A Grammar for Describing Protocol Text

Evan Hughes

February, 2005

Abstract

Some streaming network protocols contain text that cannot be described with context free grammars, because they either specify length of subsequent fields numerically, or use runtime-defined field terminators. This document describes a grammar that can parse at least some of these protocols.

1 Introduction

This document describes the network protocol grammar, or NPG. The NPG is an attempt to define a machine readable grammar that can fully describe the message structure of streaming network protocols. It is our intent that the NPG be used to automatically build parsing code for network servers and monitors, much in the style of flex[5] and bison[4].

A machine-readable grammar definition would make the automatic generation of network parsing code possible. A properly tuned code generator would produce code that avoids common coding errors, is relatively fast, and secure against known attacks. The definition could be used to generate code for any language that a generator had been written for, effectively meaning that the grammar definition could provide an implementation of a parser in any language.

Using generated parsing routines, we would have an assurance that the generated parser would parse *every* valid protocol sequence that adhered to the grammar (modulo bugs in the code generator). With hand written code, we do not have that assurance; we can only throw tests cases at the parser and hope that those cases cover all possible inputs. Because the definition of the grammar would define every legal conversation, we term it the *canonical definition* of the protocol. Ideally, the canonical definition would only have to be written once, after which it could be transformed into parsing code to work in any language.

The NPG can also be used to quickly extend network analysers. We have worked on a network analysis library called `qcap`[6], that provides reconstruction of network streams. We wish to use the NPG to quickly extend `qcap` to handle new protocols, without writing new code.

1.1 NPG? Why Not CFG?

Some network protocols cannot be described by context free grammars, because their grammar is often dependent on context. Consider the `Content-Length` header of HTTP[2]: it defines the length of the following request or response body. In order for the grammar to describe the protocol, it must provide a mechanism to use the value of the `Content-Length` field as part of the grammar definition. Also, there is the boundary field of multi-part MIME[3] messages, which defines a terminator that signals the end of a semantic entity. Both of those protocol constructs can be defined in a context-free grammar, but require vast enumerations of all possible values to be able to properly parse the protocol. We wish to explicitly state the context-dependent portions of the grammar in the grammar definition, in order to be able to build more expressive parsers, without resorting to incurring Turing-complete complexity.

Protocols often carry encoded portions as well. Encodings range in complexity from simple character encoding (a mapping of one character set to another), to non-parameterised encodings (that are a function of data flowing across the protocol), to parameterised encodings (that are a function of data flowing across the protocol, and some other value, such as a shared or asymmetric secret).

The NPG handles moderately context dependent protocol texts, and encoded portions.

1.2 Previous Work

The concept of machine-parsable grammars to describe network protocols is not new. Other authors have proposed various schemes for handling grammars that are slightly context-dependent. However, the work that we have encountered has focused on parsing bit-oriented streams within packets. We have not encountered any other authors who proposed automated parsing of streaming protocol text.

Jayaram and Cytron[7] propose using context-free grammars to describe network packets for high-speed filtering. Their solution to dealing with context-dependent portions of packets is to enumerate all possible values of length fields and terminators. Lambright and Debray[9] propose a language similar to yacc[8]. The language includes a limited set of actions that are evaluated when a rule matches. On a similar vein is the Austin Protocol Compiler by McGuire and Gouda[11], which provides a slightly more complex mechanism for handling actions.

The NPG is based on Packet Types[10], proposed by Chandra and McCann in 1999. We chose to focus on this work because it uses a syntax very similar to Backus-Naur Format, which we expect most protocol designers to understand. Packet Types adds the notion of attributes to elements in the grammar that control how each element will match. Attributes can be manipulated through small expressions attached to each rule, that are evaluated when the rule matches.

2 Language

There are many ways to define the message format of a streaming protocol, from a prose description, to an ad hoc description (as used in the definition for the Simple Mail Transfer Protocol[12]), to XML (as used in the BEEP core definition[13]). In most cases, the protocol text is described with something approximating a context free grammar.

