

Appendix D

Lisp Implementation

It has been often said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e., express it as an algorithm.

Donald E. Knuth: "Computer Science and its Relation to Mathematics,"
American Mathematical Monthly (1974)

This appendix contains the complete Common Lisp implementation of the calendar functions described in the main text; the equation numbers given here are those of the corresponding functions in the text. Some Lisp functions have no corresponding equation in the text—these are constructors, selectors, and standard mathematical operations that are also used to control the typesetting: the functions in the main text were automatically typeset from the definitions in this appendix. The Lisp functions are available over the World Wide Web at

www.cambridge.org/calendricalcalculations

Please bear in mind the limits of the License and that the copyright on this book includes the code. *Also please keep in mind that if the result of any calculation is critical, it should be verified by independent means.*

For licensing information about nonpersonal and other uses, contact the authors. The code is distributed in the hope that it may be useful but without any warranty as to the accuracy of its output and with liability limited to return of the price of this book, which restrictions are set forth on page xli.

D.1 Basics

D.1.1 Lisp Preliminaries

For readers unfamiliar with Lisp, this section provides the bare necessities. A complete description can be found in [2].

All functions in Lisp are written in prefix notation. If f is a defined function, then

```
(f e0 e1 e2 ... en)
```

applies f to the $n + 1$ arguments $e0, e1, e2, \dots, en$. Thus, for example, $+$ adds up a list of numbers; for example,

```
(+ 1 -2 3)
```

adds the three numbers and returns the value 2. The Lisp functions $-$, $*$, and $/$ work similarly, to subtract, multiply, and divide, respectively, a list of numbers. In a similar fashion, $<=$ ($\leq$) checks that the numbers are in nondecreasing order and yields true (t in Lisp) if the relations hold. For instance,

```
(<= 1 2 3)
```

evaluates to t . The Lisp functions $=$, $\neq$ (not equal), $<$, $>$, and $>=$ (greater than or equal) are similar. The predicate `evenp` tests whether an integer is even.

Lists are Lisp's main data structure. To construct a list $(e0\ e1\ e2\ \dots\ en)$ the expression

```
(list e0 e1 e2 ... en)
```

is used. The function `nth`, used as $(nth\ i\ l)$, extracts the i th element of the list l , indexing from 0; the predicate `member`, used as $(member\ x\ l)$, tests whether x is an element of l . To get the first (indexed 0), second, and so on, through tenth elements of a list, we use the functions `first`, `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth`, and `tenth`. The tail of the list, consisting of all the elements but the first, is obtained using `rest`. The empty list is represented by `nil`.

Constants are defined with the `defconstant` command, which has the syntax

```
(defconstant constant-name  
  expression)
```

For example,

```
1 (defconstant sunday  
2   ;; TYPE day-of-week  
3   ;; Residue class for Sunday.  
4   0)
```

(1.53)

```

1  (defconstant monday
2    ;; TYPE day-of-week
3    ;; Residue class for Monday.
4    1)

```

```

1  (defconstant tuesday
2    ;; TYPE day-of-week
3    ;; Residue class for Tuesday.
4    2)

```

```

1  (defconstant wednesday
2    ;; TYPE day-of-week
3    ;; Residue class for Wednesday.
4    3)

```

```

1  (defconstant thursday
2    ;; TYPE day-of-week
3    ;; Residue class for Thursday.
4    4)

```

```

1  (defconstant friday
2    ;; TYPE day-of-week
3    ;; Residue class for Friday.
4    5)

```

```

1  (defconstant saturday
2    ;; TYPE day-of-week
3    ;; Residue class for Saturday.
4    6)

```

(1.54) Notice that semicolons mark the start of comments. “Type” information is given in comments for each function. Although Common Lisp has its own system of type declarations, we preferred the simpler, untyped, Lisp, but we annotate each function and constant to aid the reader in translating our code into a typed language. The base types are defined in Table A.1, beginning on page 389.

To distinguish in the code between empty lists (`nil`) and the truth value “false,” we define

```

1  (defconstant false
2    ;; TYPE boolean
3    ;; Constant representing false.
4    nil)

```

For “true,” we define

```

1  (defconstant true
2    ;; TYPE boolean
3    ;; Constant representing true.
4    t)

```

We also use a string constant to signify an error value:

```

1  (defconstant bogus                                     (1.97)
2    ;; TYPE string
3    ;; Used to denote nonexistent dates.
4    "bogus")

```

The function `equal` can be used to check lists and strings for equality.

Functions are defined using the `defun` command, which has the following syntax:

```

(1.58) (defun function-name (param1 ... paramn)
        expression)

```

For example, we compute the day of the week of an `R.D. date` (page 33) with

```

1  (defun day-of-week-from-fixed (date)                                     (1.60)
2    ;; TYPE fixed-date -> day-of-week
3    ;; The residue class of the day of the week of date.
4    (mod (- date (rd 0) sunday) 7))

```

and we implement julian day calculations by writing

```

1  (defconstant jd-epoch                                (1.3)
2  ;; TYPE moment
3  ;; Fixed time of start of the julian day number.
4  (rd -1721424.5L0))

```

Common Lisp uses L0 after a number to specify unscaled maximum-precision (at least 50-bit) constants.

