How-To: Use of the FV Time Manager on Windows, Linux and other platforms through its command line interface

Formal Vindications S.L.*

September, 2022

*Project funded by the Spanish Ministry of Science, Innovation and Universities, the State Agency for Research and the European Regional Development Fund (ERDF).

Contents

1 Preface

1.1 Context

The FV Time Library is a Coq-verified implementation of the UTC standard with some extra utilities to make it more usable for both critical and regular programming. Currently, this library is only available in the OCaml language[1]. In order to make the power of the Time Library available from almost any computer environment, we have developed the FV Time Manager, which is a standalone executable (available for Linux and Windows) that allows us to interact with the Time Library from the command line. In this document, we give instructions on how to install and use the Time Manager.

1.2 File structure

The files directly extracted from Coq to OCaml, and thus verified, are `FVTM.ml` (the code itself) and `FVTM.mli` (the headers). They can be used directly as OCaml libraries, but in order to use them from other programming languages or from the command line, it is necessary to interface with some code, which can be called *command line interface* or simply *wrapper* (which is non-verified). Any user can write their own wrapper, but we provide one. Below we describe the files and what each of them does.

- `FVTM.ml`, `FVTM.mli`: The code and headers extracted directly from Coq.
- `uint63.ml`, `uint63.mli`: The code and headers provided by Coq for the extraction to OCaml's primitive machine integers.
- `timemanager.ml`: The wrapper or command line interface, non-verified.

In Section `Usage` we present the documentation of the functions offered by `timemanager.ml`.

1.2.1 About the wrapper code

The file `timemanager.ml` contains only data handling, in particular it takes care of:

- the input/output (converting from datatypes to `string` and viceversa, i.e., *serialization* and *deserialization*);
- converting the Coq type that expresses errors (due to invalid inputs) to OCaml exceptions that throw understandable error messages.

As an observation, recall that the compilation process is not itself verified, and neither is the file `timemanager.ml`. However, this file does not introduce a big probability of bugs, due to its simplicity.

Any user can write their own wrapper by looking at the Coq documentation of the extracted functions. We provide this wrapper just for the convenience of the user, but it is not part of the verified project.

[1] As of now, we only extract to OCaml because it is the only language that has plans to have a verified extraction process. We refer the reader to the specification of time library for more information.

1.3 Compilation and installation

In this section we describe how we obtained the executable file from the code. If the reader wishes to compile their code, they can use this explanation as a guide to compile the code on their machine.

1.3.1 Compilation

Here we describe the procedure that we follow in order to compile the code. This procedure works for both Linux and Windows (tested on Windows 10). The code files involved are the described in the previous section.

The steps that we follow to compile the code are the following:

1. First, we make sure that esy is installed. If not, we follow the instructions to install it. In section About esy we briefly explain how esy works and why it is safe to use it.
2. Then, on the root of the project, we run the following command in the terminal.

   ```
   esy install
   ```

   If we are on Windows, we run the previous command on a terminal with administrator privileges.

   Esy automatically installs the right version of the OCaml compiler, installs all the necessary dependencies and compiles the FV Time Manager. Below there is a schema that showcases the components involved in the compilation process.

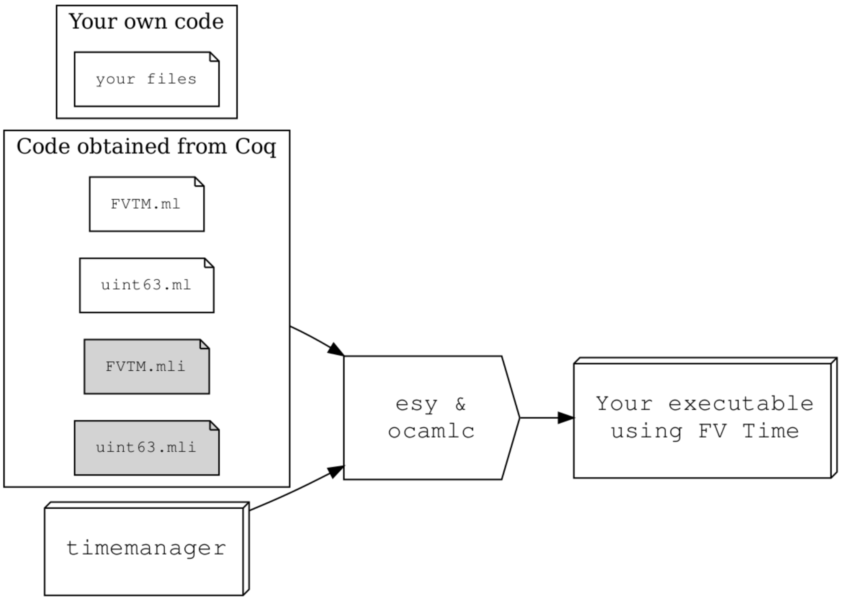

Figure 1.1: Compilation schema.

3. Finally, we are able to run the FV Time Manager with the following command.

   ```
   esy x timemanager
   ```

   This leaves the program waiting for queries from the user. To print on screen basic instructions on how to interact with it from the command line, we can type:

   ```
   --usage
   ```

If we want to specify a function and arguments, we write the name of the function and the arguments. For example:

```
from_utc_timestamp 1234567
```

After we have compiled the program, we have generated the executable `timemanager.exe` placed in the following path:

```
<compilation path>_esy/default/build/default/timemanager.exe
```

1.3.2 Using the executable

Windows

Once the executable is placed in the desired directory, it can be run from a command line with administrator privileges by typing:

```
timemanager.exe
```

If we want to make a query we will write it followed by an end-of-line character:

```
utc_timestamp 2021-12-01-15:45:32
```

All available functions are listed and documented later in this document.

