Parsing Email Adresses in PHP

By Cal Henderson

2009-05-11: Oh how time flies! There have been several new email RFCs published since this article was written and the state of the art has moved along. A much updated version of the code can be found here, which passes the more modern RFCs. The article below however, is still a good discussion on the topic and is worth reading if you want to know how the code was written.

At some point in your web apps programming life you'll come upon the need to identify something as an email address - usually when validating registration details. You often see a regular expression like this:

```
^[^@]+@[^@]+$
```

Or perhaps it's more fancy cousin:

```
^([_a-z0-9-]+)(\.[_a-z0-9-]+)*@([a-z0-9-]+)(\.[a-z0-9-]+)*(\.[a-z]{2,4})$
```

Which is great. Until someone tries to signup with a valid email address which the regexp doesn't deal with (in the case of the second example). Or an invalid address it thinks is valid (in the case of the first). For instance, this is a valid email address:

```
cal+henderson@iamcalx.com
```

Gah! If only there was some kind of standards document for email address.

Aha! There is ;)

RFC822 (published in 1982) defines, amongst other things, the format for internet text message (email) addresses. You can find the RFC's by googling - there's many many copies of them online. They're a little terse and weirdly formatted, but with a little effort we can seewhat they're getting at. The lexical tokens section of RFC822 is where we want to start. This is my favorite part of the RFCs - none of that wordy stuff, just nice clean BNF-like syntax. Here's the block we need to care about:

```
 3.3.  LEXICAL TOKENS

    The following rules are used to define an underlying lexical
analyzer,  which  feeds  tokens to higher level parsers.  See the
ANSI references, in the Bibliography.

                                          ; (  Octal, Decimal.)
CHAR        =  <any ASCII character>      ; (  0-177,  0.-127.)
ALPHA       =  <any ASCII alphabetic character>
                                          ; (101-132, 65.- 90.)
                                          ; (141-172, 97.-122.)
DIGIT       =  <any ASCII decimal digit>  ; ( 60- 71, 48.- 57.)
CTL         =  <any ASCII control         ; (  0- 37,  0.- 31.)
                character and DEL>        ; (    177,     127.)
CR          =  <ASCII CR, carriage return>; (     15,      13.)
LF          =  <ASCII LF, linefeed>       ; (     12,      10.)
SPACE       =  <ASCII SP, space>          ; (     40,      32.)
HTAB        =  <ASCII HT, horizontal-tab> ; (     11,       9.)
<">         =  <ASCII quote mark>         ; (     42,      34.)
CRLF        =  CR LF
```

```
LWSP-char   =  SPACE / HTAB                ; semantics = SPACE

linear-white-space =  1*([CRLF] LWSP-char)  ; semantics = SPACE
                                            ; CRLF => folding

specials    =  "(" / ")" / "<" / ">" / "@"  ; Must be in quoted-
            /  "," / ";" / ":" / "\" / <">  ;  string, to use
            /  "." / "[" / "]"              ;  within a word.

delimiters  =  specials / linear-white-space / comment

text        =  <any CHAR, including bare    ; => atoms, specials,
               CR & bare LF, but NOT        ;  comments and
               including CRLF>              ;  quoted-strings are
                                            ;  NOT recognized.

atom        =  1*<any CHAR except specials, SPACE and CTLs>

quoted-string = <"> *(qtext/quoted-pair) <">; Regular qtext or
                                            ;   quoted chars.

qtext       =  <any CHAR excepting <">,     ; => may be folded
               "\" & CR, and including
               linear-white-space>

domain-literal =  "[" *(dtext / quoted-pair) "]"

dtext       =  <any CHAR excluding "[",     ; => may be folded
               "]", "\" & CR, & including
               linear-white-space>

comment     =  "(" *(ctext / quoted-pair / comment) ")"

ctext       =  <any CHAR excluding "(",     ; => may be folded
               ")", "\" & CR, & including
               linear-white-space>

quoted-pair =  "\" CHAR                     ; may quote any char

phrase      =  1*word                       ; Sequence of words

word        =  atom / quoted-string
```

That block just defines all the generic rules used in the RFC - what we're also interested in is the formatting for addresses - that's a little bit further down:

```
 6.  ADDRESS SPECIFICATION

6.1.  SYNTAX
```

```
    address     =  mailbox                      ; one addressee
                /  group                        ; named list

    group       =  phrase ":" [#mailbox] ";"

    mailbox     =  addr-spec                    ; simple address
                /  phrase route-addr            ; name & addr-spec

    route-addr  =  ""

    route       =  1#("@" domain) ":"           ; path-relative

    addr-spec   =  local-part "@" domain        ; global address

    local-part  =  word *("." word)             ; uninterpreted
                                                ; case-preserved

    domain      =  sub-domain *("." sub-domain)

    sub-domain  =  domain-ref / domain-literal

    domain-ref  =  atom                         ; symbolic reference
```