Some authors have gone so far as to include code in the protocol definition[9, 11]. Our grammar avoids using code an an attempt to minimize the complexity of the definition. We solely wish to provide a mechanism for parsing context dependent protocols, not building a new programming language.

We are interested in the family of protocols defined by the Internet Engineering Task Force (or IETF), in documents called "Requests for Comment". RFCs, as they are commonly termed, often[1] use a dialect of context free grammar called Augmented Backus-Naur Form[1] (ABNF).

We have chosen ABNF as the basis of the network protocol grammar because it is often used to describe protocols in IETF RFCs. By basing the network protocol grammar on ABNF, we intend that the NPG be as compatible as possible with the protocol definitions defined in existent RFCs. Our hope is that creating definitions of existing protocols should be as simple as copying the RFC text into the NPG definition file, and adding a few small items of syntax.

When speaking of the parts of the NPG, we use much of the same terminology as that used for ABNF. We call an ABNF definition of a grammar a **definition**. According to the ABNF specification, ABNF consists of rules, terminals, and symbols. Each rule has a body (also called a production), which consist of **terminals** or **symbols**. A terminal is one or more values that map to characters. A symbol is a named reference to another rule. We assume that the reader is familiar with ABNF.

Note that our use of the word "definition" to refer to a grammar written in ABNF or NPG is not standard. We also use the nonstandard word **term** to refer to an item in a definition that may be either a symbol or a terminal.

The remainder of this section describes the additions to ABNF that we are proposing to create the network protocol grammar.

2.1 Clauses

The first change we make to ABNF is to add clauses to rules. A clause is text wrapped in braces. ABNF's normal rule format remains supported; however, we add an alternative "rule" clause, that allows a normal ABNF rule definition to be wrapped in braces.

The rule clause is syntactic sugar. It is unnecessary, but aesthetically pleasing.

```
rule = symbol TERMINAL symbol ; Standard ABNF rule

rule = {symbol TERMINAL symbol} ; Use of the rule clause
```

Figure 1: Two formats for the rule clause in NPG. The first is the standard ABNF format. The second, expressing an identical grammar, is introduced in NPG.

2.2 "where" Clause

The where clause may follow a rule clause. It contains attributes allowing context-specific changes to be made to the grammar.

```
Length = DIGIT
Text = ALPHA

rule = {
     Length Text
} where {
     Text#repetition_min = count;
     Text#repetition_max = count;
}
```

Figure 2: Example of attribute assignment in a where clause. The following grammar will match:

Rule-specific attributes are defined in the where clause. NPG defines five attributes: `repetition_min`, `repetition_max`, `repetition`, `text`, and `encoding`. These attributes potentially change the cardinality of a term, the text of terminals, or the encoding used to transform text.

This concept is based on that originally seen in Packet Types[10]. We have added the `encoding` attribute, and are applying the concept to stream-based protocols.

2.2.1 repetition_min, repetition_max, and repetition

`repetition_min` sets the minimum cardinality of its term. It is of integer type. In order for the term to match, it must occur at least `repetition_min` times. `repetition_max` sets the minimum cardinality of its term, and is also an integer. In order for the associated term to match, it must occur no more than `repetition_max` times. `repetition` is a short hand. Setting `repetition` to n is the same as setting both `repetition_min` and `repetition_max` to n.

2.2.2 text

text changes the value of a terminal. It is a string type.

```
Before = ALPHA
Text = ALPHA

rule = {
     Before "!" Text After
} where {
     After#text = Before;
}
```

Figure 3: Example of text attribute. The following grammar will match: "*a!bba*", "*ttt!attattt*".

2.2.3 encoding

encoding sets an encoding function to operate on the *contents* of a term. Encoding names are strings. Encodings are built up in a stack, and their subelements match against the unencoded terms.

```
UPPERCASE = 1*%x41-5a ; Match one or more uppercase characters.

rule = {
     "a" UPPERCASE "b"
} where {
     UPPERCASE#encoding = cap;
}
```

Figure 4: Example of encoding attribute. The `cap` function translates text into their capitol equivalents. The following grammar will match: "*ab*", "*abcdefb*".