Linux

Once the executable is placed in the desired directory, it can be run from a command line by typing:

```
./timemanager.exe
```

If we want to make a query we will write it followed by an end-of-line character:

```
utc_timestamp 2021-12-01-15:45:32
```

All available functions are listed and documented later in this document.

1.3.3 About esy

Esy is a tool which automates the process of downloading the proper version of the OCaml compiler (`ocamlc`), installing the necessary dependencies in an isolated environment and running the generated executable in that environment. It is important to emphasize that esy **does not** make any change to the OCaml code, hence, it is impossible that it introduces bugs in it.

We use it because it is a convenient tool to automatize and homogenize the compilation process in Linux and Windows.

1.4 Usage

As we have already pointed out, we call a function by appending its name and arguments on the same command.

Let's take a concrete example:

```
fun shift_utc_seconds : time → int63 → time
```

The above type signature means that the function `shift_utc_seconds` takes two arguments. The first argument is of type `time` and the second of type `int63`. The result is of type `time`.

Then, since `2021-12-31-23:23:50` is a valid `time` and `10` a valid `int63` we can perform a call to the Time Manager using the following command depending on the installation method used:

- **Using esy:**

  ```
  esy x timemanager
  shift_utc_seconds 2021-12-31-23:59:50 10
  ```

- **Using the `.exe` file:**

  ```
  timemanager.exe
  shift_utc_seconds 2021-12-31-23:59:50 10
  ```

As expected the answer will be `2022-01-01-00:00:00`, which is a valid `time`.

Sometimes it is convenient to give names to arguments so we can refer to them by name in the documentation. We use the syntax `(name : type)` to denote named arguments. Named arguments are only for documentation purposes. In other words, the following function definition is equivalent to the previous definition.

```
fun shift_utc_seconds : (some_time : time) → (offset : int) → time
```

Most functions have conditions on one or several arguments. If the conditions are not respected, the function will throw an error message.

The usage message can be printed by:

```
--usage
```

It is also possible to see the version number of the library and the date of its last update, which may be relevant to know when the leap seconds were updated:

```
--version
```

1.5 The UTC standard

The UTC (Coordinated Universal Time) is the most widespread standard for measuring and representing points in time. The counting of time is done according to **atomic clocks**.

UTC uses the Gregorian calendar to count days, which is the usual calendar: it accounts for the year, the month and the day within the month. Some years are leap and have an extra day in February according to the rule:

- Every year divisible by 400 is a leap year.
- Every year divisible by 4 but not by 100 is also a leap year.
- No other year is a leap year.

Seconds are counted following an atomic clock, i.e. all seconds have the exact same duration. But, since the Earth's rotation period slightly varies due to physical interactions, not all *solar days* have a duration of 86400 SI seconds. If no adjustment was made, the solar time would gradually differ from UTC. To avoid this effect, the UTC standard introduces **leap seconds**.

Leap seconds are introduced by a committee of experts (IERS) with at least six months of advance, and are impredictable in the long term. The only way to account for them is keeping an updated list of the past leap seconds. Theoretically, leap seconds can be positive (if added) or negative (if removed), but since the Earth tends to slow down, no negative leap second has ever happened.

The standard says that, when it occurs, a positive leap second is inserted between second 23:59:59 of a chosen UTC calendar date and second 00:00:00 of the following date. The definition of UTC states that the last day of December and June are preferred, with the last day of March or September as second preference, and the last day of any other month as third preference. All leap seconds (as of 2017) have been scheduled for either June 30 or December 31. The extra second is displayed on UTC clocks as 23:59:60. A negative leap second would suppress second 23:59:59 of the last day of a chosen month, so that second 23:59:58 of that date would be followed immediately by second 00:00:00 of the following date.

Although the UTC standard includes the format `Y-M-D-h:m:s`, time by atomic clocks is actually counted as a number of seconds (units of time) elapsed since some chosen point of time called *epoch*. This number of seconds is called **timestamp**. The most common epoch used, and the one our library chooses, is 1970-01-01-00:00:00. Then by definition its timestamp is 0 and any other timestamp starts counting from that time on.

UTC differs from International Atomic Time (TAI) by 37, the number of leap seconds that have been added. Most computer systems claim to implement UTC or GMT (which is not an official name) but actually implement TAI, or Unix, which is equal to UTC not counting leap seconds.

1.6 A new notion for consistent time arithmetic

Practically all commercial libraries for dealing with times and timestamps use Unix, even if they claim to use UTC. When adding and subtracting durations or time intervals to a given time, an issue arises due to the irregular periods that the Gregorian calendar and UTC define. For systems that work in Unix, the issue arises with months and years, because they don't have a constant duration. What these libraries do is to define an artificial operation on months and years that doesn't respect basic arithmetical properties.

