        16bit char unistring[16] “Ex Struct 1 Unicode” “exstruct1_unistring”;
    }
    else {
        8bit char asciistring[16] “Ex Struct 1 ASCII” “exstruct1_asciistring”;
    }
}

structure exstruct2 {
    name = “Example Structure 1”;
    short = “Ex Struct 2”;
    abbrev = “exstruct2”;
}{
    display “Example structure 2 is the same as Example structure 1!”
    structure exstruct1 struct1_again;
}
```

7.2 Execution example

An execution example using the protocol in 7.1 and the following payload.

```
87 46 53 21    48 65 65 6c
6c 6f 6f 6f    6f 20 77 6f
72 6c 64 21
```

The lower layer was the transport layer using the user datagram protocol on port 1234. Since the example protocol was registered on those premises, it gets chosen to do dissection of the remaining payload.

The first item in the `list_types` list is a declaration of a 32 bit integer called type, which gets the value of `0x87465321`.
The second item is an if-else statement, it does a bitwise comparison to `0x0F`, which is the 4 lowest bits. If only the lowest bit is set, then the first clause is evaluated otherwise, the second will be. However, since the values are spelled out in hexadecimal form, it is easy to see that only the lowest bit is set and the procedure then enters the first clause.
The next item to be scrutinized is the structure `exstruct1`, and since the declaration of type included the false statement about increasing the offset of the dissection, the dissection of the structure will begin at the same location as for the field type.
The structure also has a type field and it will get the same value as the type field in the protocol definition. And as in the protocol definition, this will also be compared in a similar way, however, this time, the comparison returns false and the procedure continues in clause two.
This declaration is of an array of 16 8bit characters, which will be printed as a string of ascii characters. In this case the string will be "`Heelloooo world!`".
That was the last declaration, so the dissector will return,

8 Algorithmic performance

There are mainly two types of data structures for storing multiple objects used in this dissector. First, there is the parse tree, which is a syntax defined tree. The performance of the tree is basically of the order `O(n)`[23], where `n` is the number of declarations, but since we usually want to branch out and have different options this will very rarely be the case. A valid lower bound for any protocol is impossible to estimate as it is very dependent on how the protocol is written; but a protocol may have constant time for dissecting. Since the intermediate tree is only traversed once for each dissection and that traversing is needed and cannot be optimized in any way since the bottom will always be reached for any way through the tree.

The other data structure for multiple objects is using arrays. These arrays may grow fast depending on how well the user defines the protocol with structures. Each structure and protocol has a list of all their fields so that the value from those fields can then be fetched for use in expressions. These arrays will be traversed an even amount of times for every dissected packet, first time for setting the values, and the other time for removing them. Since they reside unsorted in arrays, the worst time for finding the right value will be `O(n*m)`, where `n` is the number of structures and `m` is the number of fields for that particular structure.

[23] This is valid in simple protocols that are basically a list of fields without any kind of options.

9 Discussion, conclusion and future work

9.1 Discussion

There were quite a lot of features that was cut in the development as time neared its end.
By concentrating on the relevant features and then expand the feature list instead of working half way through all features and discarding those that couldn't quite make it in the end, would make the features in the language more robust and working before more flare was added.

There are some awkward things in the language layout which could have been better. The field name could have been converted into the description string and the filter string, or having that as an optional for easiness. The filter string could have been concatenated in lower levels with what is higher level, e.g. the field "`five`" in the protocol "`high`" would have the filter string "`high.five`".

To make protocol definition easier, there could be a tool for converting C header files and structures in them, into the syntax of the dynamic dissector. A "soft compiler" for spotting errors in the protocol definition and marking them, making them easier to correct before loading them into Wireshark.

9.2 Conclusion

It is possible to make a dissector with reasonable performance using a script language for interpreting the captured packets as input. It has been shown that the interpreted protocol dissector for Wireshark has reasonable performance even for larger protocols, where one at least can view offline captures without any noticeable performance drop. There are expectations that the product of this thesis will find itself integrated into the standard Wireshark build so that further maintenance and upgrades would be handled by the community instead of in-house development.

9.3 Future work

There is room for improvements for this interpreted dissector since the vision was greater than the time allowed. Except for earlier mentioned features that were cut, these are the features that were thought of but not intended as features from the beginning.

1. The major improvement point would be the ability to point out which line an error occurs on in the protocol definition file and not just fail and exit which is the current approach.
2. Other improvements would be the ability to define several protocols in a file or even be able to include several other protocol files so that the definition file could be split into several parts and the ability to reload a definition file while Wireshark is running.

3. For performance there would be a greater advantage using a less naive implementation for storing variables and structures than unsorted arrays.
4. For functional purposes it would be great to be able to tag a field for additional properties so that such as length and CRC checks can be performed and perhaps also add support for other hashing and encryption schemes for more all round capabilities.
5. There should be a way to declare static values so that numbers would have explanations and not seem to be magic numbers.

Of these six points the points number 1 and 2 are the two points that probably will be the most useful additions. Number 1 since it is easy to do a typing mistake and to trying to find a typing mistake is usually an exercise in futility. The second point is valuable in that if there is a possibility to separate different parts of protocols and structures, then only specific parts of protocols may be needed to redefine and common structures may be reused in new and old protocols.

10 References

[1] ISO/IEC, Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, Ref no: ISO/IEC 7498-1:1994(E)

[2] GNU Compiler Collection Internals, For GCC version 4.5.0 (pre-release), 2008 – Richard M Stallman, GCC Developer Community

[3] Pearson Addison Wesley, Compilers – Principles, Techniques, & Tools 2nd Ed, 2007 – Aho, Lam, Sethi, Ullman

[4] Prentice-Hall International, Structured Computer Organization 4th Ed, 1999, Andrew S Tanenbaum

[5] Wireshark Documentation v 1.08, README.plugins

[6] http://www.cs.chalmers.se/Cs/Research/Language-technology/BNFC/

[7] The Labelled BNF Grammar Formalism, Forsberg, Markus; Ranta, Aarne, 2005

[8] Lua 5.1 Reference Manual – http://www.lua.org/manual/5.1/ 2009-06-08

[9] Information about Lua in Wireshark, 2005, http://web.archive.org/web/20050114-19960115re_/http://wiki.wireshark.org/Lua

[10] BNF Converter Multilingual Front-End Generation from Labelled BNF Grammars, Pellauer, Michael; Forsberg, Markus; Ranta, Aarne, 2004

11 Abbreviations

TCP – Transport Control Protocol
UDP – User Datagram Protocol
IP – Internet Protocol
IPX – Internetwork Packet Exchange
SPX – Sequenced Packet Exchange
WiFi – Wireless Fidelity
GPL – GNU Public License
GNU – GNU is Not Unix
PPP – Point-to-Point Protocol
HTTP – Hyper Text Transport protocol
DNS – Domain Name System
BNFC – Backus-Naur-Form Converter
BNF – Backus-Naur-Form
XML – eXtended Markup Language
IEEE – Institute of Electrical and Electronics Engineers, a standardization organization

Appendix A

This is the LBNF-definition of the simple protocol that was created in the beginning to get started and get a feel for what may be needed and the complexity of the task.

