Writing An RFC-Date Parser With Spirit

Peter Simons

simons@computer.org

What this is all about . . .

The Spirit Parser Framework is an incredibly flexible and powerful tool for writing any kind of text processor, parser, or converter. It is also slightly, uh . . . complicated for the beginner to understand. From my own experience when learning how to program with Spirit, I remember well that it is fairly easy to get a parser for a given grammar up and running. But taking the next step --- to actually *use* the parser's results --- is difficult.

This text is intended to server as a tutorial for new Spirit users, who have mastered the basic functionality and are now struggling to get some some real work done. In order to achieve this objective, I'll explain step-by-step the development of a parser that will convert date and time specifications as per RFC822[1] (used in e-mail `Date:` headers, for example) into a tm structure, which can be manipulated with the mktime(3) system library routine.

1. What Makes An RFC822 Date

The grammar defined by RFC822 is seriously crazy. A fact that everybody who has ever tried to parse an e-mail address, for instance, will confirm. If I got an Euro for every RFC822 parser in the world, which is incorrect, I could probably settle Germany's state deficit and still afford a blonde pretty girl-friend with a credit card on my bank account. Unfortunately, RFC822 is out there and chances are small that it's ever going to be fixed, so we might as well deal with it.

According to RFC822, a "date and time specification" is defined by the following grammar (shown in "augmented BNF"):

```
date-time   =  [ day "," ] date time

day         =  "Mon"  | "Tue" |  "Wed"  | "Thu"
            |  "Fri"  | "Sat" |  "Sun"

date        =  1*2DIGIT month 2DIGIT

month       =  "Jan"  |  "Feb" |  "Mar"  |  "Apr"
            |  "May"  |  "Jun" |  "Jul"  |  "Aug"
            |  "Sep"  |  "Oct" |  "Nov"  |  "Dec"

time        =  hour zone

hour        =  2DIGIT ":" 2DIGIT [":" 2DIGIT]

zone        =  "UT"  | "GMT"
            |  "EST" | "EDT"
```

```
|  "CST" | "CDT"
|  "MST" | "MDT"
|  "PST" | "PDT"
|  1ALPHA
| ( ("+" | "-") 4DIGIT )
```

At first sight, this doesn't look so bad. But what makes things complicated is that the RFC allows for the following things to be added (and ignored by the parser) in front of and after *every* token:

- White space,
- linear white space, and
- comments.

"White space" is simply defined as one or several blank or tab characters.

"Linear white space" is defined as any amount of white space, followed by a carriage return and newline character ("`\r\n`"), followed by at least one white space character. This means, that you can add line breaks anywhere in the date as long as the first character on the new line is a blank or a tab. So effectively, these three variants of a date are all equivalent:

```
| 12 Jun 82
|
| 12
|     Jun         82
|
| 12
|  Jun
|  82
```

At last, anywhere where you can add white space, you can add comments as well. A comment is basically any text surrounded by brackets --- minus some specials such as control characters, etc. Thus, the following date is still equivalent to those above:

```
| 12 Jun 82
|
| 12 (Jun)
|     (March) Jun  (fun, isn't it?)       82
```

Now let's see how we can use Spirit to parse this.

2. Our Result Data Structure

Before starting to write any code, it is always a good idea to create the data structures in which the data should be stored and manipulated. In our case, this is fairly obvious, because the the system library provides many useful functions that operate on the tm structure:

```
struct tm
    {
    int tm_sec;         /* seconds */
```

```
    int tm_min;         /* minutes */
    int tm_hour;        /* hours */
    int tm_mday;        /* day of the month */
    int tm_mon;         /* month */
    int tm_year;        /* year */
    int tm_wday;        /* day of the week */
    int tm_yday;        /* day in the year */
    int tm_isdst;       /* daylight saving time */
    };
```

Using mktime(3), for instance, this structure can be converted into a time_t, using strftime(3), it can be formatted into an (almost) arbitrary text format. So this is a good way to represent a date --- and it is easy to fill out by the parser. There is only one problem: The tm structure, as defined by the POSIX standard, does not contain a field for the timezone, the date is in, but this is something that we might need. Hence, we derive our own on class, which adds that field:

```
struct timestamp : public tm
    {
    timestamp() { memset(this, 0, sizeof(*this)); }
    int tzoffset;
    };
```

This definition (along with on appropriate `operator<<`) can be found in the header file `timestamp.hpp`.