They define *adding a month* as adding one to the month component of the time. But of course, the result of this operation is not always correct. For example, adding a month in this sense to `2009-01-31-14:32:54` yields `2009-02-31-14:32:54`, which is not a valid time because February doesn't have 31 days. Then, the adopted solution is correct the wrong component by going back to the previous valid one, so the result would be `2009-02-28-14:32:54`. Similarly, they can add any number to the month component, carrying to the year if necessary, and then correct the wrong component. For example, adding 24 months to `2008-02-29-15:00:00` gives `2010-02-28-15:00:00`. Analogously, the operation for adding years is defined.

This operation does not behave as addition does. For example, if in the first example we subtract one month to the result, we don't get the original time. Instead, we get `2009-01-28-14:32:54`. We can say that $1 - 1 \neq 0$!

Our library uses UTC, which means that this problem affects all the components to the left of seconds. Not all minutes have the same duration, nor all hours, nor all days. Our solution is to implement two different types of operations in time arithmetic. The first one, that we call shift functions, follows the same logic that other libraries follow with months and years, but with all the components. The second one is a definition of a new standard for durations called *formal time*, and operations called add-formal with it that behave consistently with basic arithmetical properties.

1.6.1 Shift functions

In that context, we extrapolate the logic described above for months and years of libraries working in Unix to the rest of the components for the UTC case. The shift function *shifts* a component of the time, carrying to the left if necessary, and then if the result is invalid performs corrections on the wrong component(s) to give a specific close previous valid time.

An example would be *subtracting two days* to the time `2016-12-31-23:59:60` (a leap second). With the procedure above before the correction, the result is `2016-12-29-23:59:60`. Since on that date there was no leap second, a correction must be performed on the wrong component to give the previous valid time, hence the final result is `2016-12-29-23:59:59`.

We chose the shift name because these functions are not proper addition. We call them **`shift_utc`** functions, and they have six instances: `shift_utc_seconds`, `shift_utc_minutes`, `shift_utc_hours`, `shift_utc_days`, `shift_utc_months`, and `shift_utc_years`.

1.6.2 Add-formal functions

In order to have time arithmetic with the usual arithmetical properties, we have define a new standard called *formal time*, that establishes standard durations for every component. A formal second is an atomic second, and a formal minute is 60 formal seconds, etc. See `formalTime` for all the details.

We have chosen to define formal months as 30 formal days. Therefore, *adding a formal month* is a clear arithmetical operation, consisting in adding a **constant** number of seconds (30 * 24 * 60 * 60 seconds). In the example above, adding a formal month to `2009-01-31-14:32:54` yields `2009-03-02-14:32:54`. The result is always valid by construction (except when it goes beyond the minimum or maximum date).

Now, *subtracting two formal days* to the time `2016-12-31-23:59:60` yields `2016-12-30-00:00:00`, which lets us see that the constant 86400 *cdot* 2 has been subtracted.

We define these functions with the name `add_formal`, and they have six instances: `add_formal_seconds`, `add_formal_minutes`, `add_formal_hours`, `add_formal_days`, `add_formal_months`, and `add_formal_years`. We also have the generic `add_formal` and `subtract_formal`.

These functions satisfy desirable arithmetical properties, for example if $t_1 + \Delta t = t_2$, then $t_2 - \Delta t = t_1$, i.e., $1 - 1 = 0$.

Notice that the only add-formal function that behaves exactly as its shift counterpart is `add_formal_seconds`, which is exactly the same as `shift_utc_seconds`.

1.7 The Coq Time Library: Main features

1. Fully **formally verified** using the Coq proof assistant.
2. **Fully in UTC**, including **leap seconds**.
3. Definition of the **formal time standard**: Durations are not well-defined in UTC, because due to leap seconds minutes, hours, etc. do not have constant durations. Our library solves two critical issues by introducing the `formalTime` **standard** for the time durations in UTC:

   - Give constant definitions for durations or time intervals, allowing to group durations in units bigger than seconds consistently.
   - Regarding **time arithmetic**, the library implements two kind of functions for addition and subtraction of `time` and durations.
     - The `shift_utc` functions work as is common in other libraries (the detailed description is below). Since durations are not constant, they do not satisfy certain basic arithmetical properties as the one described in the next item.
     - The `add_formal` functions are a new definition of addition and subtraction that works with the `formalTime` standard, hence satisfy for a time `t` and integer `n`: [Property of `add_formal` functions:] `add_formal_X (add_formal_X t n) -n = t`.
       In an improper non-functional language, this means that if $t_1 + \Delta t = t_2$, then $t_2 - \Delta t = t_1$.

4. Includes a function to check the version number and last updated date. Different versions may give different results due to the addition of leap seconds.

2 Definitions

2.1 type int63

```
type int63 = int
```

An int63 is a machine 63-bit integer. It can be used as signed or unsigned, depending on the function. Each function controls the range of its input.

Unsigned integers range from 0 to 9223372036854775807.

Signed integers range from -4611686018427387904 to 4611686018427387903.

2.2 type bool

```
type bool = string
```

A string representing a truth-value, `true` or `false`.

2.3 type weekday

```
type weekday = string
```

A `string` with one of the following values: `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday` or `Sunday`.

2.4 type date

```
type date = string
```

A `string` of the form `Y-M-D`. Valid values range from `1970-1-1` to `9999-12-31` and need to be valid UTC dates in the sense of valid_date.

