An ABNF Extension for code generation

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on November 28, 2009.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info). Please review these documents carefully, as they describe your rights and restrictions with respect to this document.
Abstract

This document describes an ABNF extension for code generation. The extension has two features: extension rule and non-sequence group notations. The extension rules are used to direct the parser generator with things like data types, variable names, forced value for a variable, etc. The non-sequence group feature was proposed as part of RFC 2234 in the past, but dropped due to its ambiguities. The feature is proposed again in this document not as a fundamental building block, but as an add-on. The elements of a non-sequence group are unordered, and are allowed multiple appearance. We attempt to minimize the ambiguities stemmed from repetition of an element by defining specific repetition rules for elements of a non-sequence group and non-sequence group themselves.
Table of Contents

1. Introduction ................................................. 4
2. Non-sequence Group  {Rule1 Rule2} .......................... 4
   2.1. Element’s Repetition in a non-sequence group ........ 4
   2.2. Sequence, Grouping, and Alternation in a non-sequence group . 4
   2.3. Repetition of non-sequence group ...................... 4
3. Extension Rule ;--XRule ..................................... 4
4. Restrictions on ABNF for code generation .................. 5
   4.1. Limit on value length .................................. 5
   4.2. No option for alternation elements .................... 5
   4.3. Rule Termination ....................................... 5
   4.4. Maximum Repetition ..................................... 5
   4.5. Character Range ....................................... 5
5. ABNF Definition of Non-sequence Group ..................... 5
6. Security Considerations ..................................... 5
7. References .................................................. 6
   7.1. Normative References .................................. 6
   7.2. Informative References ................................ 6
Appendix A. Acknowledgements ................................. 6
Appendix B. Extension Rules .................................... 6
1. Introduction

ABNF, Augmented BNF (Backus-Naur Form) has been widely used for Internet specifications. When ABNF was documented into RFC 2234 in 1997, non-sequence group was dropped due to its ambiguities. 12 years later, we found non-sequence group still useful for code generation from it into an implementation programming language, like C. This document attempts to reduce ambiguities that stem from non-sequence group, by specifying specific repetition rules.

2. Non-sequence Group: \{Rule1 Rule2\}

Elements enclosed in braces are unordered. Its contents may occur in any order.

Hence:

\{foo bar\} baz

would match (foo bar baz) and (bar foo baz).

2.1. Element’s Repetition in a non-sequence group

Repetition in a Set has a special meaning; it is expanded before Set rule applied.

Hence, when set = \{3A X\}
it is rolled out to \{A A A X\} and
(A A A X) / (A A X A) / (A X A A) / (X A A A)

2.2. Sequence, Grouping, and Alternation in a non-sequence group

An element in a non-sequence group gets "flattened out" before Set rule applied.

When g1 = \{(3A) X\}, it is same as \{3A X\} that is expanded to \{A A A X\}, and finally,
it becomes (A A A X) / (A A X A) / (A X A A) / (X A A A)

Likewise, \{(A B C) X\} equals to \{A B C X\}

2.3. Repetition of non-sequence group

A non-sequence group with a repetition is expanded before non-sequence group rule applied.

When g1 = \{A B\},
2g1 = (g1 g1) = \{A B\} \{B A\} = (A B) / (B A) \{(B A) / (A B)\}
= (A B B A) / (A B A B) / (B A B A) / (B A A B)
3. Extension Rule ;--XRule

An Extension Rule is specified using ;--X to give data type and variable name to generated parser code. Appendix B lists predefined Extension Rules.

Each Rule has the reference index 0, and indexes on its right hand side start from 1. Alternation and repeater do not have an index value.

4. Restrictions on ABNF for code generation

4.1. Limit on value length
In case of bin-val, dec-val and hex-val, the maximum length of concatenation is 80

4.2. No option for alternation elements
An element in alternation should not have options. so the rule "A = B / C [D] / E" should change to:
"A = B / X/ E where X = C [D]"

4.3. Rule Termination
A rule should be terminated by ";" or ;--X" for an extension rule
The string from ";" to CRLF is regarded as a comment
The string from ;--X" to CRLF is regarded as an extension rule

4.4. Maximum Repetition
The maximum value of unlimited repetitions is 9999.
So *(SP|HT) can occur 0 - 9999 times.

4.5. Character Range
A character value should be ranged 0 - 255
ex) 0x01-0xfc : OK
0x01-0xffcc : Not OK

5. ABNF Definition of Non-sequence Group

non-sequence group = "{" repetition *(1*c-wsp repetition ) "}"
; repetitions in any order

6. Security Considerations

Security is believed to be irrelevant to this document.
7. References

7.1. Normative References


7.2. Informative References

Appendix A. Acknowledgements

Appendix B. Extension Rules

B.1. XPDU
XPDU indicates that the rule performs encoding or decoding
Synopsis:
    ;--XPDU
Example:
    StartLine = *CRLF ( StatusLine / ReqLine)
             ;--XPDU

