

Parsing Dates in Lisp

Gene Michael Stover

created 2004 April 18
updated Sunday, 2008 February 3

Copyright © 2004, 2006, 2008 Gene Michael Stover. All rights reserved. Permission to copy, store, & view this document unmodified & in its entirety is granted.

Contents

1	Introduction	2
1.1	To Do	2
2	License	3
3	Obtaining	3
4	Quick Test	3
5	Package CyberTiggyr-Time	4
6	Loading	5
7	FORMAT-TIME	5
7.1	Format String	6
7.2	Standards	6
7.3	Examples	8
7.4	Extending FORMAT-TIME	8
7.4.1	Adding a Format Field	9
7.5	Adding Natural Languages	10
8	PARSE-TIME	10
9	Extending PARSE-TIME	12
10	Alternate Implementations	12
A	Change Log	13

B	Cruft	13
B.1	Modified ISO 8601	14
B.2	Verbose Natural Formats	15
B.3	Terse Formats	16
C	Other File Formats	17

1 Introduction

fixme: Does not parse *yyyymmddTHHMMSS Z* god damn it all. Needs an overhaul. – gene 2006 April 8

fixme: Should also recognize partial dates, such as just the year or the year & the month without the day. – gene 2006 April

I needed a function to parse strings containing dates & return a Common Lisp `UNIVERSAL TIME` . While I was at it, I wrote a function to convert universal times to strings.

If you have a date-&-time string, such as “Saturday, July 10, 2004, 6:45 PM PDT”, you can call my `PARSE-TIME8` function to obtain the equivalent universal time. Here’s an example:

```
lisp> (parse-time "July 10, 2004, 6:45 PM")
3298499100
lisp> (decode-universal-time *)
0          ; second
45         ; minute
18         ; hour
10         ; day
7          ; month
2004      ; year
5
T          ; Daylight Stupid Time ???
8         ; time zone
lisp>
```

There is also a `FORMAT-TIME7` function which converts universal times to text. It is the reverse of `PARSE-TIME`, & it’s analogous to the Common Lisp function `FORMAT` . Here is an example of `FORMAT-TIME` in use:

```
lisp> (format-time nil "%H:%M on %A, %d %B" (get-universal-time))
"20:55 on Tuesday, 08 June"
lisp>
```

1.1 To Do

1. `FORMAT-TIME` should be able to print for a specific time zone other than the local time zone.

2. FORMAT-TIME should have a format field which prints the symbolic time zone name. Maybe that could be the %z (lower-case z) field.
3. If FORMAT-TIME has a symbolic time zone name field, *FORMAT-TIME-CEE* should be updated to use it.
4. Maybe the format field should always be a Lisp list (or other Lisp datum), not a string.
5. Maybe the recognizer functions could be created at load-time with macros. Would this simplify the code or reduce its size?

2 License

The source code I describe in this article is released in accordance with the GNU Lesser General Public License[Fou].

This article is not covered by the Gnu Lesser General Public License. It is covered by its own copyright notice which is at the beginning of the article.

3 Obtaining

The source code I describe here is in these files¹

- cybertiggyr-time.asd
- time.lisp
- demo.lisp. Contains a demo function.
- loadall.lisp. LOAD this file to load time.lisp, demo.lisp, & test.lisp. It's for convenient testing.
- test.lisp. Contains the test system. To run all the tests, evaluate (CHECK).

Each of these source files is licensed according to the terms of the GNU Lesser General Public License[Fou].