As input, we allow both the `1970-01-01` and the `1970-1-1` formats. As output, we return the latter. We choose that one because it is directly supported by the Coq native integer library, and thus we avoid writing more code in the wrapper, which is non-verified.

2.5 type time

```
type time = string
```

A `string` of the form `Y-M-D-h:m:s`. Valid values range from `1970-1-1-00:00:00` to `9999-12-31-23:59:59` and need to be valid UTC times.

Depending on the system, a `time` needs to be written between double quotes to prevent issues due to the middle space. For example, `1980-04-26-13:42:09`.

As input, we allow both the `1970-01-01-00:00:00` and the `1970-1-1-0:0:0` formats. As output, we return the latter. We choose that one because it is directly supported by the Coq native integer library, and thus we avoid writing more code in the wrapper, which is non-verified.

2.6 type clock

```
type clock = string
```

A `string` of the form `h:m:s`, where:

- $0 \leq \texttt{h} \leq 23$
- $0 \leq \texttt{m} \leq 59$
- $0 \leq \texttt{s} \leq 60$ (because we account for leap seconds)

As input, we allow both the `00:00:00` and the `0:0:0` formats. As output, we return the latter. We choose that one because it is directly supported by the Coq native integer library, and thus we avoid writing more code in the wrapper, which is non-verified.

2.7 type formalTime

```
type formalTime = string
```

A `string` of the form `fY-fM-fD-fh-fm-fs` which expresses a time interval duration in formal time units.

Formal time unit	Duration
formal second	1 atomic second
formal minute	60 formal seconds
formal hour	60 formal minutes
formal day	24 formal hours
formal month	30 formal days
formal year	365 formal days

The value of each component is restricted to avoid overflowing the maximum `int63` when operating with a `formalTime`. The following must hold:

1. $0 \leq \texttt{fY} \leq 2924712086$,
2. $0 \leq \texttt{fM} \leq 35583997055$,
3. $0 \leq \texttt{fD} \leq 1067519911673$,
4. $0 \leq \texttt{fh} \leq 25620477880152$,
5. $0 \leq \texttt{fm} \leq 1537228672809129$,
6. $0 \leq \texttt{fs} \leq 92233720368547757$.

These are the only restrictions for a `formalTime` when given as input. As output, a `formalTime` is always given in *normal form*, meaning that the components satisfy the following conditions too:

- `fs` < 60,
- `fm` < 60,
- `fh` < 24,
- `fd` < 30,
- `fm` ≤ 12, and if `fd` ≥ 5 then `fm` < 12.

This last condition is because 12 formal months are 360 formal days, less than a formal year. Thus, it is allowed to say 0-12-4-0-0-0, but if we want to add one more formal day, then the expression becomes 1-0-0-0-0-0.

2.8 type coq_error

```
type coq_error =
    InvalidDate {
        d : date
        }
    | InvalidTime {
          t : time
          }
    | InvalidFormalTime {
          ft : formalTime
          }
    | InvalidYear {
          y : int63
          }
    | InvalidMonth {
          m : int63
          }
    | InvalidDay {
          yr : int63
          ; mth : int63
          ; dy : int63
          }
    | InvalidHour {
          h : int63
          }
    | InvalidMinute {
          min : int63
          }
    | InvalidSecond {
          dt : date
          ; hr : int63
          ; mn : int63
          ; sec : int63
          }
    | InvalidClockSecond {
          cs : int63
          }
    | NumberOutOfBounds {
          n : int63
          }
    | Overflow
    | InvalidOrderOfTimes {
          t1 : time
          ; t2 : time
          }
```

These errors are handled by Coq through a sum type that gives different constructors for different error messages.

Next to each error that the Coq code gives there is the error message that the wrapper writes:

Constructors

`InvalidDate` Coq error is sent by the wrapper to error message:

> Input `d` is invalid. Only dates in UTC (starting in 1970, ending in 9999) are accepted
>
> | `d : date.`

`InvalidTime` Coq error is sent by the wrapper to error message:

> Input `t` is invalid. Only times in UTC (with leap seconds and starting in 1970, ending in 9999) are accepted
>
> | `t : time.`

`InvalidFormalTime` Coq error is sent by the wrapper to error message:

> Input `ft` is invalid. Either some component is negative, or it is too big and would cause overflow
>
> | `ft : formalTime.`

`InvalidYear` Coq error is sent by the wrapper to error message:

> Input `y` is invalid. Only years between 1970 and 9999 are accepted
>
> | `y : int63.`

`InvalidMonth` Coq error is sent by the wrapper to error message:

> Input `m` is invalid. Months are a number between 1 and 12
>
> | `m : int63.`

`InvalidDay` Coq error is sent by the wrapper to error message:

> Input `dy` is invalid. In `yr`, month `mth`, days range between 1 and `days_of_month yr mth`
>
> | `yr : int63.`
>
> | `mth : int63.`
>
> | `dy : int63.`

`InvalidHour` Coq error is sent by the wrapper to error message:

> Input `h` is invalid. Hours are a number between 0 and 23
>
> | `h : int63.`

`InvalidMinute` Coq error is sent by the wrapper to error message:

> Input `min` is invalid. Minutes are a number between 0 and 59

min : int63.

InvalidSecond Coq error is sent by the wrapper to error message:

Input sec is invalid. On dt at hr:=mn=, seconds range between 0 and max_second dt hr mn

dt : date.

hr : int63.

mn : int63.

sec : int63.