B.2. XCUT
XCUT is used to cut unnecessary parts when the struct type name is code-generated.
Synopsis:
    ;--XCUT [{<cut id>} *(, {<cut id>})]
where <cut id> is an integer
    It does not have any relation with decoding rules.
    If <cut id> is absent, then the rule does not get generated
    in the user header file
Example:
    ;--XCUT 1, 4

B.3. XTYPE
XTYPE is used to specify the data type for the rule.
If XTYPE is absent, the struct type is used by default.
Synopsis:
    ;--XTYPE [{<type id>}={<type name>} *(, {<type id>} = {<type name>})]
where <type id> is a number
and <typename> is one of structl, struct, uint, ushort, char*, char, uchar, enum, octet(num), octet, char(num), objId, bit, float, boolean, null or char*esc.

The following typenames are used only (TypeId=0) TYPE:
structl, struct, octet(num), octet, objid, bit, enum
whereas the following is used only (TypeId>0) TYPE:
null

Each typename is described below.

structl
This type name is used to generate linked struct types.
Example:

    UriPrms = 1*( ";" UriPrms)
    ;--XVAR 2
    ;--XTYPE 0=struct.

From the above ABNF definition, something like following can be generated.

typedef struct UriPrms_ {
    struct UriPrms_*        next;
    UriPrm                  value;
} *UriPrms;

struct
This type name is used to generate struct types.
char*, uint, ushort, char, uchar, enum, char(number), float, and boolean. Their namesake types in C is type-defined by the rule names.

bit
This type name is used to define a string of zero or more bits, and identical to ASN.1 BIT STRING.

Example:

    NotifyCompletionReason = (TimeOutToken
        / InterruptByEventToken
        / InterruptByNewSignalsDescrToken
        / OtherReasonToken)
    ;--XVAR 2=onTimeOut, 3=onInterruptByEvent,
    ;--XVAR 4=onInterruptByNewSignalDescr, 5=otherReason

From the above ABNF definition, something like following can be generated.
typedef unsigned char NotifyCompletion;
#define onTimeOut 0x80
#define onInterruptByEvent 0x40
#define onInterruptByNewSignalDescr 0x20
#define otherReason 0x10

null
The type name is mapped to a null type and cannot be assigned to 0’th element.
Example:
TransactionReply = ReplyToken EQUAL TransactionID LBRKT
[ ImmAckRequiredToken COMMA]
TransactionResult RBRKT
;--XCUT 1,2,4,7,9
;--XVAR 3=transactionId, 6=immAckRequired,
8=transactionResult
;--XTYPE 6=null

From the above ABNF definition, something like following can be generated.

typedef Nulltype char;
typedef struct TransactionReply {
unsigned char bit_mask;
#define immAckRequired_present 0x80
TransactionId transactionId;
Nulltype immAckRequired;
TransactionResult transactionResult;
} TransactionReply;

tonket(num)
This type name is used to define a string whose length is the given num. It can be generated into C something like:

typedef struct AAA {
unsigned short  length;
unsigned char value[num];
} AAA;

octet
This type name is used to define an 8-bit string. It can be generated into C something like:

typedef struct AAA {
unsigned short  length;
unsigned char* value;
}
This type name is used when a hex-digit value is mapped to an ASCII character. For instance, "%61lice" is mapped to "alice" using the following example definition.

Example:
```
UserInfo = 1*(%x21-3F / %x41-FF) "@"
--Xcut 4
--Xtype 0=char*esc
```

The above example definition can be generated to:
```
typedef xc8* UserInfo;
```

**B.4. XVAR**

This extension rule is used to assign variable names and values.

Synopsis:
```
;--Xvar nameId=nameVal *(, nameId=nameVal)
```

**B.5. XCHOICE, XCHOICE_S**

This extension rule is used to assign choice names and values.

**XCHOICE** for full choice names

**XCHOICE_S** for short choice names

Synopsis:
```
;--Xchoice nameId=nameVal *(, nameId=nameVal)
;--Xchoice_s 2,3,4
```

**B.6. XBITMASK, XBITMASK_S**

This extension rule is used to assign bitmask names and values.

Synopsis:
```
;--XBITMASK nameId=nameVal *(, nameId=nameVal)
;--XBITMASK_S 2,3,4
```

short bitmask selection method
full bitmask name : typename_variablename_present
short bitmask name : variablename_present

**B.7. XTDEF**

This is used to do a typedef in C.
It should not be used with the structt1 type name

Synopsis:
```
;--Xtdef digit
```

Example:
```
AddRequest = AddToken EQUAL AmmRequest
;--Xcut 1,2
;--Xtdef 3
```

Then, the following code is generated from the above:
```
typedef AmmRequest AddRequest
```
B.8.   XORDER

Synopsis:

;--XORDER digit 1*(, digit)

Example:

CommandRequest = ["O-" ] ["W-" ] Command
;--XVAR 2=optional, 4=wildcardReturn, 5=command
;--XORDER 5,4,2
A struct sequence is generated as
command, optional, wildcardReturn

B.9.   XNCASE

Used to direct case-insensitivity, when strings compared.

