CSE 341, Autumn 2010
Assignment #5 - ML: Date (50 points)
Due: Friday, November 5, 2010, 9:30 AM

This assignment focuses on ML's mechanisms to extend its type system using structures, signatures, and records. You are to complete the definition of a structure called `CalendarDate` that can be used to represent calendar dates. Your structure will provide various functions for date manipulation, comparison, and arithmetic.

Your structure should be restricted to a signature known as `CALENDARDATE`, provided to you in a file named `CALENDARSIG.sml`. This will hide any helper functions in your structure. The signature looks like this:

```
signature CALENDARDATE = sig
    type calendarDate;
    datatype day = Sun | Mon | Tue | Wed | Thu | Fri | Sat;
    exception IllegalDate;

    val new: int * int * int -> calendarDate;
    val compare: calendarDate * calendarDate -> order;
    val daysBetween: calendarDate * calendarDate -> int;
    val daysInMonth: calendarDate -> int;
    val daysInYear: calendarDate -> int;
    val isLeapYear: calendarDate -> bool;
    val next: calendarDate -> date;
    val previous: calendarDate -> date;
    val shift: calendarDate * int -> date;

    (* extra credit *)
    val dayOfWeek: calendarDate -> day;
    val today: unit -> calendarDate;
end;
```

Your task is to implement all functions, types, exceptions, etc. specified in the signature and described below. Put your code into a file named `CalendarDate.sml`. You are allowed to define helper functions in your structure if you like.

Throughout your structure you should maintain the **invariant** that every date instance that is created will always be in a valid state. A valid date is one where the year is at least 1753 (the modern calendar began on that year), the month is between 1 and 12 inclusive, and the day is between 1 and the number of days in that month inclusive. In other words, if the month is 9 (September), the day must be between 1-30; if the month is July, the day must be between 1-31; and so on. Any function that creates a new date or modifies an existing one must maintain this invariant.

The actual data to represent a date is represented as a value of **type `calendarDate`**. In your `CalendarDate` structure, you **must** define the `calendarDate` type exactly as follows:

```
type calendarDate = {year: int, month: int, day: int};
```

This will define your type to be an alias for a record with three `int` fields named `year`, `month`, and `day` . For example, internally inside your file, the date of September 19, 1979 could be represented as:

```
val example = {year=1979, month=9, day=19};       (* Sep 19, 1979 *)
```

Recall from lecture that you can write various kinds of patterns to match records that will allow you to examine the individual pieces inside of a given record instance. For example, the following patterns could all match a date instance and might be useful in various places:

```
d1
{year, month, day}                    (* various patterns that match date records *)
{month, ...}
{year=y, month=m, day=d}
d1 as {year=y1, month=m1, day=d1}
```

There is also an enumerated `datatype` called `day`, but that is used only in the extra credit function `dayOfWeek`.

Except for a few constraints at the end of this document, **this assignment will not be graded on internal correctness**. The entirety of your grade will come from the external correctness of your code and its ability to pass various test cases. Also please note that this assignment is worth half of the normal credit (50 points versus 100 points).

Implementation Details:

You must implement the following functions. Note that they are listed in alphabetical order (except `new`), not necessarily in the order that is best to solve them. You may want to read about all of the functions before starting any of them.

Note that if you declare that your structure implements the `CALENDARDATE` signature, your code won't compile until you write a version of all functions listed here. You may want to **hold off on connecting your structure to the signature** while developing your program so that you can test the functions one-by-one.

If you don't finish a particular function, write a **"stub" version** that returns a default value such as `0` or `""`, so that your code will at least compile and be testable for grading. Note that there are a few **extra credit** functions defined later in this document; even if you don't choose to implement them, you must at least insert such a "stub" to be able to test your code.

`new` `val new: int * int * int -> calendarDate`

Define a function named `new` accepts three int parameters representing a year, month, and day (in that order) and returns a date representing the given year, month, and date. This is roughly equivalent to a constructor in Java.

Your function should raise an `IllegalDate` exception if the parameters represent an invalid date.

Because your new function will always return valid dates and because it is the only way to construct a date, this means that you may assume in all other functions that any date parameters passed are valid.

`compare` `val compare: calendarDate * calendarDate -> order`

Define a function named `compare` that accepts two dates as parameters and returns a value of type order indicating whether the first date comes earlier (`LESS`) or later (`GREATER`) than the second date, or `EQUAL` if they are the same. For example, September 19, 1995 is `LESS` than July 23, 1998, and March 1, 2010 is `GREATER` than Feb 28, 2010.