InvalidClockSecond Coq error is sent by the wrapper to error message:

Input cs is invalid. Clock seconds are a number between 0 and 60

cs : int63.

NumberOutOfBounds Coq error is sent by the wrapper to error message:

Input n is out of bounds. Operating with it would lead to overflow the minimum or maximum time

n : int63.

Overflow Coq error is sent by the wrapper to error message:

Overflow: with the input you gave, the resulting time would be before 1970 or after 9999

InvalidOrderOfTimes Coq error is sent by the wrapper to error message:

The first input t1 is smaller than the second input t2. Time difference can only be computed if the first argument is greater than the second one

t1 : time.

t2 : time.

2.9 type format_error

```
type format_error =
    InvalidDateFormat
    | InvalidTimeFormat
    | InvalidFormalTimeFormat
    | InvalidInt63Format
    | UnrecognizedInput
```

These errors are not handled by the functions nor by Coq, since they are errors on the format of the input that only the wrapper can detect and handle.

Next to each kind of format error there is the message that the wrapper gives:

Constructors

InvalidDateFormat The date is not given in the correct format. Error message:

A date in the format Y-M-D was expected

InvalidTimeFormat The time is not given in the correct format. Error message:

A time in the format Y-M-D-h:m:s was expected

`InvalidFormalTimeFormat` The `formalTime` is not given in the correct format. Error message:

> A formal time in the format Y-M-D-h-m-s with non-negative components was expected

`InvalidInt63Format` The `int63` is not given in the correct format. Error message:

> An integer representable by OCaml's int63 was expected. Either the input is not an integer at all, or it is too big or too small for our language representation

`UnrecognizedInput` The input is not recognized at all. The name of the function may not exist or receive different inputs. Error message:

> The input was not recognized. Either that function does not exist or it receives differently formatted (or a different number of) inputs.
>
> In order to see the usage, run –usage

2.10 fun is_leap_year

```
fun is_leap_year : (year : int63) → bool
```

Returns `true` if `year` is a leap year in the Gregorian calendar and `false` otherwise.

Errors:

- `InvalidYear` if **not** $1970 \leq$ `year` ≤ 9999

Use examples:

- `is_leap_year 1984 = true`
- `is_leap_year 1973 = false`

2.11 fun days_of_month

```
fun days_of_month : (year : int63) → (month : int63) → int63
```

The number of days of a month with respect to that year.

Errors:

- `InvalidYear` if **not** $1970 \leq$ `year` ≤ 9999
- `InvalidMonth` if **not** $1 \leq$ `month` ≤ 12

Use examples:

- `days_of_month 1973 2 = 28`
- `days_of_month 1984 2 = 29`
- `days_of_month 1984 5 = 31`

2.12 fun max_second

```
fun max_second : (d : date) → (hour : int63) → (minute : int63) → int63
```

Returns the maximum value of the second for a given date at that `hour` and `minute`. Thus, the possible outcomes will be 59 for a regular minute, 60 for a positive leap second and 58 for a negative leap second.

Errors:

- InvalidDate if valid_date d = false
- InvalidHour if **not** $0 \leq$ hour ≤ 23
- InvalidMinute if **not** $0 \leq$ minute ≤ 59

Use examples:

- `max_second 2009-10-03 14 32 = 59`
- `max_second 2016-12-31 23 59 = 60`

2.13 fun valid_date

```
fun valid_date : date → bool
```

Checks for the validity of a date in UTC. Given a date Y-M-D, valid or invalid, returns true if and only if all the following hold:

- $1970 \leq$ Y ≤ 9999;
- $1 \leq$ M ≤ 12;
- $1 \leq$ D $\leq$ days_of_month Y M.

Use examples:

- `valid_date 2009-10-03 = true`
- `valid_date 2016-12-32 = false`

2.14 fun valid_time

```
fun valid_time : time → date
```

Checks for the validity of a time in UTC. Given a time Y-M-D-h:m:s, valid or invalid, returns true if and only if all the following hold:

- valid_date Y-M-D = true;
- $0 \leq$ h ≤ 23;
- $0 \leq$ m ≤ 59;
- $0 \leq$ s $\leq$ max_second Y-M-D h m.

Use examples:

- `valid_time 2009-10-03-14:32:60 = false`
- `valid_time 2016-12-31-23:59:60 = true`

2.15 fun version_date

```
fun version_date : date
```

Gives the date where the list of leap seconds was last updated. It has no arguments.

Use examples:

- version_date = 2021-11-29

2.16 fun date_of_time

```
fun date_of_time : (t : time) → date
```

Projection of the date part of a time.

Use examples:

- date_of_time 2009-10-03-14:32:59 = 2009-10-3
- date_of_time 2016-12-31-23:59:60 = 2016-12-31

2.17 fun clock_of_time

```
fun clock_of_time : (t : time) → clock
```

Takes as input a time and returns the clock corresponding to that time.

Errors:

- InvalidTime if valid_time t = false

Use examples:

- clock_of_time 2009-10-03-14:32:59 = 14:32:59
- clock_of_time 2016-12-31-23:59:60 = 23:59:60

2.18 fun second

```
fun second : (t : time) → int63
```

Projection of the second component of a time.