Encodings may be parameterised, passing either static text into the decoding function, or text from the protocol stream.

3 Discussion and Further Work

3.1 How Attributes are Evaluated

When the parser enters a rule, it pushes the set of all attributes onto a stack. As terms are accepted in the rule, those that are on the right hand side of an attribute are set as attribute values immediately. If the rule should fail at any point, the most recent parser state is popped off the stack, and used to overwrite all attributes that were set since the rule began evaluation. If the rule is accepted, the old parser values are popped off the stack and discarded.

Attributes have types. Those defined above are either string or integer types. Whenever an attribute is the target of an assignment, the right hand side of the assignment is run through a conversion. The default conversion for string types is a byte-wise copy. The default conversion for integers is to convert the number as an unsigned base 10 string. Other conversions may be specified by wrapping the value in an infix function call.

```
...
} where {
     term1#repetition = hex2dec(term2);
}
```

Figure 5: Example of using non-default conversions during attribute assignment. In this situation, the text <`term2`> matched is run through the function `hex2dec`, which is linked from an external library of conversion functions.

3.2 Generality of the `text` Attribute

The `text` attribute is a limited method of making changes to a grammar in response to received input. We include it in the grammar as a necessity: MIME borders require that a string be read from the input text and used later as a terminal. Ideally, we would like to use a more generalised mechanism – as we explore other protocols, it seems likely that we will encounter other strange syntactic constructs. However, a generalised mechanism for constructing rule definitions would likely have to be Turing complete, and would make the NPG more complex than we wish.

In the near future, we will use the `text` attribute. If we encounter major protocols that require more complex run-time grammar modifications, we will investigate other mechanisms.

3.3 Libraries and Linking

So far we have not mentioned how NPG definitions will be used. Currently, that is left deliberately hazy, as we are exploring various uses.

At the very least, it is sufficient to say that the NPG definition of a protocol will be loaded, and will be used to generate a parser. The parser will provide an API, allowing applications to read protocol text without having to define the structure of the protocol text in Turing-complete code. It may be necessary for applications to pass functions into the parser to provide protocol-specific decoding.

3.4 A Formal Grammar for NPG

At this time, we are creating a formal ABNF grammar for NPG. When that is complete, we will amend this report to include the ABNF.

4 Acknowledgements

I would like to thank Glenn Wurster of Carleton Computer Security Lab for his contributions to the research on the NPG, and my advisor, Dr. Anil Somiyaji, for proofreading this report.

References

[1] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 4234 (Draft Standard), Oct. 2005.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[3] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046 (Draft Standard), Nov. 1996. Updated by RFCs 2646, 3798.

[4] GNU Project. Bison homepage. `http://www.gnu.org/software/bison/` Accessed: January 30, 2006.

[5] GNU Project. Flex homepage. `http://www.gnu.org/software/flex/` Accessed: January 30, 2006.

[6] E. Hughes and A. Somayaji. Towards network awareness. In *Large Installation System Administration Conference (LISA'05)*, Dec 2005.

[7] M. Jayaram and R. K. Cytron. Efficient demultiplexing of network packets by automatic parsing. Technical Report WUCS-95-21, Washington University, July 1995.

[8] S. Johnson. Yacc: Yet another compiler-compiler. `http://dinosaur.compilertools.net/#yacc` Accessed: February 13, 2006.

[9] H. D. Lambright and S. K. Debray. Apf: A modular language for fast packet classification. URL: http://www.cs.arizona.edu/people/debray/papers/filter.ps.

[10] P. J. McCann and S. Chandra. Packet types: abstract specification of network protocol messages. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[11] T. M. McGuire. The austin protocol compiler reference manual. Technical Report UTCS-TR02-05, University of Texas, 2002.

[12] J. Postel. Simple Mail Transfer Protocol. RFC 821 (Standard), Aug. 1982. Obsoleted by RFC 2821.

[13] M. Rose. The Blocks Extensible Exchange Protocol Core. RFC 3080 (Proposed Standard), Mar. 2001.