4 Quick Test

To quickly test everything on your Lisp, download all the files, “cd” to that directory, run Lisp, & do this:

¹If you are reading this on paper, the URLs for those files are <http://cybertiggyr.com/gene/pdl/cybertiggyr-time.asd>, <http://cybertiggyr.com/gene/pdl/time.lisp>, <http://cybertiggyr.com/gene/pdl/demo.lisp>, <http://cybertiggyr.com/gene/pdl/loadall.lisp>, & <http://cybertiggyr.com/gene/pdl/test.lisp>.

input	output
2004-09-26T13:22:51 -7	2004-09-26T13:22:51 -7
now	2004-09-26T13:22:54 -7
today	2004-09-26T05:00:00 -7
2004 05 30	2004-05-30T12:00:00 -7
2004-05-30	2004-05-30T12:00:00 -7
2004 May 30	2004-05-30T12:00:00 -7
1980-jun-1T12:30:00 gmt	1980-06-01T05:30:00 -7
1980-jun-1T12:30:00 est	1980-06-01T10:30:00 -7
1980-jun-1T12:30:00 pst	1980-06-01T13:30:00 -7
2000-jan-01T05:59:59+00:00	1999-12-31T21:59:59 -8
2000-01-01T00:59:59 est	1999-12-31T21:59:59 -8
1999-12-31T23:59:59 -6	1999-12-31T21:59:59 -8
1999-12-31T22:29:59-07:30	1999-12-31T21:59:59 -8
1999-12-31T21:59:59 -8	1999-12-31T21:59:59 -8
1999-12-31T21:59:59-8:00	1999-12-31T21:59:59 -8
Mar 4, 05	2005-03-04T12:00:00 -8
3/4/05	2005-03-04T12:00:00 -8
19951025	1995-10-25T12:00:00 -7

Table 1: The output from DEMO

```

lisp> (load "loadall.lisp")
;; see commands about loading some files
lisp> (check)
;; see list of test program names
T

```

If CHECK returns NIL or dumps you into the debugger, my time formatter or parser doesn't work on your Lisp. Or maybe I broke it when making improvements. Let me know, & I'll see what I can do.

To see a table of some time strings in various formats & their parsed results (converted back to strings to you can read them), do this:

```
lisp> (demo)
```

Table 5 shows the output from DEMO.

5 Package CyberTiggyr-Time

The `time.lisp` file defines a package called CYBERTIGGYR-TIME & exports two functions & a bunch of data from it. The two functions are `FORMAT-TIME` (Section 7) & `PARSE-TIME` (Section 8).

Use the `FORMAT-TIME` function to convert universal times to strings. There are also some global variables which might be useful when formatting times; they are described in Section 7.2.

To parse times, you just need the `PARSE-TIME` function. You might `IMPORT` it or call it by its full name: `CYBERTIGGYR-TIME:PARSE-TIME`.

6 Loading

fixme this entire section.

what it says now:

```
lisp> (asdf:operate 'asdf:load-op "cybertiggyr-time")
```

what it used to say:

When loading `time.lisp`, I recommend the symbolic pathname string `"CL-LIBRARY:COM;CYBERTIGGYR;TIME;TIME.LISP"`. So you'd load it like this:

```
lisp> (load "CL-LIBRARY:COM;CYBERTIGGYR;TIME;TIME.LISP")
```

Or if you, like me, prefer explicitly translated logical pathnames, you'd do this:

```
lisp> (load
      (translate-logical-pathname
       "CL-LIBRARY:COM;CYBERTIGGYR;TIME;TIME.LISP"))
```

7 FORMAT-TIME

`FORMAT-TIME` is analogous to the Common Lisp function `FORMAT`. Its declaration looks like this:

```
;; in-package "CYBERTIGGYR-TIME"
(defun format-time (strm fmt
                  &optional
                  (ut (get-universal-time))
                  (zone *format-time-default-zone*)
                  (language *format-time-default-language*))
  ...)
```

As with the `FORMAT` function, `STRM` identifies the destination of the function's output. It may be:

- an output stream,
- `T`, which is short-hand for `*STANDARD-OUTPUT*`, or
- `NIL`, in which case `FORMAT-TIME` will return a new string.

The `FMT` argument of `FORMAT-TIME` will most often be a string, but it may also be a list. It contains format fields & literals. It is described in detail in Section 7.1.