`daysBetween` `val daysBetween: calendarDate * calendarDate -> int`

Define a function named `daysBetween` that accepts two dates as parameters and returns the number of days that separate the two. For example, there are 10 days between September 19, 2010 and September 29, 2010. If the dates are the same, your function should return 0. The result returned should always be a **non-negative** value; the days between two dates d1 and d2 is the same as the days between d2 and d1, passed in the opposite order.

`daysInMonth` `val daysInMonth: calendarDate -> int`

Define a function named `daysInMonth` that returns the number of days in the month represented by the given date. For example, if the date represents Sep. 19, 2010, your method should return 30. Here are the number of days in each month:

Month	1 Jan	2 Feb	3 Mar	4 Apr	5 May	6 Jun	7 Jul	8 Aug	9 Sep	10 Oct	11 Nov	12 Dec
Days	31	28 *	31	30	31	30	31	31	30	31	30	31

* Your code must deal with **leap years**. February normally has 28 days, but in a leap year, February has 29 days. Most years that are divisible by 4, such as 1996 and 2004, are leap years. However, years that are multiples of 100, such as 1900 or 1700, are *not* leap years, unless they are also multiples of 400, such as 1600 and 2000.

`daysInYear` `val daysInYear: calendarDate -> int`

Define a function named `daysInYear` that accepts a date as a parameter and returns the number of days in the year represented by the given date. Leap years have 366 days, and all other years have 365 days.

`isLeapYear` `val isLeapYear: calendarDate -> bool`

Define a function named `isLeapYear` that accepts a date parameter and returns `true` if that date takes place during a year that is a leap year. Most years that are divisible by 4, such as 1996 and 2004, are leap years. However, years that are multiples of 100, such as 1900 or 1700, are *not* leap years, unless they are also multiples of 400, such as 1600 and 2000.

next `val next: calendarDate -> calendarDate`

Define a function named `next` that accepts a date and returns the date that occurs one day after that date. For example, if the date passed represents September 19, 2010, a call to this method should return a date that represents September 20, 2010. Note that depending on the date, a call to this method might return a date that has advanced into the next month or year. For example, the next day after June 30, 1846 is July 1, 1846; the day after February 28, 1997 is March 1, 1997; the day after February 28, 2004 is February 29, 2004; and the day after December 31, 1999 is January 1, 2000.

previous `val previous: calendarDate -> calendarDate`

Define a function named `previous` that accepts a date as a parameter and returns the date that occurred one day before the date that was passed in. For example, if the date passed represents September 19, 2010, a call to this method should return a date representing September 18, 2010. Note that depending on the date, a call to this function might return a date that has moved into the previous month or year. For example, the previous day before July 1, 1846 is June 30, 1846; the previous day before March 1, 1997 is February 28, 1997; the day before March 1, 2004 is February 29, 2004; and the day before January 1, 2000 is December 31, 1999. If the date passed is January 1, 1753, raise an `IllegalDate` exception.

shift `val shift: calendarDate * int -> calendarDate`

Define a function named `shift` that accepts a date and an integer *shift* as parameters and returns a new date that is exactly *shift* days away from the date passed in. The *shift* can be positive, negative, or zero. For example, shifting September 19, 2010 by 10 days results in September 29, 2010; shifting it by -10 days results in September 9, 2010. Note that depending on the date and *shift* values, this function might return a date that has shifted into a different month or year. For example, shifting March 1, 2010 by 145 days results in July 24, 2010; shifting December 10, 2005 by -500 days results in July 28, 2004; and shifting September 19, 1979 by 11363 days results in October 29, 2010. You may assume that the client will not attempt to shift to a date earlier than January 1, 1753.

toString `val toString: calendarDate -> string`

Define a function named `toString` that accepts a date as a parameter and returns a string representing that date in year/month/day format. For example, if April 5, 2010 is passed, the function should return `"2010/4/5"`, and if December 25, 1998 is passed, the function should return `"1998/12/25"`.

Extra Credit (+1 point each):

For +1 point each, you can implement some or all of the following optional features. **If you don't implement the extra credit functions, you must still write a version of the functions that returns a default value, such as `0` or `Sun` or `{year=2010, month=1, day=1}`, etc. so that your structure properly implements the `CALENDARDATE` signature.**

dayOfWeek `val dayOfWeek: calendarDate -> day`

Write a function named `dayOfWeek` that accepts a date as a parameter and returns an instance of the datatype `day` representing the day of the week on which the given date falls. For example, October 28, 2010 occurred on a Thursday, so `Thu` should be returned. September 19, 1979 occurred on a Wednesday, so `Wed` should be returned.