```
-- Simple protocol
Protocol. Protocol      ::=   "protocol" String
                              "short" String
                              "abbrev" String
                              "{"   ListTypes   "}"   ;

NilTypes. ListTypes     ::=                                ;
EBuf. ListTypes         ::= EndBuf ";"                     ;
Cses. ListTypes         ::= Cases ListTypes                ;
Types. ListTypes        ::= Declaration ";" ListTypes      ;

EndBuf. EndBuf    ::= Type Ident "[]" String    ;
Identifier. Id    ::= Ident                           ;
FixedArray. Id    ::= Ident "[" Integer "]"     ;
DynArray. Id      ::= Ident "[" Ident "]"       ;

VTypes. Declaration ::= Type Id String String Base Ident    ;
BTypes. Declaration ::= Type Id String String Base          ;
PTypes. Declaration ::= Pad Integer                         ;

VDesc. Cases ::= "switch" Ident "{" [ValueCase] "}"   ;
BDesc. Cases ::= "unmask" Ident "{" [BitCase]   "}"   ;

terminator ValueCase "" ;
terminator BitCase   "" ;
VCase. ValueCase  ::= "case" Integer ":" String ";"              ;
BCase. BitCase    ::= "case" Integer ":" String "," String ";"    ;

Hex.Base ::= "HEX" ;
Dec.Base ::= "DEC" ;
Oct.Base ::= "OCT" ;
Bin.Base ::= "BIN" ;