Use examples:

- second 2009-10-03-14:32:59 = 59
- second 2016-12-31-23:59:60 = 60

2.19 fun minute

```
fun minute : (t : time) → int63
```

Projection of the minute component of a time.

Use examples:

- minute 2009-10-03-14:32:59 = 32
- minute 2016-12-31-23:59:60 = 59

2.20 fun hour

```
fun hour : (t : time) → int63
```

Projection of the hour component of a `time`.

Use examples:

- `hour 2009-10-03-14:32:59 = 14`
- `hour 2016-12-31-23:59:60 = 23`

2.21 fun day

```
fun day : (d : date) → int63
```

Projection of the day component of a `time`.

Use examples:

- `day 2009-10-03-14:32:59 = 3`
- `day 2016-12-31-23:59:60 = 31`

2.22 fun month

```
fun month : (d : date) → int63
```

Projection of the month component of a `date`.

Use examples:

- `month 2009-10-03-14:32:59 = 10`
- `month 2016-12-31-23:59:60 = 12`

2.23 fun year

```
fun year : (d : date) → int63
```

Projection of the year component of a `date`.

Use examples:

- `year 2009-10-03-14:32:59 = 2009`
- `year 2016-12-31-23:59:60 = 2016`

2.24 fun weekday_of_date

```
fun weekday_of_date : (d : date) → weekday
```

Returns the day of the week of a date.

Errors:

- InvalidDate if valid_date d = false

Use examples:

- weekday_of_date 2009-10-03 = Saturday
- weekdat_of_date 2016-12-30 = Friday

2.25 fun utc_timestamp

```
fun utc_timestamp : (t : time) → int63
```

Conversion from time to its timestamp. Both types in UTC with leap seconds.

Errors:

- InvalidTime if valid_time t = false

Use examples:

- utc_timestamp 2009-10-03-14:32:25 = 1254580369
- utc_timestamp 2016-12-31-23:59:60 = 1483228826

2.26 fun from_utc_timestamp

```
fun from_utc_timestamp : (n : int63) → time
```

Conversion from a timestamp to its time. Both types in UTC with leap seconds.

Errors:

- NumberOutOfBounds if **not** $0 \leq n \leq 253402300826$

Use examples:

- from_utc_timestamp 1254580369 = 2009-10-3-14:32:25
- from_utc_timestamp 1483228826 = 2016-12-31-23:59:60

2.27 fun le_date

```
fun le_date : (d1 : date) → (d2 : date) → bool
```

Given two dates d1 and d2, returns true if d1 is less than or equal to d2 and false otherwise. In other words, ≤ for date.

Errors:

- InvalidDate if valid_date d1 = false or valid_date d2 = false

  Use examples:

- le_date 2009-10-3 2009-10-3 = true

- le_date 2016-12-31 2009-10-3 = false

2.28 `fun lt_date`

```
fun lt_date : (d1 : date) → (d2 : date) → bool
```

Given two dates `d1` and `d2`, returns `true` if `d1` is less than or equal to `d2` and `false` otherwise. In other words, $<$ for `date`.

Errors:

- `InvalidDate` if `valid_date d1` = `false` or `valid_date d2` = `false`

Use examples:

- `lt_date 2008-8-7 2009-10-3 = true`
- `lt_date 2016-12-31 2009-10-3 = false`

2.29 `fun le_time`

```
fun le_time : (t1 : time) → (t2 : time) → bool
```

Given two times `t1` and `t2`, returns `true` if `t1` is less than or equal to `t2` and `false` otherwise. In other words, $\leq$ for `time`.

Errors:

- `InvalidTime` if `valid_time t1` = `false` or `valid_time t2` = `false`

Use examples:

- `le_time 2009-10-3-14:32:25 2009-10-3-14:32:25 = true`
- `le_time 2016-12-31-23:59:60 2009-10-3-14:32:25 = false`

2.30 `fun lt_time`

```
fun lt_time : (t1 : time) → (t2 : time) → bool
```

Given two times `t1` and `t2`, returns `true` if `t1` is less than or equal to `t2` and `false` otherwise. In other words, $<$ for `time`.

Errors:

- `InvalidTime` if `valid_time t1` = `false` or `valid_time t2` = `false`

Use examples:

- `lt_time 2008-8-7-17:21:12 2009-10-3-14:32:25 = true`
- `lt_time 2016-12-31-23:59:60 2009-10-3-14:32:25 = false`

2.31 `fun from_formalTime`

```
fun from_formalTime : (ft : formalTime) → int63
```

Returns the number of seconds in a `formalTime`.

Errors:

- InvalidFormalTime if the bounds 1-6 given in formalTime are **not** respected

Use examples:

- `from_formalTime 0-0-0-35-30-100 = 127900`
- `from_formalTime 10-5-2-20-30-50 = 328566650`

2.32 fun to_formalTime

```
fun to_formalTime : int63 → formalTime
```

Converts a number of seconds in a formalTime.

Errors:

- NumberOutOfBounds if **not** $n \geq 92233720375632000$ ($365 \cdot 86400$ multiplied by the maximum value for the formal year component).

Use examples:

- `to_formalTime 127900 = 0-0-1-11-31-40`
- `to_formalTime 328566650 = 10-5-2-20-30-50`

2.33 fun add_formal

```
fun add_formal : (t : time) → (dur : formalTime) → time
```

Computes the result of adding a certain duration dur expressed in formalTime to a given time t.