The `UT` argument should be a Lisp Universal Time or absent. If it is absent, it defaults to the current time.

The `ZONE` argument defaults to the local time zone. In the future, it will allow you to specify the time zone for the output string, but for now, it's ignored.

The `LANGUAGE` argument specifies the natural language in which some output fields are converted. For example, if the language is English, the day of week is Monday, & the `FMT` argument contains the string “%A”, the output will contain “Monday”, but if the language is French, the output will contain “Lundi”. You can add your own languages, & I'll give some examples later.

For convenience & standardization, I've created some format string constants (Section ??) which can be used for the `FMT` argument of `FORMAT-TIME`.

7.1 Format String

`FORMAT-TIME` (7) uses a *format string* argument to tell the form of the output string. A format string contains *format fields & literals*.

A format field consists of a percent character (%) followed by another character. The second character identifies the type of the format field. It is case-sensitive, so “%A” is distinct from “%a”.

Literals are anything other than format fields. They are case-sensitive.

Table 2 gives brief descriptions of the format fields for both `FORMAT-TIME` & `PARSE-TIME`. The format fields resemble those from the Standard C function `strftime`.²

The format strings for `FORMAT-TIME` resemble those for the Standard C function `strftime`. I haven't implemented all the format fields from `strftime`. Some of the missing ones are %j (Julian day) & %c. A Julian format field might be useful at times, but I think a %c is unnecessary. Instead of %c, use one of the !!!.

`FORMAT-TIME` has a strict interpretation of the format fields. If the second column Table 2 says that a field is two digits, then `FORMAT-TIME` will produce exactly two digits, with a leading zero if necessary. Also, literals (the parts of the format string which are not format fields) are literal.

7.2 Standards

For convenience & standardization³, I put some useful format strings in global variables. Table 3 shows those standardized format symbols & their values. All those symbols are within package `CYBERTIGGYR-TIME` & exported by it.

²`strftime` is described in the unix “man pages” (type “man strftime”). It is also described in [Pla92] & other sources.

³For definitions of “standard” which approximate “Gene hopes people will agree with him that it'd be nice if everyone did it”.

field	meaning
%A	full weekday name
%a	abbreviated weekday name
%B	full month name
%b	abbreviated month name
%d	two-digit day of month
%H	two-digit 24-hour
%I	two-digit 12-hour
%M	two-digit minute
%m	two-digit month number
%p	AM or PM
%S	two-digit seconds
%Y	four-digit year
%Z	numeric time zone
%%	literal %

Table 2: Brief descriptions of the format fields for FORMAT-TIME

symbol	string value
FORMAT-TIME-ISO8601-LONG	%Y-%m-%dT%H:%M:%S %Z
FORMAT-TIME-ISO8601-SHORT	%Y%m%dT%H%M%S %Z
FORMAT-TIME-DATE	%d %b %Y
FORMAT-TIME-TIME	%H:%M %Z
FORMAT-TIME-FULL	%A, %Y %B %d, %H:%M %Z
FORMAT-TIME-CEE	%a %b %d %H:%M:%S %Z %Y

Table 3: Global variables exported by CYBERTIGGYR-TIME that can help standardize format strings

fmt	format-time	comment
"%Y-%m-%dT%H:%M:%S %Z"	2036-05-08T23:28:16 -7	ISO 8601
"%b, %d %b %Y"	Thu, 08 May 2036	short date, like for a cheque
"%A, %d %B %Y"	Thursday, 08 May 2036	full date
"%b %d, %Y"	May 08, 2036	silly American date
"%H:%M GMT%Z"	23:28 GMT-7	time with semi-readable zone
'("%H" "." "%M" " GMT" "%Z")	23:28 GMT-7	ditto, as a list
"%I:%M %p"	11:28 PM	time on 12-hour clock

Table 4: Examples of `FORMAT-TIME` for different format values on the Universal Time of 4302916096

My personal favorite is `*FORMAT-TIME-ISO8601-LONG*`. It is an international standard & unambiguous. If you sort the resulting strings, you get a chronological list. The format can be read by humans without pain, though it's not very pleasing to the eye. My main complaint with it is that it numbers the month; I'd prefer an abbreviated month name.