-- Unsigned
UByte. Type       ::= "uint_8"      ;
UShort. Type      ::= "uint_16"     ;
ULong. Type       ::= "uint_32"     ;
UUltralong. Type  ::= "uint_64"     ;

Pad. Pad          ::= "padding"     ;
```

```
comment "//"             ;
comment "/*" "*/" ;
```

Appendix B

```
entrypoints Definitions;

NilDefinition. Definitions    ::=                              ;
Strctre. Definitions          ::= Structure Definitions        ;
Prtcl. Definitions            ::= Protocol Definitions         ;

Structure. Structure    ::= "structure" Ident "{" StructProperties"}"
        "{" ListTypes "}"     ;
Protocol. Protocol            ::= "protocol" "{" ProtProperties "}"
        "{" ListTypes "}"     ;

ProtNames. ProtProperties ::= Properties ProtProperties1                ;
ProtSubDiss. ProtProperties1 ::= "sub_dissector" "=" String Integer";"  ;

StrNames. StructProperties ::= Properties;

-- This will possibly be expanded later, now used for header length, total
-- length, accept fragmentation, find conversation and other uses.
LongName. Properties ::= "name" "=" String ";" Properties1        ;
ShortName. Properties1 ::= "short" "=" String ";" Properties2     ;
AbbrevName. Properties2 ::= "abbrev" "=" String ";"               ;

------------------------ Generators ------------------------
NilTypes. ListTypes          ::=                                  ;
Types. ListTypes             ::= Declaration ";" ListTypes        ;
Structures. ListTypes        ::= ValueDep ListTypes               ;
Disjunct. ListTypes          ::= IfStatement ListTypes            ;

-- If-statements
IfStatement. IfStatement ::=
        "if" "(" BoolExpr ")"
        "{" ListTypes "}"
        ElseStatement
;
NilDis. ElseStatement   ::=                                ;
ElseIf. ElseStatement   ::= "else" IfStatement             ;
Else. ElseStatement     ::= "else" "{" ListTypes "}"       ;


-- Boolean expressions and operators
BoolNot. BoolExpr ::= "not" BoolExpr1                      ;
BoolOr. BoolExpr1 ::= BoolExpr1 "||" BoolExpr2             ;
BoolAnd. BoolExpr2 ::= BoolExpr2 "&&" BoolExpr3            ;
```

```
BoolImpl. BoolExpr3 ::= BoolExpr3 "->" BoolExpr4            ;
BoolBinExpr. BoolExpr4 ::= BinExpr Comparator BinExpr       ;
BoolValue. BoolExpr4 ::= Boolean ;

True. Boolean     ::= "true"  ;
False.Boolean     ::= "false" ;
coercions BoolExpr 4          ;
-- Binary expressions, yields results other than boolean values
BEAdd. BinExpr ::= BinExpr "+" BinExpr1         ;
BESub. BinExpr ::= BinExpr "-" BinExpr1         ;
BEOr. BinExpr  ::= BinExpr "|" BinExpr1         ;
BEAnd. BinExpr ::= BinExpr "&" BinExpr1         ;
BEMul. BinExpr1 ::= BinExpr1 "*" BinExpr2       ;
BEFDiv. BinExpr1 ::= BinExpr1 "/" BinExpr2      ;
BEId. BinExpr2 ::= Ident                        ;
BEConstant. BinExpr2 ::= Constant               ;
coercions BinExpr 2;

-- To compare binary expressions, Comparator produces boolean values
Equal. Comparator             ::= "=="    ;
NotEqual. Comparator          ::= "!="    ;
Greater. Comparator           ::= ">"     ;
Less. Comparator              ::= "<"     ;
GreaterOrEqual. Comparator    ::= ">="    ;
LessOrEqual. Comparator       ::= "<="    ;

-- Identifier, basic and array variants
Identifier. Id                ::= Ident                     ;
IdentArray. Id                ::= Ident "[" BinExpr "]"     ;

-- IncOffset set to true or leave alone when offset should be incremented,
-- set to false otherwise
DeclBasic. Declaration  ::= Length   Type Id String String IncOffset  ;
DeclVarLen. Declaration ::= Constant Type Id String String IncOffset  ;
DeclStruct. Declaration ::= Struct Ident Id     IncOffset             ;
Print. Declaration      ::= "display" String                          ;
terminator Declaration ";"                                            ;