Errors:

- InvalidTime if valid_time t = false
- InvalidFormalTime if the bounds 1-6 given in formalTime are **not** respected
- Overflow if the result would be before 1970 or after 9999

Use examples:

- `add_formal 2009-10-03-14:32:25 0-0-0-35-30-100 = 2009-10-5-2:4:5`
- `add_formal 2016-12-31-23:59:60 10-5-2-20-30-50 = 2027-5-31-20:30:49`

2.34 fun subtract_formal

```
fun subtract_formal : (t : time) → (dur : formalTime) → time
```

Computes the result of subtracting a certain duration dur expressed in formalTime to a given time t.

Errors:

- InvalidTime if valid_time t = false
- InvalidFormalTime if the bounds 1-6 given in formalTime are **not** respected

- Overflow if the result would be before 1970 or after 9999

Use examples:

- subtract_formal 2009-10-03-14:32:25 0-0-0-35-30-100 = 2009-10-2-3:0:45
- subtract_formal 2016-12-31-23:59:60 10-5-2-20-30-50 = 2006-8-4-3:29:13

2.35 fun shift_utc_seconds

```
fun shift_utc_seconds : (t : time) → (shift : int63) → time
```

Shifts the second component of the time t the number of times determined by the signed integer shift, carrying if needed to the components to the left. For a more detailed description see the section on time arithmetic.

Errors:

- InvalidTime if valid_time t = false
- NumberOutOfBounds if the result would be before 1970 or after 9999

Use examples:

- shift_utc_seconds 2009-10-03-14:32:25 18320 = 2009-10-3-19:37:45
- shift_utc_seconds 2016-12-31-23:59:60 -567812 = 2016-12-25-10:16:28

2.36 fun add_formal_seconds

```
fun add_formal_seconds : (t : time) → (amount : int63) → time
```

Adds a signed amount of formal seconds to a time. For a more detailed description see the section on time arithmetic.

Errors:

- InvalidTime if valid_time t = false
- NumberOutOfBounds if the result would be before 1970 or after 9999

Use examples:

- add_formal_seconds 2009-10-03-14:32:25 18320 = 2009-10-3-19:37:45
- add_formal_seconds 2016-12-31-23:59:60 -567812 = 2016-12-25-10:16:28

2.37 fun shift_utc_minutes

```
fun shift_utc_minutes : (t : time) → (shift : int63) → time
```

Shifts the minute component of the time t the number of times determined by the signed integer shift, carrying if needed to the components to the left. If the result is between 1970 and 9999 but would be an invalid time, the wrong component is corrected to the closest previous valid second. For a more detailed description see the section on time arithmetic.

Errors:

- InvalidTime if valid_time t = false

- `NumberOutOfBounds` if the result would be before 1970 or after 9999

Use examples:

- `shift_utc_minutes 2009-10-03-14:32:25 18320 = 2009-10-16-7:52:25`
- `shift_utc_minutes 2016-12-31-23:59:60 -567812 = 2015-12-3-16:27:59`

2.38 `fun add_formal_minutes`

```
fun add_formal_minutes : (t : time) → (amount : int63) → time
```

Adds a signed `amount` of formal minutes to a `time`. For a more detailed description see the section on time arithmetic.

Errors:

- `InvalidTime` if `valid_time t` = `false`
- `NumberOutOfBounds` if `amount` ≤ -76861433640456466 (minimum `int63` divided by the formal minute duration) or if `amount` > 76861433640456465 (maximum `int63` divided by the formal minute duration), or if the result would be before 1970 or after 9999

Use examples:

- `add_formal_minutes 2009-10-03-14:32:25 18320 = 2009-10-16-7:52:25`
- `add_formal_minutes 2016-12-31-23:59:60 -567812 = 2015-12-3-16:28:0`

2.39 `fun shift_utc_hours`

```
fun shift_utc_hours : (t : time) → (shift : int63) → time
```

Shifts the `hour` component of the time `t` the number of times determined by the signed integer `shift`, carrying if needed to the components to the left. If the result is between 1970 and 9999 but would be an invalid time, the wrong component is corrected to the closest previous valid second. For a more detailed description see the section on time arithmetic.

Errors:

- `InvalidTime` if `valid_time t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

Use examples:

- `shift_utc_hours 2009-10-03-14:32:25 18320 = 2011-11-5-22:32:25`
- `shift_utc_hours 2016-12-31-23:59:60 -5678 = 2016-5-9-9:59:59`

2.40 `fun add_formal_hours`

```
fun add_formal_hours : (t : time) → (amount : int63) → time
```

Adds a signed `amount` of formal hours to a `time`. For a more detailed description see the section on time arithmetic.

Errors:

- `InvalidTime` if `valid_time t` = `false`
- `NumberOutOfBounds` if `amount` ≤ -1281023894007608 (minimum `int63` divided by the formal hour duration) or if `amount` > 1281023894007607 (maximum `int63` divided by the formal hour duration), or if the result would be before 1970 or after 9999

Use examples:

- `add_formal_hours 2009-10-03-14:32:25 18320 = 2011-11-5-22:32:25`
- `add_formal_hours 2016-12-31-23:59:60 -5678 = 2016-5-9-10:0:0`

2.41 `fun shift_utc_days`

```
fun shift_utc_days : (t : time) → (shift : int63) → time
```

Shifts the `day` component of the time `t` the number of times determined by the signed integer `shift`, carrying if needed to the components to the left. If the result is between 1970 and 9999 but would be an invalid time, the wrong component is corrected to the closest previous valid second. For a more detailed description see the section on time arithmetic.