If you need a format that is more pleasing to the human eye, or if you just don't like the `*format-time-iso8601-long*`, I recommend the `*format-time-full*` format.

The `*FORMAT-TIME-CEE*` format is similar to the `%c` format of Standard C's `strftime` function. It is not identical, though. The main difference is that time zones are numbers, not symbolic as with `strftime`'s `%c`. Also, `*FORMAT-TIME-CEE*` uses strictly two-digit numbers for the day of month, whereas `%c` might use single-digit numbers where possible or might pad the day-of-month with a space on the left when appropriate. `*FORMAT-TIME-CEE*` is pretty close to `strftime`'s `%c`, though.

7.3 Examples

The Lisp Universal Time for Wednesday, 2036 May 8, 23:28:16 daylight savings time, US Pacific zone is 4302916096. When daylight savings time is in effect, US Pacific time is seven hours behind Greenwich Mean time. Table 4 shows examples of the output of evaluating `(FORMAT-TIME NIL FMT 4302916096)` for various values of `FMT` when English is the default language.

A special example from Table 4 is the penultimate one, which uses a list of strings as the `FMT` argument. When the `FMT` argument is a single string, `FORMAT-TIME` parses it into a list of tokens like in that one example, then it processes each token. The tokens may be strings (as shown), symbols, or whatever. So if you can stomach the horrible syntax, you get more flexibility & run-time efficiency by using lists instead of strings.

7.4 Extending `FORMAT-TIME`

As of Sunday, 26 September 2004, this section might be inaccurate. I have

recently updated the documentation, & I have not double-checked this section. So beware.

FORMAT-TIME converts the format string to a list of format fields & literal fields, though when it splits the format string into those fields, FORMAT-TIME doesn't make note of which fields are literal & which are format fields. Then it iterates over the fields. It tries to find each field in a table called *FORMAT-TIME-FNS*, which is a hash table whose keys are format fields such as "%Y".

7.4.1 Adding a Format Field

To add new format fields, create a function to do the work for that field. Choose an unused format field string. Then insert the new field string & the new function into the *FORMAT-TIME-FNS* table. (Optional: If your new format field is probably of use to others, send it to me & I'll include it in the next release of this library.)

Let's do an example.

Let's say you want to implement the two-digit year format field for FORMAT-TIME. The format field string will be %y.

Now we need to write the function which actually does the formatting for %y.

When FORMAT-TIME calls a function for a field, it provides three arguments. They are:

bro A BROKEN-TIME structure, which is defined near the beginning of the `time.lisp` file.

language Identifies the natural language for the output string.

strm The character output stream to which the function should send the string it creates.

Because the BROKEN-TIME contains the time's components "broken" out of the universal time, the job of the new function is simple. We just need to create a string from the last two digits of the year. Here's how we could do that:

```
(defun format-time-evil-year (bro language strm)
  (declare (ignore language))
  (format strm "~2,'0D" (mod (broken-time-yy bro) 100))
  ;; The return value doesn't matter. In such cases,
  ;; I like to return the function's name. It's okay
  ;; if you want to return something else.
  'format-time-evil-year)
```

The final step is to insert the new function into the *FORMAT-TIME-FNS* table. Here's how:

```
(setf (gethash "%y" *format-time-fns*)
      #'format-time-evil-year)
```

You will also want to write one or more test programs for your new format field. Add them as functions to `test.lisp`.