We use the identity function

```

1  (defun rd (tee)                                       (1.1)
2  ;; TYPE moment -> moment
3  ;; Identity function for fixed dates/moments. If internal
4  ;; timekeeping is shifted, change epoch to be RD date of
5  ;; origin of internal count. epoch should be an integer.
6  (let* ((epoch 0))
7  (- tee epoch)))

```

to make it easy to adapt the code to an alternate fixed-date enumeration—all that is needed is to change the value of *epoch* in line 6 of *rd*. The Common Lisp construct *let** defines a sequence of constants (possibly in terms of previously defined constants) and ends with an expression whose value is returned by the construct.

```

1  (defun moment-from-jd (jd)                            (1.4)
2  ;; TYPE julian-day-number -> moment
3  ;; Moment of julian day number jd.
4  (+ jd jd-epoch))

```

```

1  (defun jd-from-moment (tee)                            (1.5)
2  ;; TYPE moment -> julian-day-number
3  ;; Julian day number of moment tee.
4  (- tee jd-epoch))

```

```

1  (defconstant mjd-epoch                                (1.6)
2  ;; TYPE fixed-date
3  ;; Fixed time of start of the modified julian day number.
4  (rd 678576))

```

```

1  (defun fixed-from-mjd (mjd)                            (1.7)
2  ;; TYPE julian-day-number -> fixed-date
3  ;; Fixed date of modified julian day number mjd.
4  (+ mjd mjd-epoch))

```

```

1  (defun mjd-from-fixed (date)                            (1.8)
2  ;; TYPE fixed-date -> julian-day-number
3  ;; Modified julian day number of fixed date.
4  (- date mjd-epoch))

```

```

1  (defconstant unix-epoch                                (1.9)
2  ;; TYPE fixed-date
3  ;; Fixed date of the start of the Unix second count.
4  (rd 719163))

```

```

1  (defun moment-from-unix (s)                            (1.10)
2  ;; TYPE second -> moment
3  ;; Fixed date from Unix second count s
4  (+ unix-epoch (/ s 24 60 60)))

```

```

1  (defun unix-from-moment (tee)                            (1.11)
2  ;; TYPE moment -> second
3  ;; Unix second count from moment tee
4  (* 24 60 60 (- tee unix-epoch)))

```

```

1  (defun fixed-from-jd (jd)                              (1.13)
2  ;; TYPE julian-day-number -> fixed-date
3  ;; Fixed date of julian day number jd.
4  (floor (moment-from-jd jd)))

```

```

1  (defun jd-from-fixed (date)                            (1.14)
2  ;; TYPE fixed-date -> julian-day-number
3  ;; Julian day number of fixed date.
4  (jd-from-moment date))

```

As another example of a function definition, we can define a function (inconveniently named `floor` in Common Lisp) to return the (truncated) integer quotient of two integers, $\lfloor m/n \rfloor$:

```
1 (defun quotient (m n)
2   ;; TYPE (real nonzero-real) -> integer
3   ;; Whole part of m/n.
4   (floor m n))
```

The `floor` function can also be called with one argument. Thus

```
(floor x)
```

is $\lfloor x \rfloor$, the greatest integer less than or equal to x .

As a final example of function definitions, note that the Common Lisp function `mod` *always returns a nonnegative value for a positive divisor*; we use this property occasionally, but we also need a function like `mod` with its values adjusted in such a way that the modulus of a multiple of the divisor is the divisor itself rather than 0. To define this function, we write

```
1 (defun amod (x y)                                     (1.29)
2   ;; TYPE (integer nonzero-integer) -> integer
3   ;; The value of (x mod y) with y instead of 0.
4   (+ y (mod x (- y))))
```

This is typeset as $x \bmod [1 \dots y]$ in the main text.

More generally, we use a function that shifts the modulus into a specified range of values [1]:

```
1 (defun mod3 (x a b)                                   (1.24)
2   ;; TYPE (real real real) -> real
3   ;; The value of x shifted into the range
4   ;; [a..b). Returns x if a=b.
5   (if (= a b)
6       x
7       (+ a (mod (- x a) (- b a)))))
```

This is typeset as $x \bmod [a \dots b)$; see page 22.

The function `if` has three arguments: a boolean condition, a then-expression, and an else-expression. The `cond` statement, also used in what follows, lists a sequence of tests and values and serves as a generalized case statement.

For convenience in expressing our calendar functions in Lisp, we introduce a macro to compute sums. The expression

```
(sum f i k p)
```

computes

$$\sum_{k \leq i < \min_{j \geq k} \{\neg p(j)\}} f(i);$$

that is, the expression $f(i)$ is summed for all $i = k, k+1, \dots$, continuing only as long as the condition $p(i)$ holds. The sum is 0 if $p(k)$ is false. Our Common Lisp definition of `sum` uses the versatile `loop` construct and is as follows:

```
1 (defmacro sum (expression index initial condition)      (1.30)
2   ;; TYPE ((integer->real) * integer (integer->boolean))
3   ;; TYPE -> real
4   ;; Sum expression for index = initial and successive
5   ;; integers, as long as condition holds.
6   `(loop for ,index from ,initial
7         while ,condition
8         sum ,expression))
```

This is the first of the few instances in which we use macros and not functions; it allows us to avoid the issue of passing functions to functions.

A similar macro, **prod**, is used for products:

```
1 (defmacro prod (expression index initial condition)    (1.31)
2   ;; TYPE ((integer->real) * integer (integer->boolean))
3   ;; TYPE -> real
4   ;; Product of expression for index = initial and successive
5   ;; integers, as long as condition holds.
6   `(apply '*
7         (loop for ,index from ,initial
8               while ,condition
9               collect ,expression)))
```

The `collect` construct gathers a list of factors and the function `apply` applies the multiplication operation to that list.