Forced .IncOffset ::=         ;
Choice .IncOffset ::= Boolean ;

------------ Standard type lengths ------------
Byte. Length           ::= "8bit"       ;
Short. Length          ::= "16bit"      ;
Regular. Length        ::= "32bit"      ;
Long. Length           ::= "64bit"      ;
```

```
Char. Type              ::= "char"        ;
Int. Type               ::= "int" Base    ;
Float. Type             ::= "float"       ;

-- Call for an independent structure
Struct. Struct          ::= "structure"   ;

-- Bases --
-- Only Decimal bases may have the need to show signed values
None.Base    ::= "NONE"             ;
Dec.Base     ::= Sign "DEC"         ;
Hex.Base     ::= "HEX"              ;
Oct.Base     ::= "OCT"              ;
D2H.Base     ::= Sign "DEC_TO_HEX"  ;
H2D.Base     ::= Sign "HEX_TO_DEC"  ;


Signed. Sign ::= "signed"      ;
Unsigned. Sign ::= "unsigned"  ;

-- Tokens, identified with regular expressions
-- Will only accept negative decimal integers, not negative hexs or octs.
token Hexadecimal '0' ('x'|'X') (digit | ["abcdef"] | ["ABCDEF"])+   ;
token Octal '0'["01234567"]*                                         ;
token NegativeInteger '-'["1234567"](digit)*                         ;

-- Will only accept negative decimal integers, not negative hexs or octs.
ConstDouble.      Constant ::= Double             ;
ConstChar.        Constant ::= Char               ;
ConstHex.         Constant ::= Hexadecimal        ;
ConstOct.         Constant ::= Octal              ;
ConstInt.         Constant ::= Integer            ;
ConstNegInt.      Constant ::= NegativeInteger    ;

-- By all bit-wise values, for flags and stuff, will generate a nice looking
-- thing for flags and bitmasks.
BitValue. ValueDep ::= "flag_statement" Ident "{" [BitValueCase] "}"    ;
terminator BitValueCase ";"                                             ;

-- Cases for the above ValueDeps
BitDefault. BitValueCase ::= "default" ":"      String String ;
BitCase. BitValueCase    ::= "case" BinExpr ":" String String ;

----- comments -----
comment "/*" "*/" ;
comment "//"          ;
```

Appendix C

Setting up the environment

To develop for Wireshark in Microsoft Windows environment, the following programs have to be installed and choices have to be made.

6. Visual Studio with SDK, as of late 2008 the standard build version was 2008.
7. Cygwin has to be installed with Archive/unzip, Devel/bison, Devel/flex, Interpreters/perl, Utils/patch and Web/wget.
8. Python may be installed separately or through cygwin.
9. Subversion may be installed so that the latest source version always is available. This is recommended but the source is readily available at the Wireshark home site for download.
10. To build, the batch-file vcvars32.bat has to be executed to setup additional environment variables.

To build the generated C-Files for BNFC the following are necessary:

11. Cygwin with the options Devel/bison, Devel/flex, Devel/gcc-core, Devel/cmake.
12. The bnfc executable has to be downloaded; there is a possibility to build it from source, but it is not necessary when developing on Windows.

Configure win32-setup.sh so that eventual http proxies are used. This is necessary for the build process to download some external dependencies to be included in the build.

Appendix D

This information was copied from the generated documentation from BNFC. There have been some alterations to content and formatting so that it reflects the current implementation.

The Generated Language

BNF-converter
June 7, 2008
This document was automatically generated by the BNF-Converter. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure

Identifiers

Identifiers `<Ident>` are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ ’, reserved words excluded.

Literals

String literals `<String>` have the form `"x"`, where `x` is any sequence of any characters except `"` unless preceded by `\`.
Integer literals `<Int>` are nonempty sequences of digits.
Double-precision float literals `<Double>` have the structure indicated by the regular expression `<digit> + ‘.’<digit> + (‘e’‘-’?<digit>+)?` i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.
Character literals `<Char>` have the form `’c’`, where `c` is any single character.
Hexadecimal literals are recognized by the regular expression `‘0’(‘x’ | ‘X’)(<digit> | [“abcdef”] | [“ABCDEF”])+`
Octal literals are recognized by the regular expression `‘0’[“01234567”]?`
NegativeInteger literals are recognized by the regular expression `‘-’[“1234567”]<digit>*`

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions. The reserved words used in gver4 are the following:

`DEC`	`DEC_TO_HEX`	`HEX`
`HEX_TO_DEC`	`NONE`	`OCT`
`Abbrev`	`char`	`display`
`else`	`false`	`float`
`if`	`int`	`name`
`not`	`protocol`	`short`
`signed`	`structure`	`sub_dissector`
`true`	`unsigned`	

The symbols used in gver4 are the following:

{ } =

`;`	`(`	`)`
`||`	`&&`	`->`
`+`	`-`	`|`
`&`	`*`	`/`
`==`	`!=`	`>`
`<`	`>=`	`<=`
`[`	`]`	`8bit`
`16bit`	`32bit`	`64bit`
`:`		

Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of gver4

Non-terminals are enclosed between < and >. The symbols ::= (production), | (union) and _ (empty rule) belong to the BNF notation. All other symbols are terminals.