To implement this function, you can take advantage of the fact that January 1, 1753 was a Monday and base your calculations on this. Recall that valid dates occur on or after Jan 1, 1753.

today `val today: unit -> calendarDate`

Write a function named `today` that accepts no parameters (in ML, technically we say that it accepts the `unit`, `()`, as its parameter) and returns a date that represents the current date on which the program is being run. In other words, if you run your program on October 29, 2010, calling this function returns a date representing October 29, 2010. If you run your program on February 5, 2037, calling this function returns a date representing that date instead.

To implement this function, call ML's standard library function `Time.now()`, which returns a structure representing the current time. (The time returned is in UTC time; to shift to the current time zone, subtract the local date/time offset. Write `Time.-(Time.now(), Date.localOffset())` .) Call the `Time.toSeconds` function and pass this structure to get a large integer value representing the number of seconds that have passed since 12:00 AM on January 1, 1970.

Extra Credit (+1 point each) (continued):

Efficiency Optimizations:

Several functions in this assignment are unnecessarily slow. For example, the `shift` function can take a long time if its algorithm runs *n* non-tail-recursive calls for a shift of *n* days (one call per day). A faster algorithm would be a tail-recursive function that processed an entire year at a time. For **+1 extra point**, speed up the following functions by making them fully tail-recursive as well as making sure that they process entire years at a time rather than single days at a time.

- `daysBetween`
- `shift`
- `dayOfWeek` (if you wrote it)
- `today`

One example test case would be to try to ask for the days between September 19, 1979 and October 26, 900010 (there are 327999125 such days), or to shift the former date by the previous number of days to reach the latter date. An unoptimized version of our own solution code takes over 10 seconds to perform such operations, while an optimized version returns immediately. You could also ask for the day of the week of October 26, 900010 (it's going to be a Tuesday).

Beautiful / Hideous Code Awards:

Since the assignment is essentially not graded on internal correctness at all, any code that meets the loose internal requirements in this spec could get full credit. Because of that, as an incentive for students to write interesting code, we will award **+1 point** to the student whose program is the most "beautiful," i.e., solves the assignment the most elegantly and optimally. We will also award **+1 point** to the program that is the most "hideous," i.e., has the most confusing, obfuscated, indecipherable, awful code while still being (at least mostly) correct externally. The winners will be decided by the TAs and instructor and will be shared anonymously with the class at a later date.

Grading and Submission:

Submit your `Date.sml` file electronically using the link on the class web page.

Our solution is **44 "substantive" lines** long on the class Indenter page, excluding blank lines/ comments (56 with all extra credit features implemented). You don't need to match this number or even come close to it; it is just a rough guideline.

Your program should not produce **syntax errors or warnings**, such as "non-exhaustive match." (A "polyEqual" warning is okay.) You may lose points if you name your functions or top-level values incorrectly, even if their behavior is correct.

Your code should work for both basic and advanced cases. Perform your own **testing**, and remember to test edge cases, such as dates that are very close together, very far apart, leap years, non-leap years, wrapping, the very far future, etc. It is okay to share any testing code you write with your classmates by posting it on the class message board.

Efficiency on a fine level of detail is not crucial on this assignment. But if one of your functions does not produce its result in a reasonable amount of time (a few seconds), it may not receive full credit.

This program is **not graded on internal correctness**, except for the following minor constraints:

- You may not use the built-in `Date` type from the ML Standard Basis Library, or any other date/time-related standard libraries, in your solution. You also may not download or use other date-related libraries to help you. Your solution must be your own work, in which you write the heart of the algorithms to solve the problem.
- You must use our `CALENDARDATE.sml` file and implement the `CalendarDate` structure completely to match the `CALENDARDATE` signature, such that our test code can create instances of your type through that signature.

Other than the above constraints, you may use any ML constructs and functions from the ML basis library.

We suggest that you place a **comment header** at the top of the program that has at least your name, to minimize the chance of student programs getting "mixed up." But you will not be graded on commenting and none is required.

As always, if the solution to one function is useful in helping you solve a later function, you should consider calling the earlier function from the later function, though since we aren't grading on internal correctness, this is not required. You can optionally include any testing code you develop inside your program.