7.5 Adding Natural Languages

You can add the names of months & weekdays from other languages, too. To do that, add them to the `*FORMAT-TIME-MONTHS*` or `*FORMAT-TIME-WEEKDAYS*` tables, whichever table is appropriate.

Each key in those tables is a list of two elements. The first element is the number of the item, such as 1 for January or 6 for Sunday. The second element of a key is the natural language. I recommend symbols from the keywords package, such as `:ENGLISH`, `:JIVE`, or `:WELSH`.

Each value is a list of two strings. The first item in a value list is the full name of the month or weekday. The second is the abbreviated name.

If you add any other languages & wanted to send them to me so I could include them in the next release, that'd be peachy.

8 PARSE-TIME

Function `PARSE-TIME` scans a date-&-time encoded in a human-readable string to determine the equivalent `UNIVERSAL TIME`. `PARSE-TIME` returns the universal time when it can determine it; otherwise, `PARSE-TIME` returns `NIL`.

`PARSE-TIME` understands a variety of formats, & it can fill-in items that are missing. It also understands some convenience words (*today* & *now*). Here are some examples:

```
lisp> (parse-time "2004-apr-4 19:11")
3291329460
lisp> (parse-time "4 april 2004")
3291260400
lisp> (parse-time "today")
3295684800
lisp> (parse-time "now")
3295742590
```

`PARSE-TIME` is declared like this:

```
(defun parse-time (str &optional
                  (recognizers *default-recognizers*))
  ...)
```

The *str* argument is a string. It contains the date-&-time to parse.

The optional *recognizers* argument is a list of function which know how to parse specific formats. If you don't supply a list of recognizers, you get the default list. I suspect that the default list of recognizers will be most common under normal use.

input	output
2004-09-26T13:22:51 -7	2004-09-26T13:22:51 -7
now	2004-09-26T13:22:54 -7
today	2004-09-26T05:00:00 -7
2004 05 30	2004-05-30T12:00:00 -7
2004-05-30	2004-05-30T12:00:00 -7
2004 May 30	2004-05-30T12:00:00 -7
1980-jun-1T12:30:00 gmt	1980-06-01T05:30:00 -7
1980-jun-1T12:30:00 est	1980-06-01T10:30:00 -7
1980-jun-1T12:30:00 pst	1980-06-01T13:30:00 -7
2000-jan-01T05:59:59+00:00	1999-12-31T21:59:59 -8
2000-01-01T00:59:59 est	1999-12-31T21:59:59 -8
1999-12-31T23:59:59 -6	1999-12-31T21:59:59 -8
1999-12-31T22:29:59-07:30	1999-12-31T21:59:59 -8
1999-12-31T21:59:59 -8	1999-12-31T21:59:59 -8
1999-12-31T21:59:59-8:00	1999-12-31T21:59:59 -8
Mar 4, 05	2005-03-04T12:00:00 -8
3/4/05	2005-03-04T12:00:00 -8
19951025	1995-10-25T12:00:00 -7

Table 5: Examples of some formats understood by PARSE-TIME

If PARSE-TIME can recognize the date-&-time in the string, it converts them into a UNIVERSAL TIME & returns it. If PARSE-TIME can't recognize the string, it returns NIL.

Table 5 shows examples of some strings that were fed to PARSE-TIME. The first column shows the strings that were given to PARSE-TIME. The second column shows the output of using FORMAT-TIME to convert the resulting universal time back to a readable string.

I produced Table 5 by running the DEMO function from demo.lisp. The time zone on my computer when I ran it was U.S. Pacific Daylight Stupid time, which is 7 hours west of Greenwich Mean Time (GMT). It's also 7 hours behind GMT.

Notice the second & third rows in Table 5. PARSE-TIME interprets a string of "now" as the current time.

PARSE-TIME interprets a string of "today" as noon *in the GMT time zone* on the current day. So if you feed "today" to PARSE-TIME on 2004 July 4, 13:00 in London, you'll get a universal time corresponding to 2004 July 4, 12:00. If someone in Seattle feeds "today" to PARSE-TIME at exactly the same time, they'll get the same universal time, which will be 2004 July 4, 5:00 PDT.