When we examine it a little closer, we see that the rule that represents what we think of as an email address is the addr-spec rule. Here's a breakdown of the different rules:

```
 addr-spec               cal@iamcalx.com

phrase route-addr       &quot;cal&quot; <cal@iamcalx.com>
                        &quot;cal&quot; <@foo.com:cal@iamcalx.com>

group                   &quot;a group&quot;:cal@iamcalx.com;
                        &quot;another group&quot;:cal@iamcalx.com,foo@iamcalx.com;
```

So when we ask for an email address for registration purposes or similar, we almost always mean an addr-spec.

We want to build a regular expression for the addr-spec rule. To make it obvious what we're doing, we'll create each rule in a regular php string - here's the very first part:

```
 # addr-spec = local-part "@" domain

$addr_spec = "$local_part\\x40$domain";
```

Note that we're replacing dashes with underscore in the rule names - just because php wont allow dashes in token names. I'm also using hex character references - this is so that we can easily distingish between literals and metacharacters - a literal plus '+' means the match-one-or-more metacharacter, whereas hex entitiy 2b '\\x2b' means a literal plus. Easier than worrying about backwhacking everything.

Now we need to work down the tree, filling out each variable:

```
 # local-part = word *("." word)

$local_part = "$word(\\x2e$word)*";


# domain = sub-domain *("." sub-domain)

$domain = "$sub_domain(\\x2e$sub_domain)*";
```

Onwards we go...

```
 # word = atom / quoted-string

$word = "($atom|$quoted_string)";


# sub-domain = domain-ref / domain-literal

$sub_domain = "($domain_ref|$domain_literal)";


# domain-ref = atom

$domain_ref = $atom;
```

Ok, so, we're defined all of the rules in the address specification that we need - those were easy. But they all depend on the general rules: we need to define atom, quoted_string and domain_literal. The latter two are easy and we'll do those as before:

```
 # quoted-string = <"> *(qtext/quoted-pair) <">

$quoted_string = "\\x22($qtext|$quoted_pair)*\\x22";


# domain-literal =  "[" *(dtext / quoted-pair) "]"

$domain_literal = "\\x5b($dtext|$quoted_pair)*\\x5d";


# quoted-pair =  "\" CHAR

$quoted_pair = '\\x5c[\\x00-\\x7f]';
```

In the last rule, we've skipped defining char and stuck it straight in there. You'd be forgiven for thinking we could just use the dot '.' metacharacter, but look closely: CHAR is defined as any byte between 0x00 and 0x7f, but '.' would match 0x00 to 0xff.

The last three rules we need, atom, dtext and qtext, are the real meat of the matcher - they match the actual sequences of charcters. Let's do atom first. The rule is:

```
     atom        =  1*<any CHAR except specials, SPACE and CTLs>
```

There are two sensible ways we could go about this - create a matching character class ('[abcd]') or a negative character class ('[^efgh]'). To keep things as close to the spec as possible, we'll use a negative class, so we have an item in it for each exception. We'll start off with a class that matches only CHARs:

```
 $char = '[^\\x80-\\xff]';
```

Remember that CHAR itself doesn't include the upper 127 characters - we could write the RFC rule for CHAR like this:

```
 BYTE = 0x00-0xff
HICHAR = 0x80-0xff
CHAR = <BYTE exlcuding HICHAR>
```

Now we need to take away the specials, SPACE and CTLs, so we should figure them out:

```
 # specials = "(" / ")" / "" / "@"
#             /  "," / ";" / ":" / "\" /  #            /  "." / "[" / "]"

$specials = '[\\x28\\x29\\x3c\\x3e\\x40\\x2c\\x3a\\x3b\\x5c\\x22\\x2e\\x5b\\x5d]';


# SPACE = <ASCII SP, space>

$space = '[\\x20]';


# CTL = <any ASCII control character and DEL>

$ctl = '[\\x00-\\x1f\\x7f]';
```

So we can smush all of these together to make the atom rule:

```
 $atom = '[^\\x00-\\x20\\x22\\x28\\x29\\x2c\\x2e\\x3a-\\x3c'.
 '\\x3e\\x40\\x5b-\\x5d\\x7f-\\xff]+';
```

Excellent! We're almost there - we just need to do the same for dtext and qtext, so let's do that.

```
     qtext       =  <any CHAR excepting <">,
                    "\" & CR, and including
                    linear-white-space>

    dtext       =  <any CHAR excluding "[",
                    "]", "\" & CR, & including
                    linear-white-space>

 $qtext = '[^\\x0d\\x22\\x5c\\x80-\\xff]';

$dtext = '[^\\x0d\\x5b-\\x5d\\x80-\\xff]';
```

