ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

Exercise 4

Stefan Rührup

University of Freiburg
Computer Networks and Telematics

Summer 2009

CoNe Freiburg

IIF
INSTITUT FÜR INFORMATIK FREIBURG

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

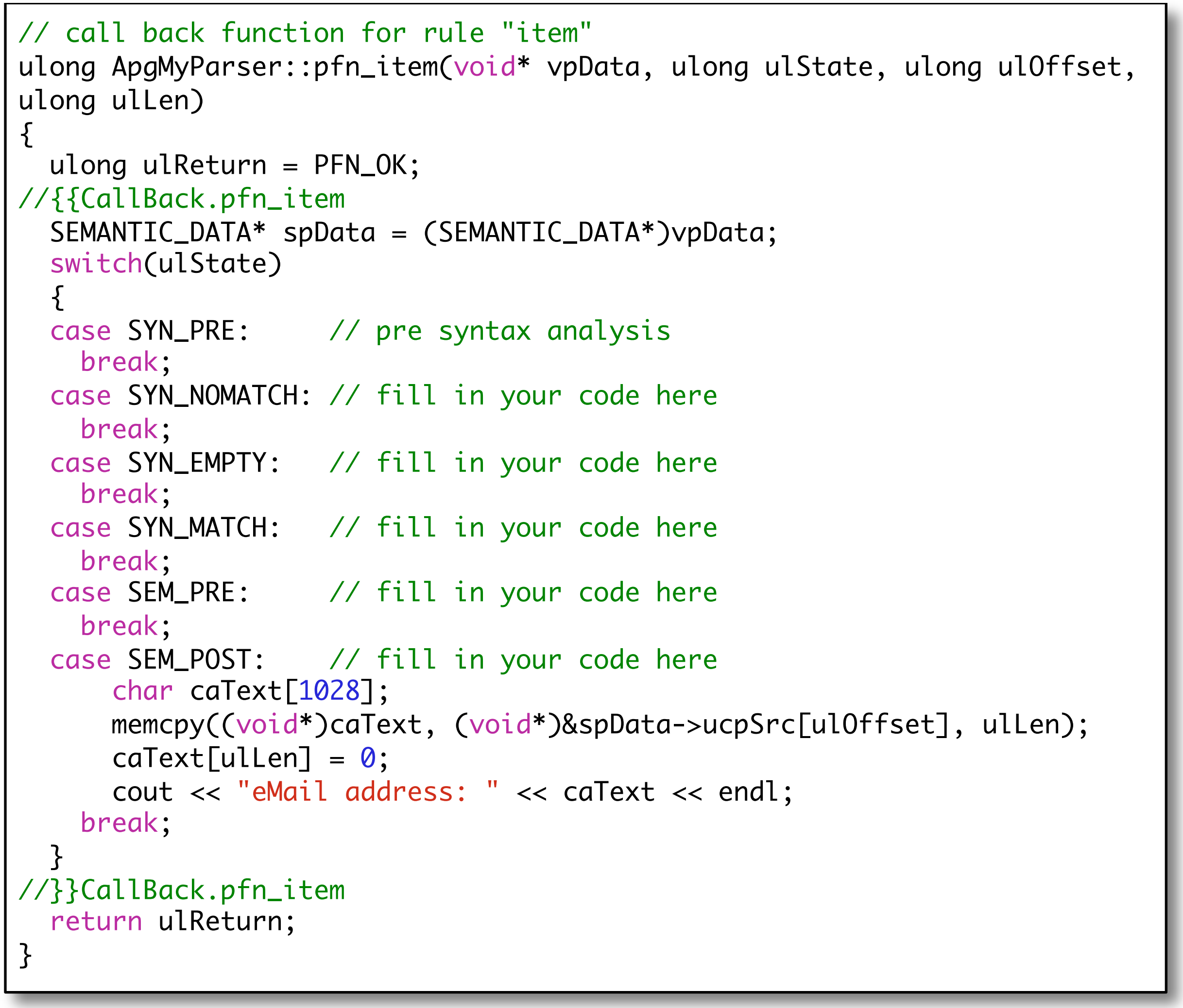
```
[SECTION GRAMMAR]
list = *(item separator) item [eol]
item = string *(period string) at-sign string *(period
string)
string = ( char / digit ) *( char / digit )
char = %x41-5A / %x61-7A / “-”
eol = [%x0d] %x0a
at-sign = "@"
period = "."
separator = ","
digit = %d48-57
[SECTION SYNTAX]
[EXCLUDE]
ALL_RULES

[SECTION SEMANTICS]
[INCLUDE]
list
item
```

optional entries that determine whether call back functions for syntax and semantic analysis should be created

(Note: this grammar doesn’t cover all valid email addresses)