So why does "today" inject noon in the GMT instead of using the current time as "now" does? I wanted "today" to evaluate to the same universal time regardless of the time zone in which it was parsed. So parsing "today" on a particular day gets you the same universal time anywhere on the world. That's why it uses a particular time of day instead of the current time of day. For that particular time of day, I choose noon in London because it is unlikely

(impossible?) to find a time zone that is more than 11 hours distant from London. So when it is noon in London, all other time zones will show different times of the day, but they will all show the same day.

9 Extending PARSE-TIME

I wrote a function similar to PARSE-TIME for C years ago. I used *yacc* for part of the work. If I remember correctly, I never did work out a single grammar that could describe all these formats. Maybe I didn't try hard enough; nevertheless, I don't want to do it that way for Lisp.

My PARSE-TIME function uses a list of *recognizers*.⁴ Each recognizer⁵ is a function that can extract the date & time from one or more formats, but it never incorrectly extracts information from another format. It returns a UNIVERSAL TIME or a BROKEN-TIME structure.

It's important that a recognizer does not accidentally or incorrectly extract & return information from a format that it doesn't understand. As long as all recognizers know when they don't understand a format, the order in which PARSE-TIME tries the recognizers doesn't matter & maintenance will be easier.

For example, a recognizer for the *yyyy-mm-dd* format, which contains the year, the month, the day, but no hours, minutes, seconds, or time zone, should extract & return information from "2004-02-01", but it must be sure to return NIL for "2004-02-01 9:21".

PARSE-TIME tries each recognizer in its list of recognizes until one of them understands the parse string.

On the one hand, it's pretty grotty⁶ It's brute-force code, & I did a lot of it by copy-&-paste programming.⁷

On the other hand, the implementation is extensible. If you want PARSE-TIME to understand a new format, write a recognizer for it & push it onto *DEFAULT-RECOGNIZERS*. If you can write a more elegant recognizer, you can push it onto *DEFAULT-RECOGNIZERS*, too.

10 Alternate Implementations

I originally wanted to use a more intelligent system, maybe one that analyzed each term in the parse string & figured out what each term was. For example, in the parse string were "May 5 2004", the "May" is certainly a month, & the 2004 is probably a year, so the "5" is probably a day. In "5/5/4", the slashes suggest that the string is in Stupid American Date Format, so the first 5 is May, the second 5 is the day, & the 4 suggests a year *y* that is closest to the

⁴Recognizer is a common term from formal languages & compiler-writing. I chose the term to remind me of those uses.

⁵Recognizer is also the name of a type of baddie in the movie *Tron*.

⁶Grotty means ugly, at least it does in this article. "Grotty" was first used in the movie *A Hard Days Night*, & if I remember correctly, it meant good.

⁷I am embarrassed about that.

current year & $y \bmod 100 = 4$. If this *reasoning recognizer* produced more than one parse for the string, it could attach a confidence value to each estimate, & PARSE-TIME could return the result with the highest confidence value.

That would be cool, but it would take a lot of programming. It would take research. It also sounds like it would be an all-or-nothing type of algorithm. Until I got the entire thing working, none of it would work. The approach I've chosen allows recognizers to be added one at a time, so it was easy to get something working, & new recognizers can be created, debugged, & added in isolation.

