NFQL Front-End Parser

by

Durim Morina

Bachelor Thesis in Computer Science

Prof. Dr. Jürgen Schönwälder
Name and title of the supervisor

Date of Submission: May 12, 2013

Jacobs University — School of Engineering and Science

Abstract

The `nfql` (Network Flow Query Language) tool can process flow records, apply filters on them, group them, invoke Allen interval algebra rules and merge the group records.
This thesis focuses on parsing of the language which is used to query the flow records. Writing manually a query in JSON as the `nfql` expects it is really difficult and time consuming. A query specifies the parameters that we want to apply on the stages of `nfql` engine. The parser accepts a query file as input and it is supposed to parse it and if the file is accepted by our grammar then it will output JSON format of the query. The source code of this project can be found in the github repository[1].

[1]Source Code. [Online]. Available: https://github.com/dmorina/nfql-parser [Accessed: May 12, 2013]

Contents

List of Figures

1 Introduction

The Internet is definitely an open and free place, everyone can send and receive data to/from anyone, with some exceptions of course where some security policies are applied. As it is like this many would like to do research on it or monitor it. Researchers, network administrators and operators are interested mainly in two things on "Who is using the Internet?" and "For what purposes?"
There are many reasons and perspectives on why the Internet should be monitored. A researcher would wish to know e.g. top 10 visited websites on a campus or city. A network admin would try to keep the network as secure as possible, find the origin of the attacks and the targets as well. A developer would like to debug his/her own implementations of protocols and so on.
Storing traffic data that is being sent/reveived to/by hosts is an issue because the amount of data flowing on the Internet is huge and also in many countries it might be illegal to do so. An idea is to store only the IP header and not the IP packet itself. It would give us enough information on a stream.
There are protocols like Cisco NetFlow[1] and IPFIX[2] which enable us to get the data to do network analysis. NetFlow was developed by Cisco, and then Version 9 of it was chosen as basis for a proposed IETF standard called IP Flow Information Export (IPFIX).
NetFlow is built into Cisco IOS[3] and runs on Cisco Routers and Layer 3 Switches. A router or L3 switch will examine packets and get only some attributes from the IP header, and it keeps track of the flows. A flow is like a "conversation" between two hosts that consist of a server and a client and a service port.
The NetFlow architecture is divided in three parts: exporter, collector and analyzer. Exporter is a router/switch which exports flows, the flow collector collects the flows and saves them, and finally analyzed by an analyzer. A flow is ready for export when it is inactive for a certain time (i.e. no packets received for the flow) or when it is active but lasts longer than the active timer. Also, the flow is ready to export when TCP flags is set (i.e. FIN, RST flag). Please see Figure-1 which shows the NetFlow architecture: After each packet is examined there are certain attributes which are used to identify packets that belong to a flow. There are 5 up to 7 attributes used for this purpose in NetFlow version 5, there have been added more attributes in NetFlow version 9 and IPFIX as well:

1. IP source address
2. IP destination address
3. Source port
4. Destination port
5. Layer 3 protocol type
6. Class of Service

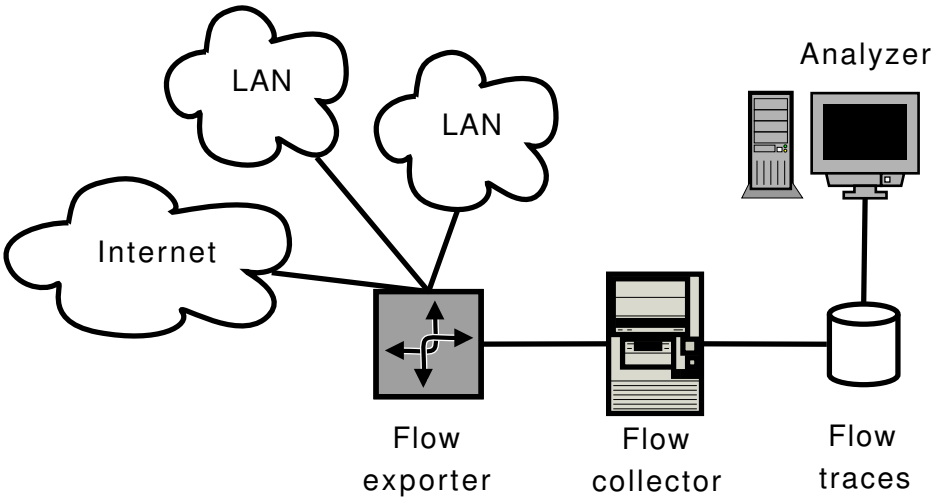

Figure 1: NetFlow architecture[4].

7. Router or switch interface

The flow collector is a device which gathers flow records, it can be a device such as a router or a switch.
The flow analyzer is a tool which is used to analyze flow records. There are many tools which can be used for this purpose, like nfdump[5], Softflowd[6], Cisco NetFlow Collector[1] and many others.
One of the options is to use `nfql`. We designed a parser for the `nfql` which parses the query files and if the parsing is successful it generates the query file in JSON format. This JSON format query file is fed to the `nfql` engine. The reason why we are doing this is that writing a query in JSON format is really difficult and time consuming. And also the person who is writing the query would have to know all the data types of each element that he/she is querying for. Having a simple query language and a parser which parses the query and also generates the JSON format of it makes things much easier.

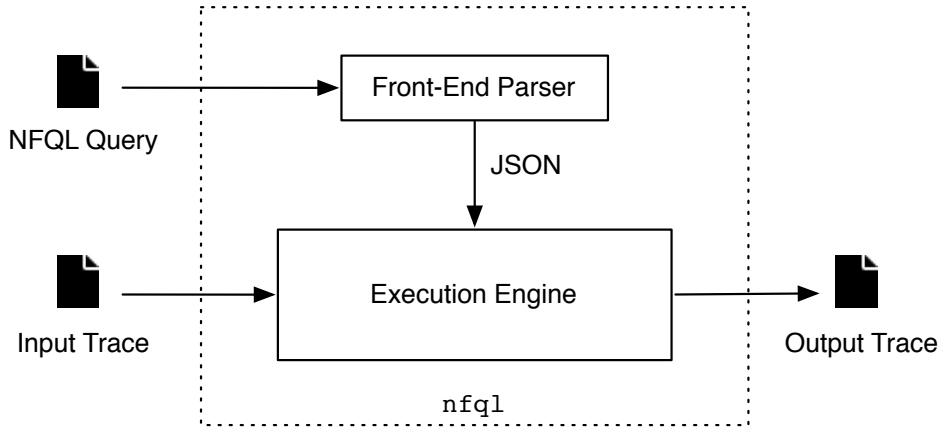

Figure 2: NFQL architecture[7].

2 Background

Flow records need to be queried in order to be able to understand them. NFQL[8] is a query language designed here at Jacobs University which allows to query the flow records just like using SQL for databases. It allows us to process flow records, apply filters, aggregate them into groups, merge them, and invoke Allen interval algebra rules. It is based on NetFlow v5. `nfql` is made of two main parts: the engine and the parser. The engine is the main part it does all the processing of flow records. The parser is supposed to parse the custom language which is used to query flow records and if parsing passes the grammar rules it will output the JSON format query. Please see Figure-2.

Parsing the query manually and converting it to JSON format is rather difficult, for this reason we want to have something that does this for us automatically.

`nfql` is designed as pipeline and each operation on the flow records is separated in stages. There are five stages but we dont necessarily have to use all of them, so some of them can be disabled. The five stages are: the filter, the grouper, the group-filter, the merger and the un-grouper. Please see Figure-3.

The filter processes the flow records, it performs absolute filtering on the input as specified in the query. The flow records that pass the criterion are then forwarded to the grouper which groups the flow-records according to the grouping criteria, just like a GROUP BY SQL clause. The grouper performs aggregation on the flow-records, possible aggregation operations are static, count, product, sum, logical and/or/xor, arithmetic mean, standard deviation, union, median, minimum and maximum.