A summation macro **sigma** and a summation function **poly** for polynomials are used mainly in the astronomical code:

```
1 (defmacro sigma (list body)
2   ;; TYPE (list-of-pairs (list-of-reals->real))
3   ;; TYPE  -> real
4   ;; list is of the form ((i1 l1)...(in ln)).
5   ;; Sum of body for indices i1...in
6   ;; running simultaneously thru lists l1...ln.
7   `(apply '+ (mapcar (function (lambda
8     , (mapcar 'car list)
9     ,body))
10    ,@(mapcar 'cadr list))))

1 (defun poly (x a)
2   ;; TYPE (real list-of-reals) -> real
3   ;; Sum powers of x with coefficients (from order 0 up)
4   ;; in list a.
5   (if (equal a nil)
6       0
7       (+ (first a) (* x (poly x (rest a))))))
```

The function `mapcar` applies a function (expressed by means of function and `lambda`) to each element of a list.

Two additional **sum**-like macros are used for searching; the first implements the **MIN** function, equation (1.32), and the second implements **MAX**, equation (1.33):

```
1 (defmacro next (index initial condition) (1.32)
2   ;; TYPE (* integer (integer->boolean)) -> integer
3   ;; First integer greater or equal to initial such that
4   ;; condition holds.
5   `(loop for ,index from ,initial
6     when ,condition
7     return ,index))
```

```
1 (defmacro final (index initial condition) (1.33)
2   ;; TYPE (* integer (integer->boolean)) -> integer
3   ;; Last integer greater or equal to initial such that
4   ;; condition holds.
5   `(loop for ,index from ,initial
6     when (not ,condition)
7     return (1- ,index)))
```

The function `1-` decrements a number by one; the similar function `1+` increments by one.

We also use binary search—see equation (1.35)—expressed as the macro **binary-search**:

```
1 (defmacro binary-search (l lo h hi x test end) (1.35)
2   ;; TYPE (* real * real * (real->boolean))
3   ;; TYPE ((real real)->boolean) -> real
4   ;; Bisection search for x in [lo..hi] such that
5   ;; end holds. test determines when to go left.
6   (let* ((left (gensym)))
7     `(do* ((,x false (/ (+ ,h ,l) 2))
8       (,left false ,test)
9       (,l ,lo (if ,left ,l ,x))
10      (,h ,hi (if ,left ,x ,h)))
11      (,end (/ (+ ,h ,l) 2))))))
```

The construct `do*` is a form of loop.

Binary search is used mainly for function inversion:

```
1 (defmacro invert-angular (f y r) (1.36)
2   ;; TYPE (real->angle real interval) -> real
3   ;; Use bisection to find inverse of angular function
4   ;; f at y within interval r.
5   (let* ((varepsilon 1/100000); Desired accuracy
6     '(binary-search l (begin ,r) u (end ,r) x
7       (< (mod (- (,f x) ,y) 360) (deg 180))
8       (< (- u l) ,varepsilon))))
```

The interval selectors, **begin** and **end**, are defined below.

D.1.2 Basic Code

To extract a particular component from a date, we use, when necessary, the functions **standard-month**, **standard-day**, and **standard-year**. For example:

```
1 (defun standard-month (date)
2   ;; TYPE standard-date -> standard-month
3   ;; Month field of date = (year month day).
4   (second date))
```

```
1 (defun standard-day (date)
2   ;; TYPE standard-date -> standard-day
3   ;; Day field of date = (year month day).
4   (third date))
```

```
1 (defun standard-year (date)
2   ;; TYPE standard-date -> standard-year
3   ;; Year field of date = (year month day).
4   (first date))
```

Such constructors and selectors could be defined as macros or Lisp structures. In languages like C or C++, these would more naturally be field selection in fixed-length records rather than lists.

We also have

```
1 (defun hour (clock)
2   ;; TYPE clock-time -> hour
3   (first clock))
```

```
1 (defun minute (clock)
2   ;; TYPE clock-time -> minute
3   (second clock))
```

```
1 (defun seconds (clock)
2   ;; TYPE clock-time -> second
3   (third clock))
```

```
1 (defun time-of-day (hour minute second)
2   ;; TYPE (hour minute second) -> clock-time
3   (list hour minute second))
```

```
1 (defun fixed-from-moment (tee)
2   ;; TYPE moment -> fixed-date
3   ;; Fixed-date from moment tee.
4   (floor tee))
```

(1.12)

```
1 (defun sign (y)
2   ;; TYPE real -> {-1,0,+1}
3   ;; Sign of y.
4   (cond
5     ((< y 0) -1)
6     ((> y 0) +1)
7     (t 0)))
```

(1.16)

```
1 (defun time-from-moment (tee)
2   ;; TYPE moment -> time
3   ;; Time from moment tee.
4   (mod tee 1))
```

(1.18)

```
1 (defun list-of-fixed-from-moments (ell)
2   ;; TYPE list-of-moments -> list-of-fixed-dates
3   ;; List of fixed dates corresponding to list ell
4   ;; of moments.
5   (if (equal ell nil)
6       nil
7       (append (list (fixed-from-moment (first ell)))
8                 (list-of-fixed-from-moments (rest ell))))))
```

(1.37)

```

1 (defun interval (t0 t1)
2   ;; TYPE (moment moment) -> interval
3   ;; Half-open interval [t0..t1].
4   (list t0 t1))