Within the architecture I've chosen, It might be possible to use a pattern-matching library to write the recognizers. That might remove the need to create a named function for each recognizer. If you look at all the recognizers in time.lisp, you'll see that I have some long, hard-to-type names for some of those recognizers, & I suspect the situation will get worse as I add more recognizers. If I had a macro that compiled patterns to functions, I might be able to create anonymous recognizer functions like this:

```
;; Pattern-based recognizer that matches strings
;; of the form "Friday, 13 July 2029".
(push
  (defrecognizer
    ((?isWeekday ?dow) ", " (?isDay ?day) (?isMonth ?mon) (?isYear ?year)))
  *default-recognizers*)
```

I don't suppose such pattern-based recognizers would be more efficient than the hand-crafted ones I've used, & PARSE-TIME would probably need as many of them as it does the hand-crafted ones, but these pattern-based ones might be easier to maintain because their definitions would be their documentation. They might be easier to debug, too, if the mythical DEFRECOGNIZER macro required only the one parameter I have shown here.

A suitable pattern-matching library might be the one described in the section "A Pattern Matching Tool" from Peter Norvig's *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* ([Nor92,]).

A Change Log

2008-Feb-03 Added the ASDF file, cybertiggyr-time.asd.

2004-Sep-26 PARSE-TIME now recognizes strings of the form *yyyymmdd*, in which every character is a digit. For example, "19991231".

B Cruft

This is old stuff I wrote for this article. I might remove it entirely some day. *1 June 2004*

B.1 Modified ISO 8601

ISO 8601:1988 is a standard for specifying dates, times, & periods of time. ?? is an unofficial but useful description of that standard.

PARSE-TIME should understand a restricted version of the ISO 8601 format. The full format is:

year – month – dayThour : minute : secondampmzone
where

- the hyphens & colons are literals,
- *year* is four digits,
- *month* two digits or an abbreviated month name,
- *day* is the two-digit day of the month,
- *T* is a literal letter *T*, in upper- or lower-case,
- *hour* is two digits, $0 \leq hour < 24$,
- *minute* is one or two digits, $0 \leq minute < 60$,
- *second* is one or two digits, $0 \leq second < 60$,
- *ampm* is the literal *AM* (case-insensitive), the literal *PM* (case-insensitive), or missing.
- *zone* is the time zone or is missing.

The hyphens & colons must all be present or must all be missing. The *seconds* & its preceeding colon are optional, but if the *seconds* are missing, their preceeding colon must be missing, too.

If the *T* is missing, the hyphens & colons must be missing & the seconds must be present.

If *ampm* is present, then $1 \leq hour \leq 12$. Otherwise, $0 \leq hour < 24$.

If *zone* is missing, PARSE-TIME assumes GMT.

I think those rules boil down to these cases:

1. All components present, giving *year – month – dayThour : minute : second*,
2. Missing *seconds*, giving *year – month – dayThour : minute*,
3. hyphens & colons missing, giving *yearmonthdayThourminutesecond* (all components smashed together, with no spaces between them),
4. *T* (& therefore hyphens & colons) missing, giving *yearmonthdayhourminutesecond*.

Table 6 shows examples of this modified ISO 8601 format.

If the AMs, PMs, & daylight savings times in some of those examples are confusing, consider it a reminder of how stupid a 12-hour clock & daylight savings time are. I'm not sure time zones haven't outlived their utility, for that matter.

2004-04-18T23:59:59	the last second of 18 April 2004 GMT
2004-apr-18T23:59:59	ditto
2004-04-18T11:59:59 pm	ditto
20040418T235959	ditto, without the hyphens & colons
20040418235959	ditto, as terse as possible
2004apr18235959	another terse version
2004-apr-18T12:00:00	noon on 18 April 2004 GMT
2004-apr-18T12:00	ditto
2004-apr-18T12:00 pm	ditto
2004-apr-18T12:00 am	first second of 18 April 2004
2004-apr-18T12:00 am pdt	equiv. to 2004-04-18T07:00 GMT
2004-nov-18T12:00 am pst	equiv. to 2004-11-18T08:00 GMT
2004-nov-18T12:00 am +8	equiv. to 2004-11-18T08:00 GMT
2004-nov-18T12:00 am +7:15	equiv. to 2004-11-18T07:15 GMT
2004-nov-18T12:00 +7:15	equiv. to 2004-11-18T19:15 GMT