Task 1 - ABNF

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

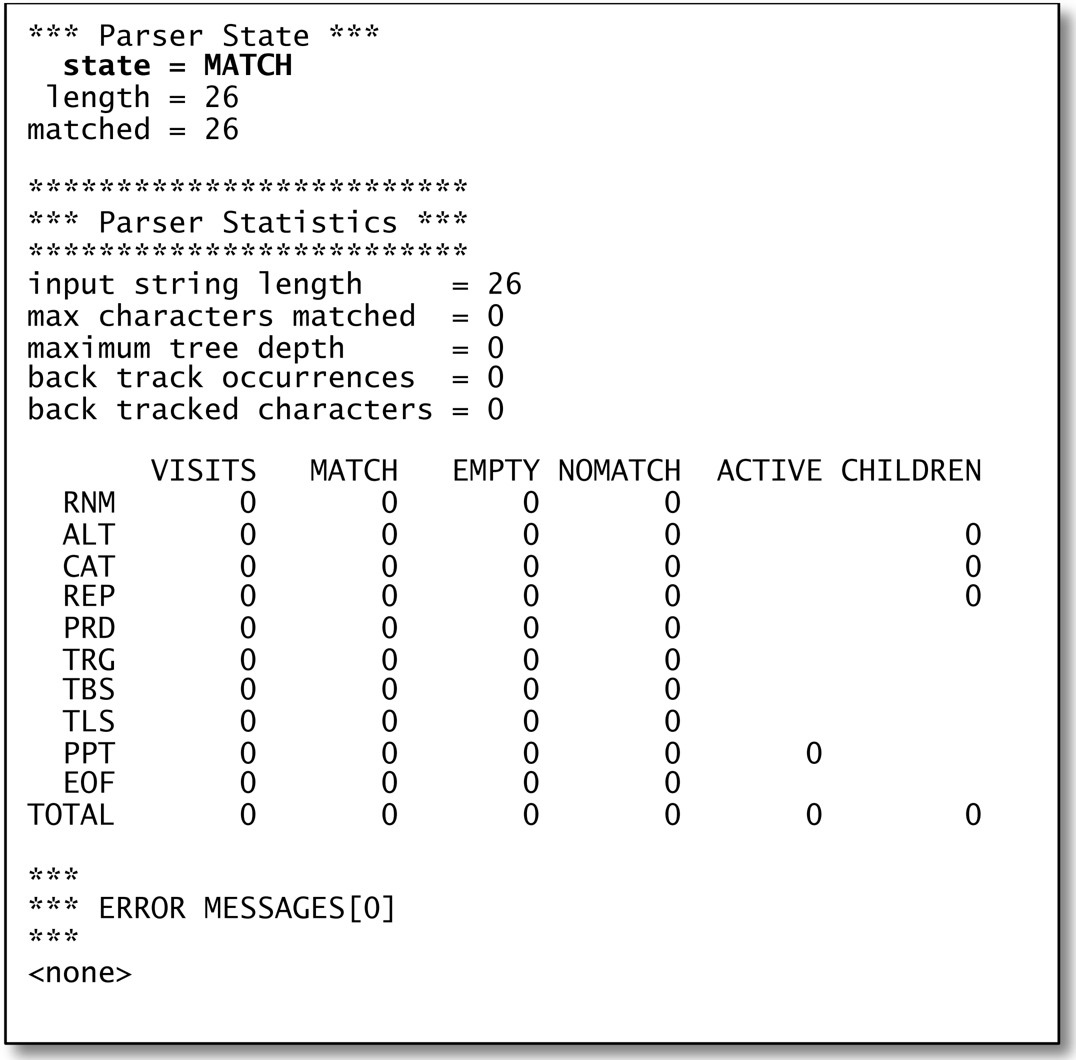


```
// call back function for rule "item"
ulong ApgMyParser::pfn_item(void* vpData, ulong ulState, ulong ulOffset,
ulong ulLen)
{
  ulong ulReturn = PFN_OK;
//{{CallBack.pfn_item
  SEMANTIC_DATA* spData = (SEMANTIC_DATA*)vpData;
  switch(ulState)
  {
  case SYN_PRE:     // pre syntax analysis
    break;
  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;
}
```

Task 1 - ABNF

Try it on your own!

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



```
*** 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

       VISITS   MATCH   EMPTY NOMATCH  ACTIVE CHILDREN
  RNM       0       0       0       0
  ALT       0       0       0       0                0
  CAT       0       0       0       0                0
  REP       0       0       0       0                0
  PRD       0       0       0       0
  TRG       0       0       0       0
  TBS       0       0       0       0
  TLS       0       0       0       0
  PPT       0       0       0       0       0
  EOF       0       0       0       0
TOTAL       0       0       0       0       0        0

***
*** ERROR MESSAGES[0]
***
<none>
```

Task 2

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.

```
< data packet > ::= < date item >**;
< octet > ::= bit (8);
< data item > ::=  <Type : octet > < Length : octet > < Value >;
< Value > ::= <Padding : octet> < bit >** < spare padding > ;
< spare padding > ::= < bit** > = 0**;
```

Task 2, Example

- Example bit stream with **T**ag, **L**ength, **P**adding length and **V**alue 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|8......1|8......1|
TTTTTTTT|LLLLLLLL|PPPPPPPP|VVVVVVVV|VVVVVVVV|TTTTTTTT|LLLLLLLL|PPPPPPPP|VVVVVVVV|
encoded string:
00100016|00000010|00000101|10101010|101xxxxx|00100016|00000001|00000011|10111xxx|
decoded string:
00100016|00000010|00000101|10101010|101xxxxx|xxxxxxxx|xxxxxxxx|xxxxxxxx|xxxxxxxx|
```

- 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.

```
< data packet > ::= < date item >**;
< octet > ::= bit (8);
< data item > ::=  <Type : bit (8) > < Length : bit (8) >
                   { < octet(val(Length)) > & < Value >  };
< Value > ::= <Padding : octet> < bit >** < spare padding > ;
< spare padding > ::= < 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

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

C code by asn1c

```
typedef struct Person {
    PrintableString_t    name;
    GeneralizedTime_t    birthdate;
    struct Person *father /* OPTIONAL */;
    struct Person *mother /* OPTIONAL */;
} Person_t;
```

Task 4

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));
}
```

← the type indicates whether the 2nd INTEGER or the DataType follows

← Items of the choice can be distinguished by their type