The group filter is the last processing stage of the branch, it performs absolute

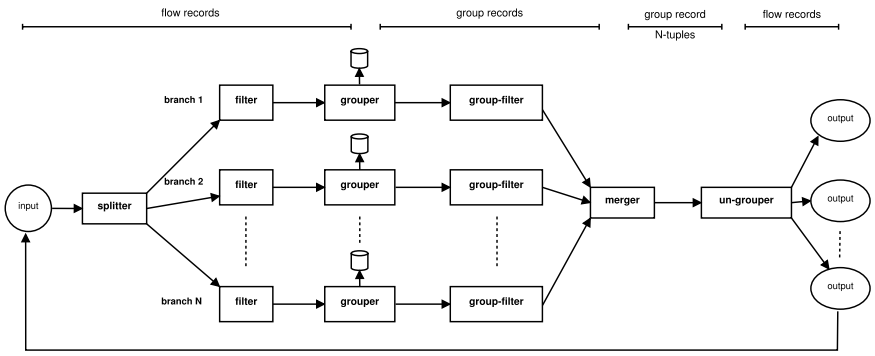

Figure 3: NFQL Pipeline[8].

filtering on the groups. Then the group-records that pass the criterion are forwarded to the merger which performs relative filtering according to some criteria specified in the query and finally the un-grouper, it extracts the group-records into flow-records ordered by their timestamps.

Formal languages are really powerful on describing concisely a language and also at the same time specifying the parser. We will go into details of the NFQL language in the next section but let's see first how the language looks like. Let's say we would like to apply a filter on destination port 443, for the TCP protocol. Then we would write our query like this:

```
branch A {
  filter https{
      tcpDestinationPort = 443
  }
}
```

we would expect some output like this from the parser:

```
"filter": {
        "dnf-expr": [
          {
            "clause": [
              {
                "term": {
                  "delta": 0,
                  "op": "RULE_EQ",
                  "offset": {
                    "datatype": "RULE_S1_16",
                    "name": "tcpDestinationPort",
```

```
                        "value": 443
                    }
                }
            }
        ]
    }
]
}
```

The query above defines a branch which includes a filter with a single rule. This filter will be applied to all records in the filtering stage and only those flow records that have tcpDestinationPort equal to 443 will pass.

Terminology

A language *L* is a set of strings from an alphabet. An alphabet is a set of symbols which are unique. An example of an alphabet is the ASCII character set or the letters in English language.

We can apply rules on an alphabet and the set of rules is called a grammar. Let *G* be a grammar then the language *L* defined by those rules will be *L(G)*.

Let *G* be a grammar and *s* be a string, then we can apply one of the two operations: check if *G* recognizes *s* or use *G* to parse *s*. Recognition is a process of checking if *s* is an element of *L(G)*, if it is then *s* is described by the grammar else *s* is part of another language. Parsing means to decompose *s* by using *G*, the process depends on the nature of the grammar.

The output can be one of the two: the grammar can accept the input *s* or it can reject it. If the input *s* is accepted we also say that it matches the grammar meaning that it is a member of the grammars language.

Backus-Naur Form[9] is one of the two main notation techniques of Context Free Grammars and Augmented BNF[9] is a special form of BNF, consisting of its own syntax and rules.

3 The NFQL Grammar

3.1 Problem Analysis

We wish to build a parser which reads a query file and parses it and if our defined grammar accepts it, it goes to the next step and converts it to JSON format. We also wish to define the parser in such a way that it is concise, easy to program and it should provide meaningful error messages on invalid inputs. The size of the query which is given to the parser is rather small so the computation time is not an issue in this case, if properly programmed it will be fast.

3.2 A Correct Parser

A parser is said to be "correct" if it correctly translates all words that belong to the language defined by the grammar. And as it is written by humans it is likely to have bugs. But fortunately we have context free grammars which allow us to have explicit specifications of the grammar and that will help us to lower the number of bugs.

3.3 Review of ABNF

As we said before a grammar is a set of rules and we chose to define those rules using ABNF, but first lets look at the basics of ABNF[9].

Every rule has a name followed by an assignment operator followed by an expression which consists of terminals or operators.

Constants are terminal values, terminals are just numbers. Numeric constants allow us to have ranges or alternatives.
Rules are defined in ASCII character set and some examples are shown below:

```
rule1 = "abc"; this matches all of the following
              ; "abc", "Abc", "ABc", "ABC", "aBc", "aBC",
              ; "abC", and "AbC"
```

If we wanted to be specific we would write it like this:

```
rule1 = %x61.62.63; which matches exactly
                  ; the string "abc"
```

Incremental alternatives allow us to add options to the set of options defined before, and it can be used like this,

```
rule1 = option1 / option2
rule1 =/ option3
```

At the end rule1 will accept *option1* or *option2* or *option3*.

We can group elements using parenthesis. Variable repetition is defined using "*" symbol.

There are other rules of ABNF which we will not cover here, but they are defined on RFC 5234 [9]. ABNF has its own core definitions which are listed below:

```
ALPHA     =  %x41-5A / %x61-7A
                    ; A-Z / a-z
BIT       =  "0" / "1"
CHAR      =  %x01-7F
                    ; any 7-bit US-ASCII character, excluding NUL
CR        =  %x0D
                    ; carriage return
CRLF      =  CR LF
                    ; Internet standard newline
CTL       =  %x00-1F / %x7F
                    ; controls
DIGIT     =  %x30-39
                    ; 0-9
DQUOTE    =  %x22
                    ; " (Double Quote)
HEXDIG    =  DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
HTAB      =  %x09
                    ; horizontal tab
LF        =  %x0A
                    ; linefeed
LWSP      =  *(WSP / CRLF WSP)
                    ; linear white space (past newline)
OCTET     =  %x00-FF
                    ; 8 bits of data
SP        =  %x20
                    ; space
VCHAR     =  %x21-7E
                    ; visible (printing) characters
WSP       =  SP / HTAB
                    ; white space
```

3.4 NFQL Grammar

The query language allows us to query flow records and apply different operations in every stage of NFQL engine, some of the stages can be disabled. A query file can have zero or one merger, zero or one ungrouper and one or more branches.

Ungrouper is optional depending whether the user wants to unfold the flows or not. Merger is used to merger different branches so it would make no sense to define a merger without defining branches.

Now lets say we want to apply a filter, and the filtering criteria are destination port number, the protocol used and the destination IP addresses. A possible query would be like this:

```
branch A {
  filter http-filter {
    destinationIPv4Address = 18.0.0.9 OR destinationIPv6Address = ::1
    tcpDestinationPort = 80
  }
}
```

First lets define the filter.

```
filter          = filterKeyword id "{" filter-rule-1n "}"
filterKeyword   = ''FILTER''
id              = string ; id is a string
string          = *VCHAR ; sequence of characters
filter-rule-1n  = [filter-rule filter-rule-1n] ;
filter-rule     = or-rule
or-rule         = rule-or-not opt-rule
opt-rule        = [ORKeyword rule-or-not opt-rule]
ORKeyword       = "OR"
rule-or-not     = rule
NOTKeyword      = "NOT"
rule            = infix-rule
infix-rule      = arg op arg
op              = EQ / LT / GT / LTEQ / GTEQ / ML / MG
                  / inKeyword / notinKeyword
inKeyword       = "IN"
notinKeyword    = "NOT IN"
prefix-rule     = id "(" args ")"
prefix-rule     = bitANDKeyword "(" args ")"
prefix-rule     = bitORKeyword "(" args ")"
args            = [arg "," args]
arg             = id / addr-IPv4 / addr-IPv6 / cidr_mask
                  / addr-MAC / int
arg             =/ hex / prefix_rule / string
cidr-mask       = addr-IPv4 "/" int
cidr-mask       =/ addr-IPv6 "/" int
int             = *DIGIT
hex             =   *HEXDIG
```

```
sign            = "+" / "-"
exponent        = 'e' 1*3DIGIT
right-decimal   = '.' 0*DIGIT
float           = [sign] 1*DIGIT [right-decimal] [exponent]
addr-MAC        = 2HEXDIG 5( ":" 2HEXDIG )
addr-IPv6       =                          6( h16 ":" ) ls32
                  /                        "::" 5( h16 ":" ) ls32
                  / [               h16 ] "::" 4( h16 ":" ) ls32
                  / [ *1( h16 ":" ) h16 ] "::" 3( h16 ":" ) ls32
                  / [ *2( h16 ":" ) h16 ] "::" 2( h16 ":" ) ls32
                  / [ *3( h16 ":" ) h16 ] "::"    h16 ":"   ls32
                  / [ *4( h16 ":" ) h16 ] "::"              ls32
                  / [ *5( h16 ":" ) h16 ] "::"              h16
                  / [ *6( h16 ":" ) h16 ] "::"

h16             = 1*4HEXDIG
ls32            = ( h16 ":" h16 ) / addr-IPv4
addr-IPv4       = dec-octet "." dec-octet "." dec-octet "." dec-octet
dec-octet       = DIGIT                     ; 0-9
                 / %x31-39 DIGIT            ; 10-99
                 / "1" 2DIGIT               ; 100-199
                 / "2" %x30-34 DIGIT        ; 200-249
                 / "25" %x30-35             ; 250-255
LT              = "<"
GT              = ">"
EQ              = "="
MG              = ">>"
LTEQ            = "<="
GTEQ            = ">="
ML              = "<<"
```