```

```

1 (defun interval-closed (t0 t1)
2   ;; TYPE (moment moment) -> interval
3   ;; Closed interval [t0..t1].
4   (list t0 t1))

```

```

1 (defun begin (range)
2   ;; TYPE interval -> moment
3   ;; Start t0 of range [t0..t1] or [t0..t1].
4   (first range))

```

```

1 (defun end (range)
2   ;; TYPE interval -> moment
3   ;; End t1 of range [t0..t1] or [t0..t1].
4   (second range))

```

```

1 (defun in-range? (tee range)
2   ;; TYPE (moment interval) -> boolean
3   ;; True if tee is in half-open range.
4   (and (<= (begin range) tee) (< tee (end range))))

```

```

1 (defun list-range (ell range)
2   ;; TYPE (list-of-moments interval) -> list-of-moments
3   ;; Those moments in list ell that occur in range.
4   (if (equal ell nil)
5       nil
6       (let* ((r (list-range (rest ell) range)))
7         (if (in-range? (first ell) range)
8             (append (list (first ell)) r)
9             r))))

```

```

1 (defun positions-in-range (p c cap-Delta range)
2   ;; TYPE (nonnegative-real positive-real
3   ;; TYPE nonnegative-real interval) -> list-of-moments
4   ;; List of occurrences of moment p of c-day cycle
5   ;; within range.
6   ;; cap-Delta is position in cycle of RD moment 0.
7   (let* ((a (begin range))
8          (b (end range))
9          (date (mod3 (- p cap-Delta) a (+ a c))))
10    (if (>= date b)
11        nil
12        (append (list date)
13                (positions-in-range p c cap-Delta
14                                    (interval (+ a c) b))))))

```

The following two functions for mixed-radix conversions (see Section 1.10) take an optional third parameter for the fractional part of the basis:

```

(1.38)
1 (defun from-radix (a b &optional c)
2   ;; TYPE (list-of-reals list-of-rationals list-of-rationals)
3   ;; TYPE -> real
4   ;; The number corresponding to a in radix notation
5   ;; with base b for whole part and c for fraction.
6   (/ (sum (* (nth i a)
7              (prod (nth j (append b c))
                     j i (< j (+ (length b) (length c))))))
8      i 0 (< i (length a)))
9   (apply '* c)))
(1.39)
10

```

where length measures the length of a list; and

```

1 (defun to-radix (x b &optional c)
2   ;; TYPE (real list-of-rationals list-of-rationals)
3   ;; TYPE -> list-of-reals
4   ;; The radix notation corresponding to x
5   ;; with base b for whole part and c for fraction.
6   (if (null c)
7       (if (null b)
8           (list x)
9           (append (to-radix (quotient x (nth (1- (length b)) b))
10                          (butlast b) nil)
11                  (list (mod x (nth (1- (length b)) b))))))
12   (to-radix (* x (apply 'c)) (append b c))))

```

which is implemented recursively.

```

1 (defun time-from-clock (hms)
2   ;; TYPE clock-time -> time
3   ;; Time of day from hms = hour:minute:second.
4   (/ (from-radix hms nil (list 24 60 60)) 24))

```

```

1 (defun clock-from-moment (tee)
2   ;; TYPE moment -> clock-time
3   ;; Clock time hour:minute:second from moment tee.
4   (rest (to-radix tee nil (list 24 60 60))))

```

```

1 (defun angle-from-degrees (alpha)
2   ;; TYPE angle -> list-of-reals
3   ;; List of degrees-arcminutes-arcseconds from angle alpha
4   ;; in degrees.
5   (let* ((dms (to-radix (abs alpha) nil (list 60 60))))
6     (if (>= alpha 0)
7         dms
8         (list ; degrees-minutes-seconds
9               (- (first dms)) (- (second dms)) (- (third dms))))))

```

(1.42)

(1.43)

(1.44)

(1.45)

D.1.3 The Egyptian and Armenian Calendars

```

1 (defun egyptian-date (year month day)
2   ;; TYPE (egyptian-year egyptian-month egyptian-day)
3   ;; TYPE -> egyptian-date
4   (list year month day))

```

```

1 (defconstant egyptian-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Egyptian (Nabonassar)
4   ;; calendar.
5   ;; JD 1448638 = February 26, 747 BCE (Julian).
6   (fixed-from-jd 1448638))

```

(1.46)

```

1 (defun fixed-from-egyptian (e-date)
2   ;; TYPE egyptian-date -> fixed-date
3   ;; Fixed date of Egyptian date e-date.
4   (let* ((month (standard-month e-date))
5          (day (standard-day e-date))
6          (year (standard-year e-date)))
7     (+ egyptian-epoch ; Days before start of calendar
        (* 365 (1- year)); Days in prior years
        (* 30 (1- month)); Days in prior months this year
        day -1))) ; Days so far this month

```

(1.47)

```

1 (defun alt-fixed-from-egyptian (e-date)
2   ;; TYPE egyptian-date -> fixed-date
3   ;; Fixed date of Egyptian date e-date.
4   (+ egyptian-epoch
5       (sigma ((a (list 365 30 1))
6              (e-date e-date))
7           (* a (1- e-date)))))

