Network Protocol Design and Evaluation

Exercise 4

Stefan Rührup

University of Freiburg
Computer Networks and Telematics
Summer 2009
Task 1

Task 1  ABNF

An eMail server accepts a comma-separated list of the recipients' email addresses. Valid eMail addresses consist of a name, the at-symbol, and a domain name. Invalid addresses such as "@ietf.org" or "bg@ms..gov" should not be accepted. Write an ABNF specification for this list and implement a parser for your specified grammar.
Task 1 - ABNF

(Note: this grammar doesn’t cover all valid email addresses)
Call back function for the rule “item” generated by APG with user-defined code

```c
ulong ApgMyParser::pfn_item(void* vpData, ulong ulState, ulong ulOffset, ulong ulLen)
{
    ulong ulReturn = PFN_OK;
    //{{{CallBack.pfn_item
    SEMANTIC_DATA* spData = (SEMANTIC_DATA*)vpData;
    switch(ulState)
    {
    case SYN_PRE:   // pre syntax analysis
        break;
    case SYN_NOMATCH: // fill in your code here
        break;
    case SYN_EMPTY:  // fill in your code here
        break;
    case SYN_MATCH:  // fill in your code here
        break;
    case SEM_PRE:    // fill in your code here
        break;
    case SEM_POST:   // fill in your code here
        char caText[1028];
        memcpy((void*)caText, (void*)spData->ucpSrc[ulOffset], ulLen);
        caText[ulLen] = 0;
        cout << "eMail address: " << caText << endl;
        break;
    }
    //}}}CallBack.pfn_item
    return ulReturn;
}
```

Call back function for the rule "item" generated by APG with user-defined code

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
Try it on your own!

Hopefully you get a parser that accepts valid lists and rejects invalid ones...

---

### Task 1 - ABNF

```text
*** Parser State ***
  state = MATCH
  length = 26
  matched = 26

************************************************************
*** Parser Statistics ***
************************************************************
  input string length = 26
  max characters matched = 0
  maximum tree depth = 0
  back track occurrences = 0
  back tracked characters = 0

<table>
<thead>
<tr>
<th>VISITS</th>
<th>MATCH</th>
<th>EMPTY</th>
<th>NOMATCH</th>
<th>ACTIVE</th>
<th>CHILDREN</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNM</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CAT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>REP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PRD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TRG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TBS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TLS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PPT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EOF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TOTAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

***
*** ERROR MESSAGES[0]
***

<none>
```
**Task 1**  
*CSN.1*

Consider the following specification in CSN.1. A data packet consists of a sequence of TLVs, where tag and value fields have a length of 1 octet each. In the value field we store bit strings, with padded bits at the end. The length field gives the length of the value field in octets, while the padding value contains the number of padded bits.

\<\text{data packet}\> ::= \<\text{data item}\>^*; \\
\<\text{octet}\> ::= \text{bit} (8); \\
\<\text{data item}\> ::= \<\text{Type} :\ \text{octet}\> \ <\text{Length} :\ \text{octet}\> \ <\text{Value}\>; \\
\<\text{Value}\> ::= \ <\text{Padding} :\ \text{octet}\> \ <\text{bit}\>^* \ <\text{spare padding}\>; \\
\<\text{spare padding}\> ::= \ <\text{bit}\>^* = 0^*;
Task 2, Example

- Example bit stream with Tag, Length, Padding length and Value fields:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 [octet] |
|---|---|---|---|---|---|---|---|---|---|
| 8...|1|8...|1|8...|1|8...|1|8...|1|8...|1|8...|1|
| TTTTTTTT|LLL|LLLLLL|PPPPP|PPP|VVV|VVV|VVV|TTTTTTT|LLL|LLLLLL|PPPPP|PPP|VVV|VVV|

- Problem: The decoder might read the first TLV and interpret all following bits as spare bits.

- How can we make sure in the specification that the padding is limited and the next data element will be identified?
Task 2

The solution is to use the intersection (&) to force an alignment of the value fields with a general octet string of the length given in the Length field.

\[
\text{& data packet} ::= \text{& date item}^{**};
\text{& octet} ::= \text{bit (8)};
\text{& data item} ::= \text{Type : bit (8)} \& \text{Length : bit (8)}
\{ \text{octet(val(Length))} \& \text{Value} \};
\text{& Value} ::= \text{Padding : octet} \& \text{bit}^{**} \& \text{spare padding} ;
\text{& spare padding} ::= \text{bit}^{**} = 0^{**};
\]
Task 3

Task 3  ASN.1

Specify a data structure in ASN.1 where you can store a person’s name and birthdate and also the names and birthdates of her mother and father and of their grandparents, great-grandparents and so on.

ASN.1

```asn1
Person ::= SEQUENCE {
    name VisibleString;
    birthdate GeneralizedTime;
    father [1] Person OPTIONAL;
    mother [2] Person OPTIONAL }
```

C code by asn1c

```c
typedef struct Person {
    PrintableString_t name;
    GeneralizedTime_t birthdate;
    struct Person *father /* OPTIONAL */;
    struct Person *mother /* OPTIONAL */;
} Person_t;
```
Task 4  Tagging in ASN.1

Tagging is used to disambiguate message field. However, tags are not always necessary. Which tags can be removed from the following specification?

Packet ::= [1] SEQUENCE {
   seqno [2] INTEGER,
   ttl   [3] INTEGER OPTIONAL,
   data  [4] DataType } 

DataType ::= [5] CHOICE {
   plaintext  [6] PrintableString(SIZE(206)),
   ciphertext [7] OCTET STRING(SIZE(206)),
   publickey  [8] BIT STRING(SIZE(16)) }
Task 4

In this example all tags can be removed. Different types result in different tags, which makes explicit tagging unnecessary.

Packet ::= [1] SEQUENCE {
  seqno ::= [2] INTEGER;
  ttl  ::= [3] INTEGER OPTIONAL;
  data ::= [4] DataType; }

DataType ::= [5] CHOICE {
  plaintext ::= [6] PrintableString(SIZE(206));
  ciphertext ::= [7] OCTET STRING(SIZE(206));
  publickey ::= [8] BIT STRING(SIZE(16));
}