Table 6: Examples of my Modified ISO 8601 date-&-time format

B.2 Verbose Natural Formats

Verbose natural formats are those written in long-hand. Here are the rules:

1. In particular, the months have complete names or abbreviated names, but the months are never numbers.
2. Day-of-week may be present. It will probably be ignored, but there is a chance it could serve as a hint.
3. Time zone may be present. If it is absent, local time zone is assumed if it is known; otherwise, GMT is assumed to be the local time zone.
4. *AM* or *PM* marker may be present. Otherwise, a 24-hour clock is assumed.
5. The literal “noon” may be used in place of the time.
6. The literal “midnight” may be used in place of the time. Midnight is considered the first second of the day.
7. The literal “o’clock” may be present. If so, it invokes a sloppy heuristic that tries to automagically determine AM or PM. If you don’t like the sloppy heuristic, that’s what you get for describing the time in an ambiguous way.
8. Components of the date-&-time may occur in any order, but certain orders are recommended while others should be avoided. Commas may serve as hints.

18 April 2004	2004-Apr-18T12:00 local time zone
April 18, 2004	ditto
midnight 18 April 2004	2004-Apr-18T00:00 local time zone
3 o'clock apr 18, 2004	2004-Apr-18T15:00 local time zone
10 o'clock apr 18, 2004	2004-Apr-18T10:00 local time zone
10 o'clock Friday 18 apr	see note A
18 apr 3	2003-Apr-18T12:00 local time zone
apr 18, 3	ditto
apr 18 3	ditto
3 apr 18	2018-Apr-03T12:00 local time zone
3 apr 2018 3:30 pm	2018-Apr-03T15:30 local time zone

Table 7: Example date-&-times in the “natural” formats

18-Apr-2004	2004-Apr-18T12:00 local time zone
18-Apr-04	ditto
18-04-04	ditto
04/04/18	ditto
Apr/04/18	ditto
18/04/04	Error! No 18 th month
18/Apr/2004 3 pm	2004-Apr-18T15:00 local time zone

Table 8: Example date-&-times in the “natural” formats

Table 7 shows examples.

For the example marked “see note A”, PARSE-TIME should assume the current year. It ignores the “Friday” because “18 apr” has enough information. So “10 o'clock Friday 18 apr”, if parsed during the year 2004 in the timezone of the United States’s Pacific Coast, is equivalent to 2004-Apr-18T17:00.

B.3 Terse Formats

The terse formats often use numbers for months & only two digits for years. The month may be numbered, named in full, or abbreviated, but there will be two other numbers. One of them is the day; it must be one or two digits. The other is the year; it must be two or four digits.

I think these formats stink because people so often number the months & also use two-digit years. This creates ambiguity. When this occurs, the separators determine the order. There are always two separators. Both must be hyphens (-) or both must be slashes (/). Hyphens indicate the European order of day, month, year. Slashes indicate the stupid American format of month, day, year.

Times may be present in the same way as for the verbose natural formats.

Table 8 shows examples

I dislike numbered months & two-digit years because of the potential for misunderstanding. In addition, I dislike the American convention of month/day/year. That's like saying 123 is three hundred twelve.

C Other File Formats

- This document is available in multi-file HTML format at <http://cybertiggyr.com/gene/pdl/>.
- This document is available in Pointless Document Format at <http://cybertiggyr.com/gene/pdl/pdl.pdf>

References

- [Fou] Free Software Foundation. Lesser general public license. world wide web. <http://www.gnu.org/licenses/licenses.html#LGPL>.
- [Nor92] Peter Norvig. *Paradigms of Artificial Intelligence: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, 1992. ISBN 1-55860-191-0.
- [Pla92] P. J. Plauger. *The Standard C Library*. Prentice Hall, 1992. ISBN 0-13-131509-9.