```

(1.48)

```

1 (defun egyptian-from-fixed (date)
2   ;; TYPE fixed-date -> egyptian-date
3   ;; Egyptian equivalent of fixed date.
4   (let* ((days ; Elapsed days since epoch.
5          (- date egyptian-epoch))
6          (year ; Year since epoch.
7              (1+ (quotient days 365)))
8          (month; Calculate the month by division.
9              (1+ (quotient (mod days 365)
10                           30)))
11         (day ; Calculate the day by subtraction.
12             (- days
13               (* 365 (1- year))
14               (* 30 (1- month))
15               -1)))
16   (egyptian-date year month day)))

```

```

1 (defun armenian-date (year month day)
2   ;; TYPE (armenian-year armenian-month armenian-day)
3   ;; TYPE -> armenian-date
4   (list year month day))

```

```

1 (defconstant armenian-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Armenian calendar.
4   ;; = July 11, 552 CE (Julian).
5   (rd 201443))

```

```

1 (defun fixed-from-armenian (a-date)
2   ;; TYPE armenian-date -> fixed-date
3   ;; Fixed date of Armenian date a-date.
4   (let* ((month (standard-month a-date))
5          (day (standard-day a-date))

```

```

(1.49) 6 (year (standard-year a-date)))
7 (+ armenian-epoch
8   (- (fixed-from-egyptian
9       (egyptian-date year month day))
10      egyptian-epoch)))

```

```

1 (defun armenian-from-fixed (date) (1.52)
2   ;; TYPE fixed-date -> armenian-date
3   ;; Armenian equivalent of fixed date.
4   (egyptian-from-fixed
5    (+ date (- egyptian-epoch armenian-epoch))))

```

D.1.4 Cycles of Days

```

1 (defun kday-on-or-before (k date) (1.62)
2   ;; TYPE (day-of-week fixed-date) -> fixed-date
3   ;; Fixed date of the k-day on or before fixed date.
4   ;; k=0 means Sunday, k=1 means Monday, and so on.
5   (- date (day-of-week-from-fixed (- date k))))

```

```

1 (defun kday-on-or-after (k date) (1.65)
2   ;; TYPE (day-of-week fixed-date) -> fixed-date
3   ;; Fixed date of the k-day on or after fixed date.
4   ;; k=0 means Sunday, k=1 means Monday, and so on.
5   (kday-on-or-before k (+ date 6)))

```

```

1 (defun kday-nearest (k date) (1.66)
2   ;; TYPE (day-of-week fixed-date) -> fixed-date
3   ;; Fixed date of the k-day nearest fixed date.
4   ;; k=0 means Sunday, k=1 means Monday, and so on.
5   (kday-on-or-before k (+ date 3)))

```

```

1 (defun kday-before (k date)
2   ;; TYPE (day-of-week fixed-date) -> fixed-date
3   ;; Fixed date of the k-day before fixed date.
4   ;; k=0 means Sunday, k=1 means Monday, and so on.
5   (kday-on-or-before k (- date 1)))

```

```

1 (defun kday-after (k date)
2   ;; TYPE (day-of-week fixed-date) -> fixed-date
3   ;; Fixed date of the k-day after fixed date.
4   ;; k=0 means Sunday, k=1 means Monday, and so on.
5   (kday-on-or-before k (+ date 7)))

```

D.1.5 Akan Calendar

```

1 (defun akan-day-name (n)
2   ;; TYPE integer -> akan-name
3   ;; The n-th name of the Akan cycle.
4   (akan-name (amod n 6)
5             (amod n 7)))

```

```

1 (defun akan-name (prefix stem)
2   ;; TYPE (akan-prefix akan-stem) -> akan-name
3   (list prefix stem))

```

```

1 (defun akan-prefix (name)
2   ;; TYPE akan-name -> akan-prefix
3   (first name))

```

```

1 (defun akan-stem (name)
2   ;; TYPE akan-name -> akan-stem
3   (second name))

```

```

(1.67) 1 (defun akan-name-difference (a-name1 a-name2)
2       ;; TYPE (akan-name akan-name) -> nonnegative-integer
3       ;; Number of names from Akan name a-name1 to the
4       ;; next occurrence of Akan name a-name2.
5       (let* ((prefix1 (akan-prefix a-name1))
6              (prefix2 (akan-prefix a-name2))
7              (stem1 (akan-stem a-name1))
8              (stem2 (akan-stem a-name2))
9              (prefix-difference (- prefix2 prefix1))
10             (stem-difference (- stem2 stem1)))
11         (amod (+ prefix-difference
12                (* 36 (- stem-difference
13                        prefix-difference))))
13         42)))
14

```

```

(1.76) 1 (defconstant akan-day-name-epoch
2       ;; TYPE fixed-date
3       ;; RD date of an epoch (day 0) of Akan day cycle.
4       (rd 37))

```

```

1 (defun akan-name-from-fixed (date)
2   ;; TYPE fixed-date -> akan-name
3   ;; Akan name for date.
4   (akan-day-name (- date akan-day-name-epoch)))

```

```

(1.80) 1 (defun akan-day-name-on-or-before (name date)
2       ;; TYPE (akan-name fixed-date) -> fixed-date
3       ;; Fixed date of latest date on or before fixed date
4       ;; that has Akan name.
5       (mod3
6         (akan-name-difference (akan-name-from-fixed 0) name)
7         date (- date 42)))