As we can see it all comes down to the core definitions. Now lets explain some of these rules which are related to our example given above.

filter is the *filterKeyword* as defined above and it is case sensitive, *http-filter* is the `id` which is defined as a string, in the next section we will see that an `id` is a special string which is alphanumeric, then we have defined three rules: one of the specifies the destination port with id *tcpDestinationPort* the equal sign *"="* and a value which is an integer(16-bit), the other two filter rules have id *sourceIPv4Address*, *sourceIPv6Address*, the equality operator and the value which is an IPv4, IPv6 respectively. We know the format of an IPv4 that it is made of 4 octets separated by a dot. The range is 0.0.0.0 to 255.255.255.255, we have also defined that an octet can be a single-digit, two-digit or three digit number with a maximum value 255.

The complete grammar definition as of now can be found on Appendix A, it might change in the future in case we decide to add more options to the query language.

3.5 Choosing the parser generator tool

We have some options when it comes to the parser generator tools, the options are bison[10], ANTLR[11] and some others. We also have some options on the programming language which will be used. Bison supports C/C++/Java and there is a bison python port (PLY)[12]. We are going to use bison and as a programming language Python because it is easier to write python code, development is fast, and provides all those data structures which we can make use of. The parser tool will be made of three main parts: lexer, parser and the jsonifier. We are going to use flex[10] as a lexer and as for the jsonifier, the converter to JSON format, python provides a library called json which allows JSON encoding and decoding.

We are going to use tools like abnfgen[13] in order to generate test cases which match our grammar and then feed them to our parser and check if the results are correct. Initially we have generated 100 test queries using abnfgen, the queries can be found on the github repository[2].

[2]Test Queries. [Online]. Available: https://github.com/dmorina/nfql-testing [Accessed: May 12, 2013]

4 The NFQL Parser Implementation

The NFQL Parser is divided in three main parts: the tokenizer, the parser and the jsonifier.
The tokenizer is the lexer which identifies the tokens, the parser creates the abstract syntax tree and finally the jsonifier creates the JSON parsed query file.

4.1 Lexer

In the lexer we define the reserved keywords used in the NFQL parser:

```
reserved = {
        'in': 'inKeyword',
        'notin': 'notinKeyword',
        'branch': 'branchKeyword',
        'OR': 'ORKeyword',
        'NOT': 'NOTKeyword',
        'filter':'filterKeyword',
        'grouper':'grouperKeyword',
        'groupfilter':'groupfilterKeyword',
        'merger':'mergerKeyword',
        'ungrouper':'ungrouperKeyword',
        'or': 'bitORKeyword',
        'delta':'deltaKeyword',
        'aggregation':'aggregateKeyword',
        's':'sKeyword',
        'min':'minKeyword',
        'ms':'msKeyword',
        'max':'maxKeyword',
        'sum':'sumKeyword',
        'mean':'meanKeyword',
        'static':'staticKeyword',
        'count':'countKeyword',
        'union':'unionKeyword',
        'and': 'bitANDKeyword',
        'm': 'mKeyword',
        'mi': 'miKeyword',
        'o': 'oKeyword',
        'oi': 'oiKeyword',
        'si': 'siKeyword',
        'd': 'dKeyword',
        'di': 'diKeyword',
        'f': 'fKeyword',
```

```
        'fi': 'fiKeyword',
        'eq':'eqKeyword',
    }
```

The lexer understands the following literals "+-*/(),.".
In addition to the operators defined in the reserved keywords other operators are tokenized as follows:

```
Less than or equal: r'<='
Greater than or equal: r'>='
Much less than: r'<<'
Much greater than: r'>>'
Less than: r'<'
Greater than: r'>'
Equal: '='
```

An IPv4 Address is tokenized using the regular expression below:

```
r'''
    (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)
    {3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
 '''
```

An IPv6 Address is tokenized using the regular expression below:

```
r'''
    (?:(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:)
    {0,6}(?:[0-9]{1,3}\.){3}[0-9]{1,3}(?![:.\w]))
    (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))
    (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)
    {3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)|
    (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|(?=(?:[A-F0-9]
    {0,4}:){0,7}[A-F0-9]{0,4}(?![:.\w]))(([0-9A-F]{1,4}:)
    {1,7}|:)((:[0-9A-F]{1,4}){1,7}|:))(?![:.\w])
 '''
```

The regular expressions to identify IPv4 and IPv6 addresses have been taken from Regular Expressions Cookbook [14].
MAC addresses are identified as follows:

```
r'([a-fA-F0-9]{2}[:\-]){5}[a-fA-F0-9]{2}'
```

Integers are identified using the following regular expression:

```
r'\d+'
```

New line is defined using the following regex:

```
r'\n+'
```

The *id* is identified using the following regular expression:

```
r'[a-zA-Z_][a-zA-Z_0-9]*'
```

Comments are defined using the regex below and they are like Python single line comments using the "#" symbol:

```
r"[ ]*\043[^\n]*"
```

Whitespaces are ignored by the lexer, but new lines are not ignored because they are used in the parsing rules later.
The fields for which we can query for are defined in the RFC 5102[15]. The lexer fetches the IPFIX Information Element(IE) names and their abstract data types from an XML document named *ipfix.xml* which is retrieved from [3].
An IE name without a data type entry in the XML file will be invalidated by the parser. The lexer identifies IDs, integers, MAC addresses, IPv4 and IPv6 Addresses, operators, newlines and literals as shown above.
While working on the lexer there was a problem with PLY that some regular expressions such as the ones from RFC 6021 [16], they did not match the whole IP address for example if we would write sourceIPv4Address=192.168.1.254 the lexer would recognize 192.168.1.2 as an IP address and 54 as an integer. These were the regular expressions which were used to identify IPv4 and IPv6 addresses:
IPv4

```
r'''
    (([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.)
    {3}([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])
    (%[\p{N}\p{L}]+)?
 '''
```

IPv6

```
r'''
    ((:|[0-9a-fA-F]{0,4}):)([0-9a-fA-F]{0,4}:){0,5}
    ((([0-9a-fA-F]{0,4}:)?(:|[0-9a-fA-F]{0,4}))|
    (((25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])\.){3}
    (25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])))(%[\p{N}\p{L}]+)?"""
```

This problem was fixed by using other regular expressions defined in Regular Expressions Cookbook[14].
Another thing we had to take care of was order of tokens, the two rules defined in the PLY documentation [4].

[3] IP Flow Information Export (IPFIX) Entities. [Online]. Available: http://www.iana.org/assignments/ipfix/ipfix.xml [Accessed: May 12, 2013]

[4] PLY Documentation. [Online]. Available: http://www.dabeaz.com/ply/ply.html [Accessed: May 12, 2013]

1. ”All tokens defined by functions are added in the same order as they appear in the lexer file.”
2. ”Tokens defined by strings are added next by sorting them in order of decreasing regular expression length (longer expressions are added first).”

helped to order them in the right way. Some of the tokens might be multiply defined mainly keywords since they are strings as well.