So we have the whole block - let's take a look. Note: We define them backwards so the sub-rules

are defined when the parent-rules use them.

```
  $qtext = '[^\\x0d\\x22\\x5c\\x80-\\xff]';

 $dtext = '[^\\x0d\\x5b-\\x5d\\x80-\\xff]';

 $atom = '[^\\x00-\\x20\\x22\\x28\\x29\\x2c\\x2e\\x3a-\\x3c'.
  '\\x3e\\x40\\x5b-\\x5d\\x7f-\\xff]+';

 $quoted_pair = '\\x5c[\\x00-\\x7f]';

 $domain_literal = "\\x5b($dtext|$quoted_pair)*\\x5d";

 $quoted_string = "\\x22($qtext|$quoted_pair)*\\x22";

 $domain_ref = $atom;

 $sub_domain = "($domain_ref|$domain_literal)";

 $word = "($atom|$quoted_string)";

 $domain = "$sub_domain(\\x2e$sub_domain)*";

 $local_part = "$word(\\x2e$word)*";

 $addr_spec = "$local_part\\x40$domain";
```

All we're really doing up to this point is building a string, so let's see what the contents of $addr_spec really are:

```
  $addr_spec = '([^\\x00-\\x20\\x22\\x28\\x29\\x2c\\x2e\\x3a-\\x3c'.
   '\\x3e\\x40\\x5b-\\x5d\\x7f-\\xff]+|\\x22([^\\x0d'.
   '\\x22\\x5c\\x80-\\xff]|\\x5c[\\x00-\\x7f])*\\x22)'.
   '(\\x2e([^\\x00-\\x20\\x22\\x28\\x29\\x2c\\x2e'.
   '\\x3a-\\x3c\\x3e\\x40\\x5b-\\x5d\\x7f-\\xff]+|'.
   '\\x22([^\\x0d\\x22\\x5c\\x80-\\xff]|\\x5c\\x00'.
   '-\\x7f)*\\x22))*\\x40([^\\x00-\\x20\\x22\\x28'.
   '\\x29\\x2c\\x2e\\x3a-\\x3c\\x3e\\x40\\x5b-\\x5d'.
   '\\x7f-\\xff]+|\\x5b([^\\x0d\\x5b-\\x5d\\x80-\\xff'.
   ']|\\x5c[\\x00-\\x7f])*\\x5d)(\\x2e([^\\x00-\\x20'.
   '\\x22\\x28\\x29\\x2c\\x2e\\x3a-\\x3c\\x3e\\x40'.
   '\\x5b-\\x5d\\x7f-\\xff]+|\\x5b([^\\x0d\\x5b-'.
   '\\x5d\\x80-\\xff]|\\x5c[\\x00-\\x7f])*\\x5d))*';
```

Not as readable as we might hope, so it's probably best to write it in it's uncomposed form - certainly easier to debug.

Now how do we use this regexp? Very easy:

```
  if (preg_match("!^$addr_spec$!", $email)){
```

```
  echo "It's an email address!";
 }
```

We use the '^' and '$' metacharacters to check we're matching the entire string. We use exclamation marks as regexp delimiters because we know we didn't use any in the regexp itself. So we can pack all of this up into a little function:

```
  function is_valid_email_address($email){

  $qtext = '[^\\x0d\\x22\\x5c\\x80-\\xff]';

  $dtext = '[^\\x0d\\x5b-\\x5d\\x80-\\xff]';

  $atom = '[^\\x00-\\x20\\x22\\x28\\x29\\x2c\\x2e\\x3a-\\x3c'.
   '\\x3e\\x40\\x5b-\\x5d\\x7f-\\xff]+';

  $quoted_pair = '\\x5c[\\x00-\\x7f]';

  $domain_literal = "\\x5b($dtext|$quoted_pair)*\\x5d";

  $quoted_string = "\\x22($qtext|$quoted_pair)*\\x22";

  $domain_ref = $atom;

  $sub_domain = "($domain_ref|$domain_literal)";

  $word = "($atom|$quoted_string)";

  $domain = "$sub_domain(\\x2e$sub_domain)*";

  $local_part = "$word(\\x2e$word)*";

  $addr_spec = "$local_part\\x40$domain";

  return preg_match("!^$addr_spec$!", $email) ? 1 : 0;
 }
```

And there we have it - an RFC822 compliant email address matcher. You can download it here to avoid copying and pasting.

Also: Tim Fletcher has translated the function to ruby and python.
(end of article)