```

D.2 The Gregorian Calendar

```
1 (defun gregorian-date (year month day)
2   ;; TYPE (gregorian-year gregorian-month gregorian-day)
3   ;; TYPE -> gregorian-date
4   (list year month day))
```

```
1 (defconstant gregorian-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the (proleptic) Gregorian
4   ;; calendar.
5   (rd 1))
```

```
1 (defconstant january
2   ;; TYPE standard-month
3   ;; January on Julian/Gregorian calendar.
4   1)
```

```
1 (defconstant february
2   ;; TYPE standard-month
3   ;; February on Julian/Gregorian calendar.
4   2)
```

```
1 (defconstant march
2   ;; TYPE standard-month
3   ;; March on Julian/Gregorian calendar.
4   3)
```

```
1 (defconstant april
2   ;; TYPE standard-month
3   ;; April on Julian/Gregorian calendar.
4   4)
```

```
1 (defconstant may
2   ;; TYPE standard-month
3   ;; May on Julian/Gregorian calendar.
4   5) (2.8)
```

```
1 (defconstant june
2   ;; TYPE standard-month
3   ;; June on Julian/Gregorian calendar.
4   6) (2.9)
```

```
1 (defconstant july
2   ;; TYPE standard-month
3   ;; July on Julian/Gregorian calendar.
4   7) (2.10)
```

```
1 (defconstant august
2   ;; TYPE standard-month
3   ;; August on Julian/Gregorian calendar.
4   8) (2.11)
```

```
1 (defconstant september
2   ;; TYPE standard-month
3   ;; September on Julian/Gregorian calendar.
4   9) (2.12)
```

```
1 (defconstant october
2   ;; TYPE standard-month
3   ;; October on Julian/Gregorian calendar.
4   10) (2.13)
```

```

1  (defconstant november                                     (2.14) 16      (- (* 367 month) 362); ...assuming 30-day Feb
2      ;; TYPE standard-month                               17      12)
3      ;; November on Julian/Gregorian calendar.           18      (if (<= month 2) ; Correct for 28- or 29-day Feb
4      11)                                                  19      0
                                                            20      (if (gregorian-leap-year? year)
                                                            21          -1
                                                            22          -2))
1  (defconstant december                                   (2.15) 23      day))) ; Days so far this month.
2      ;; TYPE standard-month
3      ;; December on Julian/Gregorian calendar.
4      12)

1  (defun gregorian-leap-year? (g-year)                     (2.16) 5
2      ;; TYPE gregorian-year -> boolean
3      ;; True if g-year is a leap year on the Gregorian
4      ;; calendar.
5      (and (= (mod g-year 4) 0)
6            (not (member (mod g-year 400)
7                          (list 100 200 300)))))

1  (defun fixed-from-gregorian (g-date)                     (2.17)
2      ;; TYPE gregorian-date -> fixed-date
3      ;; Fixed date equivalent to the Gregorian date g-date.
4      (let* ((month (standard-month g-date))
5             (day (standard-day g-date))
6             (year (standard-year g-date)))
7          (+ (1- gregorian-epoch); Days before start of calendar
8             (* 365 (1- year)); Ordinary days since epoch
9             (quotient (1- year)
10                      4); Julian leap days since epoch...
11             (- ; ...minus century years since epoch...
12               (quotient (1- year) 100))
13             (quotient ; ...plus years since epoch divisible...
14               (1- year) 400) ; ...by 400.
15             (quotient ; Days in prior months this year...

1  (defun gregorian-new-year (g-year)                       (2.18) 2
2      ;; TYPE gregorian-year -> fixed-date
3      ;; Fixed date of January 1 in g-year.
4      (fixed-from-gregorian
5        (gregorian-date g-year january 1)))

1  (defun gregorian-year-end (g-year)                       (2.19) 2
2      ;; TYPE gregorian-year -> fixed-date
3      ;; Fixed date of December 31 in g-year.
4      (fixed-from-gregorian
5        (gregorian-date g-year december 31)))

1  (defun gregorian-year-range (g-year)                     (2.20) 2
2      ;; TYPE gregorian-year -> range
3      ;; The range of moments in Gregorian year g-year.
4      (interval (gregorian-new-year g-year)
5                (gregorian-new-year (1+ g-year))))

1  (defun gregorian-year-from-fixed (date)                  (2.21) 2
2      ;; TYPE fixed-date -> gregorian-year
3      ;; Gregorian year corresponding to the fixed date.
4      (let* ((d0 ; Prior days.
5              (- date gregorian-epoch))
6              (n400 ; Completed 400-year cycles.
7                (quotient d0 146097)))

```

```

8      (d1      ; Prior days not in n400.
9      (mod d0 146097))
10     (n100    ; 100-year cycles not in n400.
11     (quotient d1 36524))
12     (d2      ; Prior days not in n400 or n100.
13     (mod d1 36524))
14     (n4      ; 4-year cycles not in n400 or n100.
15     (quotient d2 1461))
16     (d3      ; Prior days not in n400, n100, or n4.
17     (mod d2 1461))
18     (n1      ; Years not in n400, n100, or n4.
19     (quotient d3 365))
20     (year (+ (* 400 n400)
21             (* 100 n100)
22             (* 4 n4)
23             n1)))
24     (if (or (= n100 4) (= n1 4))
25         year      ; Date is day 366 in a leap year.
26         (1+ year))) ; Date is ordinal day (1+ (mod d3 365))
27                     ; in (1+ year).