4.2 Parser & jsonifier

The Parser was written in Python3.3 and it might not work in the previous versions because of the module `ipaddress`.
The parser creates the abstract syntax tree, and also while parsing we are saving the query parameters in classes, lists and dictionaries.
Jsonifier is the main of the NFQL Parser, it reads the input and generates the JSON file if parsing was successful, otherwise it will generate an error message and halt.
The query file in the case where we would want to use all the stages of the `nfql` would expect a single merger, a single ungrouper and one or more branches. It is very important that the order is the following, first `branch`, second `merger` and third `ungrouper`, if the order is not like this the parser will give a syntax error and the parsing will stop.
A single branch may contain a single filter, a single groupfilter and a single grouper. If multiple defined the parsing will stop and the json query will not be generated. In the case when we would like to disable some of the stages we can leave them empty or not write them at all, we will give some examples later in this section.

4.2.1 Branch

Branches are defined as follows:

```
1  branch A {  }
```

New lines are allowed after `id`, for example:

```
1  branch A
2  {
3  }
```

The JSON output for both cases will be:

```
1  {
2   "branchset": [
3      {
```

```
      "groupfilter": {
        "dnf-expr": []
      },
      "filter": {
        "dnf-expr": []
      },
      "grouper": {
        "aggregation": {
          "clause": []
        },
        "dnf-expr": []
      }
    }
  ],
  "merger": {
    "dnf-expr": []
  }
}
```

We can define empty `filter`, `grouper` and `groupfilter` as follows:

```
branch A
{
   filter f1 { }
   grouper g1 { }
   groupfilter gf1 { }
}
```

The corresponding JSON output will be:

```
{
 "branchset": [
   {
     "filter": {
       "dnf-expr": [
         {
           "clause": []
         }
       ]
     },
     "groupfilter": {
       "dnf-expr": [
         {
           "clause": []
         }
```

```
          ]
        },
        "grouper": {
          "aggregation": {
            "clause": []
          },
          "dnf-expr": [
            {
              "clause": []
            }
          ]
        }
      }
    ],
    "merger": {
      "dnf-expr": []
    }
  }
```

We can also define empty `merger` and `ungrouper`:

```
branch A {
    filter f1 {}
    grouper g1 {}
    groupfilter gf1{}
}

merger {}

ungrouper {}
```

JSON output is:

```
{
  "ungrouper": {},
  "branchset": [
    {
      "filter": {
        "dnf-expr": [
          {
            "clause": []
          }
        ]
      },
      "grouper": {
```

```
          "dnf-expr": [
            {
              "clause": []
            }
          ],
          "aggregation": {
            "clause": []
          }
        },
        "groupfilter": {
          "dnf-expr": [
            {
              "clause": []
            }
          ]
        }
      }
    ],
    "merger": {
      "dnf-expr": []
    }
  }
```

Branch IDs must be unique as they are stored as keys in a dictionary.
The grammar definition for the branch is as follows:

```
branch : branchKeyword id '{' pipeline_stage_1n '}'
        | branchKeyword id '{' newline pipeline_stage_1n '}'

        | branchKeyword id '{' pipeline_stage_1n newline '}'

        | branchKeyword id '{' newline pipeline_stage_1n newline '}'

        | branchKeyword id newline '{' pipeline_stage_1n '}' newline

        | branchKeyword id newline '{' pipeline_stage_1n newline '}'

        | branchKeyword id newline '{' newline pipeline_stage_1n
          newline '}'

        | branchKeyword id newline '{' newline pipeline_stage_1n '}'
branch :
pipeline_stage_1n : pipeline_stage newline pipeline_stage_1n
                  | newline pipeline_stage newline pipeline_stage_1n
                    newline
```

```
pipeline_stage_1n :
pipeline_stage : filter
               | grouper
               | groupfilter
```

branchKeyword is `branch`, the `id` is a sequence of characters A-Z, a-z and can be combined with numbers as well as defined by the regex in the lexer, `pipeline_stage_1n` can be one or more `pipeline_stage` which is a `filter`, a `grouper` or a `groupfilter`. A branch is stored in a class `Branch` which saves the `filter`, the `grouper` and the `groupfilter` and branch `id`. At this point we assign integer IDs to the branches in the order they are defined starting from 0, the integer IDs are used later in the merger stage. We map each branch ID which is a sequence of characters as defined above to an integer value. And the reason why we do this is that `nfql` expects integer IDs in the JSON file.
We map the data types and operators to the IPFIX Entity Model abstract data types and operator keywords.
The mapping is as follows:

```
datatype_mappings = {
        'unsigned64': 'RULE_S1_64', 'unsigned32': 'RULE_S1_32',
        'unsigned16': 'RULE_S1_16', 'unsigned8': 'RULE_S1_8',
        'EQ':'RULE_EQ','GT':'RULE_GT','LT':'RULE_LT',
        'LTEQ':'RULE_LE', 'GTEQ':'RULE_GE','IN':'RULE_IN',
        'SUM':'RULE_SUM','COUNT':'RULE_COUNT','STATIC':'RULE_STATIC',
        'ABS':'RULE_ABS','REL':'RULE_REL','PROD':'RULE_PROD',
        'STDDEV':'RULE_STDDEV','BITOR':'RULE_OR',
        'BITAND':'RULE_AND', 'MIN':'RULE_MIN','MAX':'RULE_MAX',
        'UNION':'RULE_UNION','MEDIAN':'RULE_MEDIAN',
        'MEAN':'RULE_MEAN', 'ipv4Address': 'RULE_S1_32',
        'ipv6Address': 'RULE_S1_128', 'macAddress': 'RULE_S1_48',
}
```

4.2.2 Filter

A `filter` is defined inside a `branch`, for example:

```
1 branch A {
2   filter ospfigp {
3     destinationIPv4Address=224.0.0.5 OR destinationIPv4Address=224.0.0.6
4     protocolIdentifier=OSPFIGP
5   }
6 }
```

The respective JSON would look like this:

```
{
  "branchset": [
    {
      "filter": {
        "dnf-expr": [
          {
            "clause": [
              {
                "term": {
                  "offset": {
                    "datatype": "RULE_S1_8",
                    "value": 86,
                    "name": "protocolIdentifier"
                  },
                  "op": "RULE_EQ",
                  "delta": 0
                }
              },
              {
                "term": {
                  "offset": {
                    "datatype": "RULE_S1_32",
                    "value": "224.0.0.5",
                    "name": "destinationIPv4Address"
                  },
                  "op": "RULE_EQ",
                  "delta": 0
                }
              }
            ]
          },
          {
            "clause": [
              {
                "term": {
                  "offset": {
                    "datatype": "RULE_S1_8",
                    "value": 86,
                    "name": "protocolIdentifier"
                  },
                  "op": "RULE_EQ",
                  "delta": 0
                }
              },
```

```
                {
                  "term": {
                    "offset": {
                      "datatype": "RULE_S1_32",
                      "value": "224.0.0.6",
                      "name": "destinationIPv4Address"
                    },
                    "op": "RULE_EQ",
                    "delta": 0
                  }
                }
              ]
            }
          ]
        },
        "grouper": {
          "aggregation": {
            "clause": []
          },
          "dnf-expr": []
        },
        "groupfilter": {
          "dnf-expr": []
        }
      }
    ],
    "merger": {
      "dnf-expr": []
    }
}
```

The filter grammar rules are:

```
filter : filterKeyword id '{' filter_rule_1n '}'
filter_rule_1n : filter_rule newline filter_rule_1n
filter_rule_1n :
filter_rule : or_rule
or_rule : rule_or_not opt_rule
opt_rule :
opt_rule : ORKeyword rule_or_not opt_rule
rule_or_not : rule
rule : infix_rule
infix_rule : arg_names op arg deltaKeyword delta_arg
infix_rule : arg_names op arg
op : EQ
```

```
        | LT
        | GT
        | LTEQ
        | GTEQ
        | ML
        | MG
        | inKeyword
        | notinKeyword
arg :     IPv6
            | IPv4
            | CIDR
            | MAC
            | int
            | args
arg_names : id
args : '(' args_n ')'
args_n : arg ','  args_n
args_n : arg
args_n :
CIDR : IPv4 '/' int
             | IPv6 '/' int
```

The filterKeyword is `filter`, `filter_rule_1n` is one or more filter rules. Filter rules are written one per line and they are connected by an implicit `AND`, but we can write multiple rules in one line connected via an `OR` operator. The filter rule expects a valid Information Element defined in the IPFIX Information Model [15], an operator and a value, and optionally a delta. If we specify `tcpSourcePort=20 delta 2` this means that `tcpSourcePort` can be any value in the range 18 to 22.