Having that out of the way, we can start with the parser.

3. Building The Parser

The most parts of the grammer that are the most obvious to solve are `month` and `day`, which are easily solved by the `spirit::symbols` parser:

```
struct month_parser : spirit::symbols<int>
    {
    month_parser()
        {
        add ("jan", 0)("feb", 1)("mar", 2)("apr", 3)
            ("may", 4)("jun", 5)("jul", 6)("aug", 7)
            ("sep", 8)("oct", 9)("nov", 10)("dec", 11);
        }
    };
const month_parser month_p;

struct wday_parser : spirit::symbols<int>
    {
    wday_parser()
        {
        add ("sun", 0)("mon", 1)("tue", 2)("wed", 3)
            ("thu", 4)("fri", 5)("sat", 6);
        }
    };
const wday_parser wday_p;
```

The parser instances `month_p` and `wday_p` can be used like any other Spirit primitive. What's special about them is that they map the string they matched to the integer value paired to the string. This value is available through the `operator[]` of the parser. We'll see below, how this is used. Don't panic!

The values that we map to the appropriate string are chosen to conform to the specification of the tm structure; you can refer to the manual page for further details.

The next part of the date grammar that we take care of is the `zone` token. This one is a bit trickier: The literals are easy to parse with a `spirit::symbols` parser as well, but the explicit specification of the time zone offset cannot be solved with that. Unless you fancy having a symbol table with 3600 entries, of course. The best solution -- at least the best solution I could come up with -- is to split that token into two tokens:

```
struct timezone_parser : spirit::symbols<int>
    {
    timezone_parser()
        {
        add ("ut", 0)("gmt", 0)
            ("est", -18000)("edt", -14400)
            ("cst", -21600)("cdt", -18000)
            ("mst", -25200)("mdt", -21600)
            ("pst", -28800)("pdt", -25200)
            ("z", 0)("a", -3600)("m", -43200)("n", +3600)("y", +43200);
        }
    };
const timezone_parser timezone_p;

uint_parser<int, 10, 4, 4>  uint4_p;

zone      = ch_p('+')  >>  uint4_p
          | ch_p('-')  >>  uint4_p
          | lexeme_d
            [
                nocase_d
                [
                    timezone_p
                ]
            ]
```

Alright … Breathe, Neo, breathe! This is simpler than it looks. The `timezone_p` parser is straight-forward, no problem. The `uint4_p` parser is just a variation of the normal number parsing primitives: It's declared to parse an unsigned number (in radix 10 notation --- "decimal", that is) with exactly 4 digits. So what will the `zone` rule do? It looks for either a plus or minus character followed by a 4 digit number, or it will try to match the explicit time zone names defined in the symbol parser

The symbol parser has been wrapped into two directives: The `lexeme_d` will switch Spirit into character-parsing mode. This is useful in case `spirit::parse` has been called with a "skipper". Assuming, for instance, we had omitted the `lexeme_d` and had called this rule as follows:

```
spirit::parse(input, zone, spirit::space_p);
```

Then the `timezone_p` parser would match the string "g m t" as well as "gmt"; and this is not what we want.

The second directive deployed is more obvious: `nocase_d` will turn the symbol parser case-insensitive matching-mode. Thus, the inputs “GmT” and “gMt” are equivalest.

One more word concerning the `timezone_p` parser: It may be obvious, but just to be sure let me point out that the numbers returned by the parser when it matches a timezone name is the offset of that timezone to UTC in seconds.