```
<Definitions> ::= _
              | <Structure> <Definitions>
              | <Protocol> <Definitions>

<Structure>   ::= structure <Ident> { <StructProperties> } { <ListTypes> }

<Protocol>    ::= protocol { <ProtProperties> } { <ListTypes> }

<ProtProperties>      ::= <Properties> <ProtProperties1>
```

```
<ProtProperties1>     ::= sub_dissector = <String> <Integer> ;

<StructProperties> ::= <Properties>

<Properties>  ::= name  = <String> ; <Properties1>
<Properties1> ::= short  = <String> ; <Properties2>
<Properties2> ::= abbrev  = <String> ;

<ListTypes>   ::= _
              | <Declaration> ; <ListTypes>
              | <ValueDep> <ListTypes>
              | <IfStatement> <ListTypes>

<IfStatement>         ::= if ( <BoolExpr> ) { <ListTypes> } <ElseStatement>
<ElseStatement>       ::= _
                      | else <IfStatement>
                      | else { <ListTypes> }

<BoolExpr>    ::= not  <BoolExpr1>
              | <BoolExpr1>
<BoolExpr1>   ::= <BoolExpr1> || <BoolExpr2>
              | <BoolExpr2>
<BoolExpr2>   ::= <BoolExpr2> && <BoolExpr3>
              | <BoolExpr3>
<BoolExpr3>   ::= <BoolExpr3> -> <BoolExpr4>
              | <BoolExpr4>
<BoolExpr4>   ::= <BinExpr> <Comparator> <BinExpr>
              | <Boolean>
              | ( <BoolExpr> )
<Boolean>     ::= true
              | false

<BinExpr>     ::= <BinExpr> + <BinExpr1>
              | <BinExpr> - <BinExpr1>
              | <BinExpr> | <BinExpr1>
              | <BinExpr> & <BinExpr1>
              | <BinExpr1>
```

```
BinExpr1>      ::= <BinExpr1> * <BinExpr2>
               | <BinExpr1> / <BinExpr2>
               | <BinExpr2>
<BinExpr2>     ::= <Ident>
               | <Constant>
               | ( <BinExpr> )

<Comparator> ::= ==
               | !=
               | >
               | <
               | >=
               | <=

<Id>    ::= <Ident>
        | <Ident> [ <BinExpr> ]

<Declaration> ::= <Length> <Type> <Id> <String> <String> <IncOffset>
               | <Constant> <Type> <Id> <String> <String> <IncOffset>
               | <Struct> <Ident> <Id> <IncOffset>
               | display  <String>

<ListDeclaration>      ::= _
                       | <Declaration> ; <ListDeclaration>
<IncOffset>    ::= _
               | <Boolean>

<Length>       ::= 8bit
               | 16bit
               | 32bit
               | 64bit

<Type>         ::= char
               | int <Base>
               | float

<Struct>       ::= structure
```

```
<Base>        ::= NONE
              | <Sign> DEC
              | HEX
              | OCT
              | <Sign> DEC_TO_HEX
              | <Sign> HEX_TO_DEC

<Sign>        ::= signed
              | unsigned

<Constant>    ::= <Double>
              | <Char>
              | <Hexadecimal>
              | <Octal>
              | <Integer>
              | <NegativeInteger>
```