Errors:

- `InvalidTime` if `valid_time t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

Use examples:

- `shift_utc_days 2009-10-03-14:32:25 10 = 2009-10-13-14:32:25`
- `shift_utc_days 2016-12-31-23:59:60 -5 = 2016-12-26-23:59:59`

2.42 `fun add_formal_days`

```
fun add_formal_days : (t : time) → (amount : int63) → time
```

Adds a signed `amount` of formal days to a `time`. For a more detailed description see the section on time arithmetic.

Errors:

- `InvalidTime` if `valid_time t` = `false`
- `NumberOutOfBounds` if `amount` ≤ -53375995583651 (minimum `int63` divided by the formal day duration) or if `amount` > 53375995583650 (maximum `int63` divided by the formal day duration), or if the result would be before 1970 or after 9999

Use examples:

- `add_formal_days 2009-10-03-14:32:25 10 = 2009-10-13-14:32:25`
- `add_formal_days 2016-12-31-23:59:60 -5 = 2016-12-27-00:00:00`

2.43 `fun shift_utc_months`

```
fun shift_utc_months : (t : time) → (shift : int63) → time
```

Shifts the month component of the time t the number of times determined by the signed integer shift, carrying if needed to the components to the left. If the result is between 1970 and 9999 but would be an invalid time, the wrong component is corrected to the closest previous valid date (if the date is invalid), and after that, the second component is corrected to the previous valid second (if the time is still invalid). For a more detailed description see the section on time arithmetic.

Errors:

- InvalidTime if valid_time t = false
- NumberOutOfBounds if the result would be before 1970 or after 9999

Use examples:

- `shift_utc_months 2009-10-03-14:32:25 18320 = 3536-6-3-14:32:25`
- `shift_utc_months 2016-12-31-23:59:60 -560 = 1970-4-30-23:59:59`

2.44 fun add_formal_months

```
fun add_formal_months : (t : time) → (amount : int63) → time
```

Adds a signed amount of formal months to a time. For a more detailed description see the section on time arithmetic.

Errors:

- InvalidTime if valid_time t = false
- NumberOutOfBounds if amount ≤ -1779199852789 (minimum int63 divided by the formal month duration) or if amount > 1779199852788 (maximum int63 divided by the formal month duration) or if the result would be before 1970 or after 9999

Use examples:

- `add_formal_months 2009-10-03-14:32:25 18320 = 3514-7-6-14:32:22`
- `add_formal_months 2016-12-31-23:59:60 -560 = 1971-1-3-0:0:26`

2.45 fun shift_utc_years

```
fun shift_utc_years : (t : time) → (shift : int63) → time
```

Shifts the year component of the time t the number of times determined by the signed integer shift. If the result is between 1970 and 9999 but would be an invalid time, the wrong component is corrected to the closest previous valid date (if the date is invalid), and after that, the second component is corrected to the previous valid second (if the time is still invalid). For a more detailed description see the section on time arithmetic.

Errors:

- InvalidTime if valid_time t = false
- NumberOutOfBounds if the result would be before 1970 or after 9999

Use examples:

- `shift_utc_years 2009-10-03-14:32:25 1832 = 3841-10-3-14:32:25`
- `shift_utc_years 2016-12-31-23:59:60 -45 = 1971-12-31-23:59:59`

2.46 fun add_formal_years

```
fun add_formal_years : (t : time) → (amount : int63) → time
```

Adds a signed amount of formal years to a time. For a more detailed description see the section on time arithmetic.

Errors:

- InvalidTime if valid_time t = false
- NumberOutOfBounds if amount ≤ -146235604339 (minimum int63 divided by the formal year duration) or if amount > 146235604338 (maximum int63 divided by the formal year duration), or if the result would be before 1970 or after 9999

Use examples:

- add_formal_years 2009-10-03-14:32:25 1832 = 3840-7-16-14:32:22
- add_formal_years 2016-12-31-23:59:60 -45 = 1972-1-13-0:0:26

2.47 fun time_difference

```
fun time_difference : (t1 : time) → (t2 : time) → formalTime
```

Given two times t1 and t2, computes the duration expressed in formalTime between them.

Errors:

- InvalidTime if valid_time t1 = false or valid_time t2 = false
- InvalidOrderOfTimes if lt_time t1 t2 = false

Use examples:

- time_difference 2016-12-31-23:59:60 2009-10-03-14:32:25 = 7-3-1-9-27-37
- time_difference 3840-7-16-4:2:34 2009-10-03-14:32:25 = 1831-12-4-13-30-12

2.48 fun sec_time_difference

```
fun sec_time_difference : (t1 : time) → (t2 : time) → int63
```

Given two times t1 and t2, computes the duration expressed in seconds between them.

Errors:

- InvalidTime if valid_time t1 = false or valid_time t2 = false
- InvalidOrderOfTimes if lt_time t1 t2 = false

Use examples:

- sec_time_difference 2016-12-31-23:59:60 2009-10-03-14:32:25 = 228648457
- sec_time_difference 3840-7-16-4:2:34 2009-10-03-14:32:25 = 57773914212