Synopsis:

;--XNCASE ncaseId *(, ncaseId)

B.10.   XLEN

This specifies a length for an element name.

Synopsis:

;--XLEN number

Example:

MessageBody = Token
;--XLEN ContentLength

B.11.   XDUP

Used to specify byte values which the compiler can take.

Synopsis:

;--XDUP <type id>=<hex-digit> *(, <hex-digit>)

Example:

CallId = ( "Call-ID" / "i" ) HCOLON Payload CRLF
;--XCUT 2,3,4,6
;--XNCASE 2,3
;--XTDEF 5
;--XDUP 3=0x20,0x09,0x3a
Then, the following code can be generated:

typedef Payload CallId
typedef struct Payload {
xu16                   length;
xu8*                   value;
} Payload;

B.12.   XALT

This extension rule is used to assign a fall-back option

Synopsis:

;--XALT <ext id>

where <ext id> is a number

Example:
Host = Ipv4 / Ipv6 / HostName
    ;--XALT 3
HostName = 1*(Alphanumeric / "-"/ ".")
    ;--XTYPE 0=char

B.13. XTOK
This extension rule is used only with the struct type name and does not appear in the generated user struct code. It has relation with decoding rule.
Synopsis:
    ;--XTOK <tok id>
    where <tok id> is a number

B.14. XSTRL
This is used to appoint a char as delimiter if there exists no delimiter when one is needed.
Synopsis:
    ;--XSTRL <ext id> = <byte> *(, <byte>)
    where <ext id> is a number and <byte> is a hex-digit.
Example:
ExtHdrList = 1*(ExtHdr)
    ;--XTYPE 0=struct1
    ;--XSTRL 2=0x21,0x23-27,0x2a-2b,0x2d-2e,0x30-39,0x41-5a,0x5e-7a,0x7c,0x7e
    ;In this case, it specifies all the character suitable for ExtHdr
    = Token
    ;
    ExtHdr = Token HCOLON 0*1Payload CRLF
    ;--XCUT 2,4
    ;--XVAR 1=mHdrName, 3=mHdrValue

B.15. XNRPT
Extension data are used as a general rule, when pre-conditions are not met. If extension data use XSTRL for its struct1 type, it’s necessary to check if the pre-conditions are met by setting XNRPT to 1. The following example shows that once ExtHdr is met, there is a chance for it to continue to be met. To avoid this situation, it’s forced to check ExtHdr only after AcceptList and ViaList are checked.
Synopsis:
    ;--XNRPT <ext id>
    where <ext id> is a number
Example:
MsgHdrList = *(0*1AcceptList
    ViaList
    0*1ExtHdrList
}) CRLF
ExtHdrList = 1*(ExtHdr)
;--XTYPE 0=structl
;--XSTRL 2=0x41-5A,0x61-7A,0x30-39,0x2d,0x2e,0x21
 ,0x25, 0x2a,0x5f,0x2b,0x60,0x27,0x7e
 ;--XNRPT 1
ExtHdr = Token HCOLON Payload CRLF
;--XCUT 2,4
;--XVAR 1=mHdrName, 3=mHdrValue

B.16. XFENC
This extension rule is used when a variable needs to be overridden with a given value.
Synopsis:
;--XFENC <exit id> = <value>
where <exit id> and <value> are a number.
Example:
HCOLON = *( SP / HTAB ) ":" 0*1LWS
 ;--XCUT 0
 ;--XFENC 5=0x20
LWS = [*WSP CRLFLWS] 1*WSP
 ;--XCUT 0

B.17. XNCMP
This extension rule is used when the length in PDU rule and the size of the input stream are not to be compared.
Synopsis:
;--XNCMP
Example:
SIPMessage = StartLine MsgHdrList
 ;--XPDU
StartLine = *CRLF (StatusLine / ReqLine)
 ;--XNLCMP
 ;--XCUT 1
 ;--XALT 4
 ;--XCHOICE 3=SL_mStatus_chosen, 4=SL_mReq_chosen

Authors’ Addresses

Jong Sung Kim
VineGen Inc.
(461-714)5303 Dong-seoul college Business Incubator center
423 Bockjung-dong Sujung-gu Sungnam-si Gyunggy-do S. Korea
Phone: +82 031-756-5307
EMail: jskim@vinegen.com

Munjo Yu
VineGen Inc.
310 Ambermore Place
Cary, NC 27519
Phone: +1 919-523-5146
EMail: Munjo.Yu@gmail.com