Now we can parse the weekday, the name of the month, and the timezone . . . What’s left is the actual date and time. These is really straight-forward:

```
date      =  uint_p
             >> lexeme_d
                [
                    nocase_d
                    [
                        month_p
                    ]
                ]
             >> (    limit_d(0u, 99u)
                     [
                         uint_p
                     ]
                |    min_limit_d(1900u)
                     [
                         uint_p
                     ]
                );

time      = uint_p
            >> ':'
            >> uint_p
            >> !(    ':'
                     >> uint_p
                );
```

The `date` rule matches the day of months, followed by the name of the month, followed by the year. Here I introduced a slight variation from the RFC’s grammar --- and thereby added one Euro into my state-deficit-and-blond-girl-friend-fund ---: The RFC says that the year is specified by a two digit number. As we know, this is not a good idea, so I took the liberty of allowing a year to be an arbitrarily long number. As you can see above, I added a distinction between the case that a two-digit year has been specified and that a longer year has been specified. This is not strictly necessary to build the parser, but this distinction allows us to handle the two cases separately when we add the code that fills out the tm structure later!

Also, please note the clever use of the `limit_d` and `min_limit_d` directives! This is an extraordinarily smart move, because the parser will not only check the grammar but the semantics of the data as well; we don’t have to worry about getting years back that we cannot express in a tm structure at all. I have to say that these directives are my invention. I hope this makes some people who believe me to be an idiot think!

Actually, though, these are not really my invention, in the strictest sense of the word, because the `limit_d` directive was there already when I found Spirit and I just used it. But the `min_limit_d` directive is my invention.

Even though “invention” does not really mean that I wrote the code or had the idea. To be perfectly honest, Joel had the idea and wrote the code. I just used them.

After he told me: "Peter, why don't you use the `limit_d` directives? This would make your parser simpler ..." and so on.

Alright, maybe the people who think I am an idiot are right.

Anyhow, the `time` rule is pretty obious. Again I violated RFC822 knowingly and withouth remorse: According to the RFC, the seconds part of `time` must be provided, but my parser treats it as optional. (To make matters worse, I hereby solemnly declare that I have the full intention of making the whole `time` part optional later when the assemble the complete parser.!)

Now that all the components are there, we can write down the complete parser fairly easily:

Figure 1. The RFC822 Date And Time Specification Parser

```
struct rfcdate_parser : public spirit::grammar<rfcdate_parser>
    {
    rfcdate_parser()
        {
        }
    template<typename scannerT>
    struct definition
        {
        definition(const rfcdate_parser& self)
            {
            using namespace spirit;

            first =
                (
                    date_time = !(    lexeme_d
                                      [
                                          nocase_d
                                          [
                                              wday_p
                                          ]
                                      ]
                                      >> ',')
                                  )
                                  >> date
                                  >> !time
                                  >> !zone,

                    date      =  uint_p
                                 >> lexeme_d
                                    [
                                        nocase_d
                                        [
                                            month_p
                                        ]
                                    ]
                                 >> (    limit_d(0u, 99u)
                                         [
                                             uint_p
                                         ]
                                    |    min_limit_d(1900u)
```

```
                                        [
                                            uint_p
                                        ]
                                    ),

                    time        = uint_p
                                  >> ':'
                                  >> uint_p
                                  >> !(
                                          ':'
                                          >> uint_p
                                      ),

                    zone        = ch_p('+')  >>  uint4_p
                                | ch_p('-')  >>  uint4_p
                                | lexeme_d
                                  [
                                      nocase_d
                                      [
                                          timezone_p
                                      ]
                                  ]
                );
            }
        const spirit::rule<scannerT>& start() const
            {
            return first;
            }
        spirit::subrule<0> date_time;
        spirit::subrule<1> date;
        spirit::subrule<2> time;
        spirit::subrule<3> zone;
        spirit::uint_parser<int, 10, 4, 4>  uint4_p;
        spirit::rule<scannerT> first;
        };
    };
const rfcdate_parser rfcdate_p;
```

So after all, a `date_time` is made up of an optional weekday, followed by the date (day, month, year), optionally followed by the time (hours, minutes, optionally seconds), optionally followed by the timezone. I guess you can see how easily this parser can be modified to understand other formats as well! Modifying this grammar to deal with ISO, ASN.1, or HTTP header dates is a matter of minutes, not hours.

The only thing left is to actually return some result, and this is worth a chapter of its own.

4. Returning A Result From The Parser

To be written.

Notes

1. http://ietf.org/rfc/rfc0822.txt
2. timestamp.hpp