The `nfql` expects the filter in JSON format in Disjunctive Normal Form (DNF) so as the filter rules are not define in DNF we had to convert them to this form. As we said we expect a filter rule per line or more rules connected via `OR` operator, and after each newline there is an implicit `AND`. To convert these to DNF we save each line in a list and if multiple rules in a list they are added to the same list. Then we add all the rules to another list. To get the DNF all we need to do is to find the combinations of these list elements but not the ones inside the same list.

To convert the example above to DNF we add the first two rules to a list ['destinationIPv4Address=224.0.0.5', 'destinationIPv4Address=224.0.0.6'] and the third rule to another list ['protocolIdentifier=86']

Now if we do the combination of these lists we get the following lists:

```
['destinationIPv4Address=224.0.0.5','protocolIdentifier=86']
['destinationIPv4Address=224.0.0.6','protocolIdentifier=86']
```

This way we get the DNF,

```
destinationIPv4Address=224.0.0.5 and protocolIdentifier=86
or
destinationIPv4Address=224.0.0.6 and protocolIdentifier=86
```

At this point the elements each list is a clause and the elements of these lists will be terms of that clause. Note that there is no clear mapping between a `rule` and a `clause` or a `term` in the JSON file.
Protocol identifier is an 8-bit number and is defined by Internet Assigned Numbers Authority(IANA). There is a feature added to the parser that it understands also the protocol names not only the numbers so if you wanted protocolIdentifier to be the one for TCP you could write protocolIdentifier=6 or protocolIdentifier=TCP.
Below you can find the protocol numbers:

```
protocol_numbers_mappings = {'HOPOPT': 0, 'ICMP': 1, 'IGMP': 2,
      'GGP': 3, 'IPv4': 4, 'ST': 5, 'TCP': 6, 'CBT': 7, 'EGP': 8,
      'IGP':9,'BBN-RCC-MON':10,'NVP-II':11,'PUP':12,'ARGUS':13,
      'EMCON':14,'XNET':15,'CHAOS':16,'UDP':17,'MUX':18,
      'DCN-MEAS':19,'HMP':20,'PRM':21,'XNS-IDP':22,'TRUNK-1':23,
      'TRUNK-2':24, 'LEAF-1':25, 'LEAF-2':26, 'RDP':27, 'IRTP':28,
      'ISO-TP4':29, 'NETBLT':30,'MFE-NSP':31, 'MERIT-INP':32,
      'DCCP':33, '3PC':34, 'IDPR':35, 'XTP':36, 'DDP':37,
      'IDPR-CMTP':38,'TP++':39,'IL':40,'IPv6':41,'SDRP':42,
      'IPv6-Route':43,'IPv6-Frag':44, 'IDRP':45, 'RSVP':46,
      'GRE':47, 'DSR':48, 'BNA':49, 'ESP':50, 'AH':51, 'I-NLSP':52,
      'SWIPE':53,'NARP':54,'MOBILE':55,'TLSP':56,'SKIP':57,
      'IPv6-ICMP':58,'IPv6-NoNxt':59,'IPv6-Opts':60, 'CFTP':61,
      'SAT-EXPAK':62, 'KRYPTOLAN':63, 'RVD':64, 'IPPC':65,
      'SAT-MON':66,'VISA':67,'IPCV':68,'CPNX':69,'CPHB':70,
      'WSN':71,'PVP':72,'BR-SAT-MON':73,'SUN-ND':74, 'WB-MON':75,
      'WB-EXPAK':76, 'ISO-IP':77, 'VMTP':78, 'SECURE-VMTP':79,
      'VINES':80,'TTP':81, 'IPTM':82, 'NSFNET-IGP':83, 'DGP':84,
      'TCF':84, 'EIGRP':85, 'OSPFIGP':86,'Sprite-RPC':87,'LARP':88,
      'MTP':89,'AX.25':90,'IPIP':91,'MICP':92,'SCC-SP':93,
      'ETHERIP':94,'ENCAP':95,'GMTP':96,'IFMP':97,'PNNI':98,
      'PIM':99,'ARIS':100, 'SCPS':101, 'QNX':102, 'A/N':103,
      'IPComp':104, 'SNP':105, 'Compaq-Peer':106, 'IPX-in-IP':107,
      'VRRP':108,'PGM':109,'L2TP':110,'DDX':111,'IATP':112,
      'STP':113,'SRP':114, 'UTI':115, 'SMP':116, 'SM':117, 'PTP':118,
      'ISIS over IPv4':119, 'FIRE':120, 'CRTP':121, 'CRUDP':122,
      'SSCOPMCE':123, 'IPLT':124,'SPS':125, 'PIPE':126, 'SCTP':127,
      'FC':128, 'RSVP-E2E-IGNORE':129, 'Mobility Header':130,
      'UDPLite':131, 'MPLS-in-IP':132, 'manet':133,'HIP':134,
      'Shim6':135,'WESP':136,'ROHC':137
}
```

The filter can be left as empty if the user wouldn't like to filter for anything this can be done in two ways: by writing an empty filter or by not writing the filter at all.
If it is an empty filter:

```
branch A {
    filter f1 {}
}
```

The corresponding JSON output:

```
{
  "branchset": [
    {
      "filter": {
        "dnf-expr": [
          {
            "clause": []
          }
        ]
      },
      "groupfilter": {
        "dnf-expr": []
      },
      "grouper": {
        "aggregation": {
          "clause": []
        },
        "dnf-expr": []
      }
    }
  ],
  "merger": {
    "dnf-expr": []
  }
}
```

If the filter is not written:

```
branch A {
}
```

the JSON output will be:

```
{
  "branchset": [
    {
```

```
        "groupfilter": {
          "dnf-expr": []
        },
        "filter": {
          "dnf-expr": []
        },
        "grouper": {
          "aggregation": {
            "clause": []
          },
          "dnf-expr": []
        }
      }
    ],
    "merger": {
      "dnf-expr": []
    }
  }
```

4.2.3 Grouper

The grouper is very similar to the filter, and example of a grouper would be:

```
grouper g1
    {
        tcpSourcePort=80 OR sourceIPv6Address=destinationIPv6Address
        tcpDestinationPort=tcpSourcePort
        aggregation
        {
            sum(packetDeltaCount)
        }
    }
```

The corresponding JSON query is:

```
{
  "branchset": [
    {
      "filter": {
        "dnf-expr": []
      },
      "grouper": {
        "aggregation": {
          "clause": [
```

```
            {
              "term": {
                "op": "RULE_SUM",
                "offset": {
                  "name": "packetDeltaCount",
                  "datatype": "RULE_S1_64"
                }
              }
            }
          ]
        },
        "dnf-expr": [
          {
            "clause": [
              {
                "term": {
                  "offset": {
                    "f1_datatype": "RULE_S1_16",
                    "f2_datatype": "RULE_S2_16",
                    "f2_name": "tcpSourcePort",
                    "f1_name": "tcpDestinationPort"
                  },
                  "delta": 0,
                  "op": {
                    "type": "RULE_REL",
                    "name": "RULE_EQ"
                  }
                }
              },
              {
                "term": {
                  "offset": {
                    "f1_datatype": "RULE_S1_16",
                    "f2_datatype": "RULE_S2_16",
                    "f2_name": 80,
                    "f1_name": "tcpSourcePort"
                  },
                  "delta": 0,
                  "op": {
                    "type": "RULE_ABS",
                    "name": "RULE_EQ"
                  }
                }
              }
```

```
            ]
          },
          {
            "clause": [
              {
                "term": {
                  "offset": {
                    "f1_datatype": "RULE_S1_16",
                    "f2_datatype": "RULE_S2_16",
                    "f2_name": "tcpSourcePort",
                    "f1_name": "tcpDestinationPort"
                  },
                  "delta": 0,
                  "op": {
                    "type": "RULE_REL",
                    "name": "RULE_EQ"
                  }
                }
              },
              {
                "term": {
                  "offset": {
                    "f1_datatype": "RULE_S1_128",
                    "f2_datatype": "RULE_S2_128",
                    "f2_name": "destinationIPv6Address",
                    "f1_name": "sourceIPv6Address"
                  },
                  "delta": 0,
                  "op": {
                    "type": "RULE_REL",
                    "name": "RULE_EQ"
                  }
                }
              }
            ]
          }
        ]
      },
      "groupfilter": {
        "dnf-expr": []
      }
    }
  ],
  "merger": {
```

```
98      "dnf-expr": []
99    }
100 }
```