```

```

1  (defun gregorian-from-fixed (date) (2.23)
2  ;; TYPE fixed-date -> gregorian-date
3  ;; Gregorian (year month day) corresponding to fixed date.
4  (let* ((year (gregorian-year-from-fixed date))
5         (prior-days; This year
6         (- date (gregorian-new-year year)))
7         (correction; To simulate a 30-day Feb
8         (if (< date (fixed-from-gregorian
9                 (gregorian-date year march 1)))
10             0
11             (if (gregorian-leap-year? year)
12                 1
13                 2)))
14         (month      ; Assuming a 30-day Feb

```

```

15     (quotient
16     (+ (* 12 (+ prior-days correction)) 373)
17     367))
18     (day      ; Calculate the day by subtraction.
19     (1+ (- date
20           (fixed-from-gregorian
21           (gregorian-date year month 1)))))
22     (gregorian-date year month day))

```

```

1  (defun gregorian-date-difference (g-date1 g-date2) (2.24)
2  ;; TYPE (gregorian-date gregorian-date) -> integer
3  ;; Number of days from Gregorian date g-date1 until
4  ;; g-date2.
5  (- (fixed-from-gregorian g-date2)
6     (fixed-from-gregorian g-date1))

```

```

1  (defun day-number (g-date) (2.25)
2  ;; TYPE gregorian-date -> positive-integer
3  ;; Day number in year of Gregorian date g-date.
4  (gregorian-date-difference
5   (gregorian-date (1- (standard-year g-date)) december 31)
6   g-date))

```

```

1  (defun days-remaining (g-date) (2.26)
2  ;; TYPE gregorian-date -> nonnegative-integer
3  ;; Days remaining in year after Gregorian date g-date.
4  (gregorian-date-difference
5   g-date
6   (gregorian-date (standard-year g-date) december 31))

```

```

1  (defun last-day-of-gregorian-month (g-year g-month) (2.27)
2  ;; TYPE (gregorian-year gregorian-month) -> gregorian-day
3  ;; Last day of month g-month in Gregorian year g-year.
4  (gregorian-date-difference

```

```

5      (gregorian-date g-year g-month 1)
6      (gregorian-date (if (= g-month 12)
7                          (1+ g-year)
8                          g-year)
9                          (amod (1+ g-month) 12)
10                         1)))

```

```

1  (defun alt-fixed-from-gregorian (g-date)
2    ;; TYPE gregorian-date -> fixed-date
3    ;; Alternative calculation of fixed date equivalent to the
4    ;; Gregorian date g-date.
5    (let* ((month (standard-month g-date))
6           (day (standard-day g-date))
7           (year (standard-year g-date))
8           (m-prime (mod (- month 3) 12))
9           (y-prime (- year (quotient m-prime 10))))
10      (+ (1- gregorian-epoch)
11         -306 ; Days in March...December.
12         (* 365 y-prime); Ordinary days.
13         (sigma ((y (to-radix y-prime (list 4 25 4)))
14                 (a (list 97 24 1 0)))
15                (* y a))
16         (quotient ; Days in prior months.
17                   (+ (* 3 m-prime) 2)
18                     5)
19         (* 30 m-prime)
20         day))) ; Days so far this month.

```

```

1  (defun alt-gregorian-from-fixed (date)
2    ;; TYPE fixed-date -> gregorian-date
3    ;; Alternative calculation of Gregorian (year month day)
4    ;; corresponding to fixed date.
5    (let* ((y (gregorian-year-from-fixed
6              (+ (1- gregorian-epoch)

```

(2.28)

(2.29)

```

7      date
8      306)))
9      (prior-days
10      (- date (fixed-from-gregorian
11              (gregorian-date (1- y) march 1))))
12      (month
13      (amod (+ (quotient
14                (+ (* 5 prior-days) 2)
15                  153)
16              3)
17              12)))
18      (year (- y (quotient (+ month 9) 12)))
19      (day
20      (1+ (- date
21            (fixed-from-gregorian
22              (gregorian-date year month 1))))))
23      (gregorian-date year month day)))

```

```

1  (defun alt-gregorian-year-from-fixed (date)
2    ;; TYPE fixed-date -> gregorian-year
3    ;; Gregorian year corresponding to the fixed date.
4    (let* ((approx ; approximate year
5            (quotient (- date gregorian-epoch -2)
6                      146097/400))
7           (start ; start of next year
8                 (+ gregorian-epoch
9                     (* 365 approx)
10                     (sigma ((y (to-radix approx (list 4 25 4)))
11                             (a (list 97 24 1 0)))
12                             (* y a)))))
13      (if (< date start)
14          approx
15          (1+ approx))))

```

(2.30)

```

1  (defun independence-day (g-year)
2    ;; TYPE gregorian-year -> fixed-date

```

(2.32)

```

3  ;; Fixed date of United States Independence Day in
4  ;; Gregorian year g-year.
5  (fixed-from-gregorian (gregorian-date g-year july 4)))

```

```

1  (defun nth-kday (n k g-date) (2.33)
2  ;; TYPE (integer day-of-week gregorian-date) -> fixed-date
3  ;; If n>0, return the n-th k-day on or after
4  ;; g-date. If n<0, return the n-th k-day on or
5  ;; before g-date. If n=0 return bogus. A k-day of
6  ;; 0 means Sunday, 1 means Monday, and so on.
7  (cond ((> n 0)
8         (+ (* 7 n)
9            (kday-before k (fixed-from-gregorian g-date))))
10        (< n 0)
11        (+ (* 7 n)
12           (kday-after k (fixed-from-gregorian g-date))))
13        (t bogus)))