Grouper contains the grouping rules and the aggregation rules.
Grammar rules for the grouper are:

```
grouper : grouperKeyword id '{' grouper_rule1_n aggregate '}'
grouper_rule1_n : grouper_rule_n newline grouper_rule1_n
grouper_rule1_n :
grouper_rule_n : grouper_or_rule
grouper_or_rule : grouper_rule g_opt_rule
g_opt_rule :
g_opt_rule : ORKeyword grouper_rule g_opt_rule
grouper_rule : id grouper_op id  //relative rules
grouper_rule : id grouper_op g_arg //absolute rules
g_arg : IPv6
            | IPv4
            | CIDR
            | MAC
            | int
grouper_rule : id grouper_op id deltaKeyword delta_arg
grouper_op : EQ
                    | LT
                    | GT
                    | GTEQ
                    | LTEQ
delta_arg :     time
                    | int
time :  int sKeyword
                | int msKeyword
                | int minKeyword
aggregate :
aggregate : aggregateKeyword '{' aggr1_n '}'
aggr1_n : aggr_rule newline aggr1_n
aggr1_n :
aggr_rule : aggr_op '(' id ')'
aggr_op : minKeyword
                | maxKeyword
                | sumKeyword
                | avgKeyword
                | staticKeyword
                | unionKeyword
                | countKeyword
                | bitANDKeyword
```

```
                | bitORKeyword
                | id
```

The rules are very similar to the filter rules, as before grouper rules are in DNF, grouper rules can be relative or absolute, if it is relative we have Information Element operator Information Element, if it is absolute we have Information Element operator value.
In the example given earlier the grouper rule `tcpSourcePort=80` is an absolute rule, the other two rules `sourceIPv6Address=destinationIPv6Address`, `tcpDestinationPort=tcpSourcePort` are relative rules.
Supported aggregation functions are SUM, MIN, MAX, MEAN, STATIC, UNION, COUNT, AND, OR.
Since there is only one aggregation per grouper we dont need an ID. Grouper can be empty or not written at all, same goes for aggregation.
If aggregation is not written:

```
branch A {
    filter f1 {}
    grouper g1
    {
        sourceIPv4Address=destinationIPv4Address
    }
    groupfilter gf1{}

}
```

the JSON output will be:

```
{
  "merger": {
    "dnf-expr": []
  },
  "branchset": [
    {
      "groupfilter": {
        "dnf-expr": [
          {
            "clause": []
          }
        ]
      },
      "filter": {
        "dnf-expr": [
          {
            "clause": []
```

```
            }
          ]
        },
        "grouper": {
          "dnf-expr": [
            {
              "clause": [
                {
                  "term": {
                    "op": {
                      "type": "RULE_REL",
                      "name": "RULE_EQ"
                    },
                    "delta": 0,
                    "offset": {
                      "f1_datatype": "RULE_S1_32",
                      "f2_datatype": "RULE_S2_32",
                      "f2_name": "destinationIPv4Address",
                      "f1_name": "sourceIPv4Address"
                    }
                  }
                }
              ]
            }
          ],
          "aggregation": {
            "clause": []
          }
        }
      }
    ]
  }
```

If grouper is empty:

```
branch A {
   grouper g1 {}
}
```

the JSON output will be:

```
{
  "merger": {
    "dnf-expr": []
  },
```

```
  "branchset": [
    {
      "grouper": {
        "dnf-expr": [
          {
            "clause": []
          }
        ],
        "aggregation": {
          "clause": []
        }
      },
      "filter": {
        "dnf-expr": []
      },
      "groupfilter": {
        "dnf-expr": []
      }
    }
  ]
}
```

If the grouper is not written the output will be the same as in the example where we did not define a filter as well.

```
{
  "branchset": [
    {
      "groupfilter": {
        "dnf-expr": []
      },
      "filter": {
        "dnf-expr": []
      },
      "grouper": {
        "aggregation": {
          "clause": []
        },
        "dnf-expr": []
      }
    }
  ],
  "merger": {
    "dnf-expr": []
  }
```

```
}
```

4.2.4 Groupfilter

GroupFilter is about the same as the Filter, an example would be the following:

```
branch A {
  groupfilter gf1
      {
          packetDeltaCount > 800 delta 7
      }
}
```

And the corresponding JSON is the following:

```
{
  "merger": {
    "dnf-expr": []
  },
  "branchset": [
    {
      "groupfilter": {
        "dnf-expr": [
          {
            "clause": [
              {
                "term": {
                  "delta": 7,
                  "op": "RULE_GT",
                  "offset": {
                    "name": "packetDeltaCount",
                    "value": 800,
                    "datatype": "RULE_S1_64"
                  }
                }
              }
            ]
          }
        ]
      },
      "grouper": {
        "dnf-expr": [],
        "aggregation": {
          "clause": []
```

```
          }
        },
        "filter": {
          "dnf-expr": []
        }
      }
    ]
}
```

GroupFilter grammar rules are:

```
groupfilter : groupfilterKeyword id '{' groupfilter_rule_1n '}'
groupfilter_rule_1n : groupfilter_rule newline groupfilter_rule_1n
groupfilter_rule_1n :
groupfilter_rule : gf_or_rule
gf_or_rule : gf_rule gf_opt_rule
gf_opt_rule :
gf_opt_rule : ORKeyword gf_rule gf_opt_rule
gf_rule : arg_names op arg
gf_rule : arg_names op arg deltaKeyword delta_arg
```

If we do not want a groupfilter we just write an empty groupfilter or we dont write the groupfilter at all. In the case when the groupfilter is empty:

```
branch A {
        groupfilter gf1 {}
}
```

the JSON output will be:

```
{
  "branchset": [
    {
      "grouper": {
        "dnf-expr": [],
        "aggregation": {
          "clause": []
        }
      },
      "groupfilter": {
        "dnf-expr": [
          {
            "clause": []
          }
        ]
      },
```

```
        "filter": {
          "dnf-expr": []
        }
      }
    ],
    "merger": {
      "dnf-expr": []
    }
  }
```

If the groupfilter is not written the output will be as in the example with an empty grouper and empty filter:

```
  {
   "branchset": [
      {
        "groupfilter": {
          "dnf-expr": []
        },
        "filter": {
          "dnf-expr": []
        },
        "grouper": {
          "aggregation": {
            "clause": []
          },
          "dnf-expr": []
        }
      }
    ],
    "merger": {
      "dnf-expr": []
    }
  }
```

4.2.5 Merger

Merger is a bit different from the others because it uses predefined branches as it is used to merge different branches.
An example would be:

```
merger {
    A.sourceIPv4Address = B.destinationIPv4Address
    A.destinationIPv6Address = B.sourceIPv6Address
```

```
}
```

And the corresponding JSON would be:

```
"merger": {
    "dnf-expr": [
      {
        "clause": [
          {
            "term": {
              "branch1_id": 0,
              "op": {
                "name": "RULE_EQ",
                "type": "RULE_REL"
              },
              "delta": 0,
              "branch2_id": 1,
              "offset": {
                "f1_name": "destinationIPv6Address",
                "f2_datatype": "RULE_S2_128",
                "f1_datatype": "RULE_S1_128",
                "f2_name": "sourceIPv6Address"
              }
            }
          },
          {
            "term": {
              "branch1_id": 0,
              "op": {
                "name": "RULE_EQ",
                "type": "RULE_REL"
              },
              "delta": 0,
              "branch2_id": 1,
              "offset": {
                "f1_name": "sourceIPv4Address",
                "f2_datatype": "RULE_S2_32",
                "f1_datatype": "RULE_S1_32",
                "f2_name": "destinationIPv4Address"
              }
            }
          }
        ]
      }
    ]
```

```
42    }
```

As we mentioned before we assign integer IDs to the branches in the order they are defined in the query file, these IDs are used in the `merger` as we can see in the JSON output.
The grammar rules for the merger are:

```
merger : mergerKeyword '{' merger_rule_1n '}'
merger :
merger_rule_1n : merger_rule newline merger_rule_1n
merger_rule_1n :
merger_rule : merger_or_rule
merger_or_rule : merger_m_rule merger_opt_rule
merger_opt_rule :
merger_opt_rule : ORKeyword merger_m_rule merger_opt_rule
merger_m_rule : id '.' id allen_op id '.' id
allen_op : LT
          | GT
          | EQ
          | mKeyword
          | miKeyword
          | oKeyword
          | oiKeyword
          | sKeyword
          | siKeyword
          | dKeyword
          | diKeyword
          | fKeyword
          | fiKeyword
          | eqKeyword
```

We allow empty merger on undefined merger, they are the same.

```
1  branch A {
2     filter f1 {}
3     grouper g1 {}
4     groupfilter gf1{}
5  }
6
7  merger {}
```

The JSON output will be the same in both cases:

```
1  {
2   "branchset": [
3     {
```

```
        "filter": {
          "dnf-expr": [
            {
              "clause": []
            }
          ]
        },
        "grouper": {
          "dnf-expr": [
            {
              "clause": []
            }
          ],
          "aggregation": {
            "clause": []
          }
        },
        "groupfilter": {
          "dnf-expr": [
            {
              "clause": []
            }
          ]
        }
      }
    ],
    "merger": {
      "dnf-expr": []
    }
  }
```

4.2.6 Ungrouper

Ungrouper doesn't have any rules, if defined ungrouping will be requested from the engine, if not it will not be requested. Ungrouping means that the flow records will be unfolded by `nfql`.
If we want to ungroup the flows we define ungrouper as follows

```
branch A {
   filter f1 {}
   grouper g1 {}
   groupfilter gf1{}
}
```

```
merger {}
ungrouper {}
```

The JSON output will be:

```
{
 "ungrouper": {},
 "branchset": [
   {
     "filter": {
       "dnf-expr": [
         {
           "clause": []
         }
       ]
     },
     "grouper": {
       "dnf-expr": [
         {
           "clause": []
         }
       ],
       "aggregation": {
         "clause": []
       }
     },
     "groupfilter": {
       "dnf-expr": [
         {
           "clause": []
         }
       ]
     }
   }
 ],
 "merger": {
   "dnf-expr": []
 }
}
```

4.2.7 Error Checking

The errors which can be identified are: invalid data types, invalid IE names and keywords, more than one definition of merger, filter, grouper or groupfilter.
In the cases when the user defines `filter`, `grouper` or `groupfilter` outside the `branch` the parser will generate an error "Syntax error, line n" where n is the line number where one of the stages is being defined at. It is the same if we define `aggregation` outside the `grouper`, or if we define `merger`, `ungrouper` inside `branch`. No JSON file will be generated in all these cases.
If we define the `groupfilter` but we do not define the `grouper` as below:

```
1  branch A {
2      filter f1 {}
3      grouper g1
4      {
5
6      }
7      groupfilter gf1
8      {
9          packetDeltaCount > 800
10     }
11 }
```

there will be no JSON file generated and the following error message will be produced:

```
'Groupfilter defined without a grouper!'
```

If we try to use an undefined branch in the merger for example:

```
1  branch A {
2      filter f1 {}
3      grouper g1
4      {
5
6      }
7      groupfilter gf1
8      {
9          packetDeltaCount > 800
10     }
11 }
12 merger {
13     A.sourceIPv4Address = B.destinationIPv4Address OR A.tcpSourcePort = B
14     A.destinationIPv6Address = B.sourceIPv6Address
15 }
```

there will be no JSON file generated and the following error message will be produced:

```
'Undefined branch id 'B' used line 13.'
```

In case the query file is empty then no JSON file will be generated. Error message:

```
'Empty query file!'
```

If multiple filters defined inside one branch:

```
branch A {
    filter f1
    {
        tcpSourcePort=443 delta 7
        sourceIPv4Address in 192.168.0.0/30
    }
    filter f2
    {
        tcpSourcePort=80 delta 7
        sourceIPv4Address in 192.168.0.0/31
    }
    grouper g1{

    }

}
```

there will be no JSON file generated and the following error message will be produced:

```
'Multiple filters defined inside one branch'
```

Same goes if multiple groupers and groupfilters are defined inside one branch. Error messages:

```
'Multiple groupers defined inside one branch'
```

```
'Multiple groupfilters defined inside one branch'
```

In case the branch ID is being reused in the query file:

```
branch A {
    filter f
    {
        tcpSourcePort=443 delta 7
        sourceIPv4Address in 192.168.0.0/30
    }
```

```
    grouper g
    {

    }

}

branch A {
    filter f
    {
        tcpSourcePort=443 delta 7
        sourceIPv4Address in 192.168.0.0/30
    }
    grouper g
    {

    }

}
```

there will be no JSON file generated and the following error message will be produced:

```
'Branch ID 'A' already used, line 14'
```

If the user is trying to apply the aggregation operators SUM, MEAN, UNION for example:

```
branch A {
    filter v4
    {
        tcpSourcePort=443 delta 7
        sourceIPv4Address in 192.168.0.0/30
    }
    grouper g1
    {
      sourceIPv4Address=destinationIPv4Address
      aggregation {
        sum(sourceIPv4Address)
      }
    }

}
```

there will be no JSON file generated and the following error message will be produced:

```
'''Operators SUM, MEAN and UNION are not allowed
on data types ipv4Address, ipv6Address and macAddress'''
```

5 Future Work

The parser is working very good but there might be some bugs especially with new lines. The has been some testing done for the Filter stage but not for the others and not for the complete Parser.
The Parser needs a regression test suite, and also some more semantic error checking if not all of the cases are covered.

6 Conclusions

The task was very interesting and challenging as well, it was difficult to get the Filter working after that everything was easy. The project is completed successfully and the code can be modified easily for further extensions that can be made or language modifications in the future. As agreed the parser allows querying above 300 entities which are defined in the IPFIX entities document but the NFQL engine currently supports about 20 of them. This will lead to a segmentation fault in the engine. There are also some data types which are defined in the same document but the engine does not support them yet. Everything was completed on time, but there was not much of testing done.

7 Appendix A

Complete grammar definition in ABNF [17]

```
 ; grammar definition
stage                       = [branches] [merger] [ungrouper]
branches                 = branch CRLF branches
branches                  =/
branch                   = branchKeyword id '{' pipeline_stage_1n '}'
pipeline-stage-1n        = [pipeline-stage pipeline-stage-1n]
pipeline-stage           = filter
pipeline-stage                =/ grouper
pipeline-stage           =/ groupfilter
;filter
filter                   = filterKeyword id "{"
                           filter-rule-1n "}"
filterKeyword            = "FILTER"
id                         = 1*string
string                   = ALPHA
filter-rule-1n           = [filter-rule CRLF filter-rule-1n]
filter-rule              = or-rule
or-rule                  = rule-or-not opt-rule
opt-rule                 = [ORKeyword rule-or-not opt-rule]
ORKeyword                = "OR"
rule-or-not              = rule
NOTKeyword               = "NOT"
rule                     = infix-rule
infix-rule               = arg_names op arg deltaKeyword delta_arg
infix-rule               =/ arg_names op arg
op                       = EQ / LT / GT / LTEQ / GTEQ / ML / MG /
                           inKeyword / notinKeyword
inKeyword                = "IN"
notinKeyword             = "NOT IN"
LT                       = "<"
GT                       = ">"
EQ                       = "="
MG                       = ">>"
LTEQ                     = "<="
GTEQ                     = ">="
ML                       = "<<"
args                     = ''('' args_n '')''
args_n                   = arg '',''  args_n
args_n                   =/ arg
```

```
args_n              =/
arg                 = id / addr-IPv4 / addr-IPv6 / cidr-mask /
                      addr-MAC / int / args
arg_names           = id
cidr-mask           = addr-IPv4 "/" int
cidr-mask           =/ addr-IPv6 "/" int
addr-IPv6   =                            6( h16 ":" ) ls32
                /                        "::" 5( h16 ":" ) ls32
                / [                h16 ] "::" 4( h16 ":" ) ls32
                / [ *1( h16 ":" ) h16 ] "::" 3( h16 ":" ) ls32
                / [ *2( h16 ":" ) h16 ] "::" 2( h16 ":" ) ls32
                / [ *3( h16 ":" ) h16 ] "::"    h16 ":"   ls32
                / [ *4( h16 ":" ) h16 ] "::"              ls32
                / [ *5( h16 ":" ) h16 ] "::"              h16
                / [ *6( h16 ":" ) h16 ] "::"

h16         = 1*4HEXDIG
ls32        = ( h16 ":" h16 ) / addr-IPv4
addr-IPv4   = dec-octet "." dec-octet "." dec-octet "." dec-octet
addr-MAC    = 2HEXDIG 5( ":" 2HEXDIG )
dec-octet   = DIGIT                     ; 0-9
              / %x31-39 DIGIT           ; 10-99
              / "1" 2DIGIT              ; 100-199
              / "2" %x30-34 DIGIT       ; 200-249
              / "25" %x30-35            ; 250-255
int         = 1*DIGIT
hex         = *HEXDIG
sign        = "+" / "-"
exponent    = 'e' 1*3DIGIT
right-decimal = '.' 0*DIGIT
float         = [sign] 1*DIGIT [right-decimal] [exponent]
grouper     =  grouperKeyword id '{' grouper_rule1_n aggregate '}'
grouperKeyword      = "grouper"
grouper_rule1_n     = [grouper_rule_n CRLF grouper_rule1_n]
grouper_rule_n      = grouper_or_rule
grouper_or_rule     = grouper_rule [g_opt_rule]
g_opt_rule          = ORKeyword grouper_rule g_opt_rule
grouper_rule        = id grouper_op id
grouper_rule        =/ id grouper_op g_arg
g_arg               = IPv6
/ IPv4

                     / CIDR
/ MAC / int
```

```
grouper_rule        =/ id grouper_op id deltaKeyword delta_arg
deltaKeyword            = "delta"
grouper_op          = EQ
/ LT

                     / GT / GTEQ
/ LTEQ
delta_arg           =   time
/ int
time                =  int sKeyword

time                =/ int msKeyword

time                =/ int minKeyword
sKeyword             = "s"
msKeyword           = "ms"
minKeyword          = "min"
aggregate           = [aggregateKeyword '{' aggr1_n '}']
aggregateKeyword    = "aggregation"
aggr1_n             = [aggr_rule CRLF aggr1_n]
aggr_rule           = aggr-op '(' id ')'
aggr-op             = minKeyword / maxKeyword / sumKeyword / meanKeyword
                      / staticKeyword
                     / unionKeyword / countKeyword / bitANDKeyword
                     / bitORKeyword
minKeyword          = "MIN"
maxKeyword          = "MAX"
sumKeyword          = "SUM"
meanKeyword         = "MEAN"
unionKeyword        = "UNION"
staticKeyword       = "STATIC"
countKeyword        = "COUNT"
bitANDKeyword       = "and"
bitORKeyword        = "or"

;group-filter
group-filter        = groupFilterKeyword id "{" filter-rule-1n "}"
groupFilterKeyword  = "GROUPFILTER"

;ungrouper
ungrouper           = ungrouperKeyword  id  "{" SP "}"
ungrouperKeyword    = "UNGROUPER"
;merger
merger              = [mergerKeyword '{' merger_rule_1n '}']
```

```
mergerKeyword       = "MERGER"
merger_rule_1n      = [merger_rule CRLF merger_rule_1n]
merger_rule         = merger_or_rule
merger_or_rule      = merger_m_rule [merger_opt_rule]
merger_opt_rule     = ORKeyword merger_m_rule merger_opt_rule
merger_m_rule       = id '.' id allen-op id '.' id
allen-op            = LT / GT / EQ / mKeyword / miKeyword /
                      oKeyword / oiKeyword / sKeyword /
                      siKeyword / dKeyword / diKeyword /
                      fKeyword / fiKeyword / eqKeyword
mKeyword            = "M"
miKeyword           = "MI"
oKeyword            = "O"
oiKeyword           = "OI"
sKeyword            = "S"
siKeyword           = "SI"
dKeyword            = "D"
diKeyword           = "DI"
fKeyword            = "F"
fiKeyword           = "FI"
eqKeyword           = "EQ"

;other

boolean-yes     = "1" / "y" / "Y" / "t" / "T"
boolean-no      = "0" / "n" / "N" / "f" / "F"
boolean         = boolean-yes / boolean-no
date-fullyear   = 4DIGIT
date-month      = 2DIGIT  ; 01-12
date-mday       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on
                          ; month/year
time-hour       = 2DIGIT  ; 00-23
time-minute     = 2DIGIT  ; 00-59
time-second     = 2DIGIT  ; 00-58, 00-59, 00-60 based on leap second
                          ; rules
time-ms         = 1DIGIT / 2DIGIT / 3DIGIT
time-secfrac    = "." 1*DIGIT
time-numoffset  = ("+" / "-") time-hour ":" time-minute
time-offset     = "Z" / time-numoffset
partial-time    = time-hour ":" time-minute ":" time-second
                  [time-secfrac]
full-date       = date-fullyear "-" date-month "-" date-mday
full-time       = partial-time time-offset
date-time       = full-date "T" full-time
```

References

[1] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[2] B. Trammell and E. Boschi. An Introduction to IP Flow Information Export (IPFIX). *Communications Magazine, IEEE*, 49(4):89 –95, April 2011.

[3] Kevin Dooley and Ian Brown. *Cisco IOS Cookbook*. O'Reilly Media, Inc., 2006.

[4] Vaibhav Bajpai. A Complete System Integration of the Network Flow Query Language. Master's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, August 2012.

[5] Haag, P. NFDUMP. [Online]. Available: http://nfdump.sourceforge.net [Accessed: December 19, 2012].

[6] C.M. Inacio and B. Trammell. YAF: Yet Another Flowmeter. In *Proceedings of the 24th international conference on Large installation system administration*, pages 1–16. USENIX Association, 2010.

[7] Vladislav Marinov and Jürgen Schönwälder. Design of a Stream-Based IP Flow Record Query Language. *DSOM 2009, Venice*, October 2009.

[8] Vaibhav Bajpai, Johannes Schauer, and Jürgen Schönwälder. An Efficient Tool for Querying Network Flow Records. *13th IFIP/IEEE International Symposium on Integrated Network Management, Ghent*, May 2013 (to appear).

[9] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 5234 (Standard), January 2008.

[10] J. Levine. *Flex & Bison*. O'Reilly Media, Incorporated, 2009.

[11] T.J. Parr and R.W. Quong. ANTLR: A Predicated-LL (k) Parser Generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[12] Beazley, D. PLY (Python Lex-Yacc). [Online]. Available: http://www.dabeaz.com/ply/ [Accessed: December 19, 2012].

[13] Jutta Degener. ABNFGEN. [Online]. Available: http://www.quut.com/abnfgen [Accessed: December 19, 2012].

[14] Jan Goyvaerts and Steven Levithan. *Regular expressions cookbook*. O'reilly, 2009.

[15] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer. Information Model for IP Flow Information Export. RFC 5102 (Proposed Standard), January 2008. Updated by RFC 6313.

[16] J. Schoenwaelder. Common YANG Data Types. RFC 6021 (Proposed Standard), October 2010.

[17] Kaloyan Kanev. Source Code of the Previous Parser. [Online]. Available: https://github.com/vbajpai/nfql/blob/master/parser/src/parser.py [Accessed: December 19, 2012].

[18] Kaloyan Kanev. Flowy - Network Flow Analysis Application. Master's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, August 2009.

[19] Kaloyan Kanev, Nikolay Melnikov, and Jürgen Schönwälder. Implementation of a stream-based IP flow record query language. In *Proceedings of the Mechanisms for autonomous management of networks and services, and 4th international conference on Autonomous infrastructure, management and security*, AIMS'10, pages 147–158, Berlin, Heidelberg, 2010. Springer-Verlag.