```

```

1  (defun first-kday (k g-date) (2.34)
2  ;; TYPE (day-of-week gregorian-date) -> fixed-date
3  ;; Fixed date of first k-day on or after Gregorian date
4  ;; g-date. A k-day of 0 means Sunday, 1 means Monday,
5  ;; and so on.
6  (nth-kday 1 k g-date))

```

```

1  (defun last-kday (k g-date) (2.35)
2  ;; TYPE (day-of-week gregorian-date) -> fixed-date
3  ;; Fixed date of last k-day on or before Gregorian date
4  ;; g-date. A k-day of 0 means Sunday, 1 means Monday,
5  ;; and so on.
6  (nth-kday -1 k g-date))

```

```

1  (defun labor-day (g-year) (2.36)
2  ;; TYPE gregorian-year -> fixed-date
3  ;; Fixed date of United States Labor Day in Gregorian
4  ;; year g-year (the first Monday in September).
5  (first-kday monday (gregorian-date g-year september 1)))

```

```

1  (defun memorial-day (g-year) (2.37)
2  ;; TYPE gregorian-year -> fixed-date
3  ;; Fixed date of United States Memorial Day in Gregorian
4  ;; year g-year (the last Monday in May).
5  (last-kday monday (gregorian-date g-year may 31)))

```

```

1  (defun election-day (g-year) (2.38)
2  ;; TYPE gregorian-year -> fixed-date
3  ;; Fixed date of United States Election Day in Gregorian
4  ;; year g-year (the Tuesday after the first Monday in
5  ;; November).
6  (first-kday tuesday (gregorian-date g-year november 2)))

```

```

1  (defun daylight-saving-start (g-year) (2.39)
2  ;; TYPE gregorian-year -> fixed-date
3  ;; Fixed date of the start of United States daylight
4  ;; saving time in Gregorian year g-year (the second
5  ;; Sunday in March).
6  (nth-kday 2 sunday (gregorian-date g-year march 1)))

```

```

1  (defun daylight-saving-end (g-year) (2.40)
2  ;; TYPE gregorian-year -> fixed-date
3  ;; Fixed date of the end of United States daylight saving
4  ;; time in Gregorian year g-year (the first Sunday in
5  ;; November).
6  (first-kday sunday (gregorian-date g-year november 1)))

```

```

1 (defun christmas (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of Christmas in Gregorian year g-year.
4   (fixed-from-gregorian
5     (gregorian-date g-year december 25)))

```

```

1 (defun advent (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of Advent in Gregorian year g-year
4   ;; (the Sunday closest to November 30).
5   (kday-nearest sunday
6     (fixed-from-gregorian
7       (gregorian-date g-year november 30))))

```

```

1 (defun epiphany (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of Epiphany in U.S. in Gregorian year
4   ;; g-year (the first Sunday after January 1).
5   (first-kday sunday (gregorian-date g-year january 2)))

```

```

1 (defun unlucky-fridays-in-range (range)
2   ;; TYPE range -> list-of-fixed-dates
3   ;; List of Fridays within range of dates
4   ;; that are day 13 of Gregorian months.
5   (let* ((a (begin range))
6          (b (end range))
7          (fri (kday-on-or-after friday a))
8          (date (gregorian-from-fixed fri)))
9     (if (in-range? fri range)
10        (append
11          (if (= (standard-day date) 13)
12              (list fri)
13              nil)

```

```

(2.41) 14 (unlucky-fridays-in-range
15         (interval (1+ fri) b)))
16         nil)))

```

```

(2.42) 1 (defun unlucky-fridays (g-year)
2         ;; TYPE gregorian-year -> list-of-fixed-dates
3         ;; List of Fridays within Gregorian year g-year
4         ;; that are day 13 of Gregorian months.
5         (unlucky-fridays-in-range
6           (gregorian-year-range g-year)))
(2.45)

```

D.3 The Julian Calendar

In the Lisp code we use *-n* for year *n* B.C.E. (Julian):

```

(2.43) 1 (defun bce (n)
2         ;; TYPE standard-year -> julian-year
3         ;; Negative value to indicate a BCE Julian year.
4         (~ n))

```

and positive numbers for C.E. (Julian) years:

```

(2.44) 1 (defun ce (n)
2         ;; TYPE standard-year -> julian-year
3         ;; Positive value to indicate a CE Julian year.
4         n)

```

```

1 (defun julian-date (year month day)
2   ;; TYPE (julian-year julian-month julian-day)
3   ;; TYPE -> julian-date
4   (list year month day))

```

```

1 (defun julian-leap-year? (j-year)
2   ;; TYPE julian-year -> boolean
3   ;; True if j-year is a leap year on the Julian calendar.
4   (= (mod j-year 4) (if (> j-year 0) 0 3)))
(3.1)

```

```

1 (defconstant julian-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Julian calendar.
4   (fixed-from-gregorian (gregorian-date 0 december 30)))

1 (defun fixed-from-julian (j-date)
2   ;; TYPE julian-date -> fixed-date
3   ;; Fixed date equivalent to the Julian date j-date.
4   (let* ((month (standard-month j-date))
5          (day (standard-day j-date))
6          (year (standard-year j-date))
7          (y (if (< year 0)
8                (1+ year) ; No year zero
9                year)))
10    (+ (1- julian-epoch) ; Days before start of calendar
11       (* 365 (1- y)) ; Ordinary days since epoch.
12       (quotient (1- y) 4); Leap days since epoch...
13       (quotient ; Days in prior months this year...
14        (- (* 367 month) 362); ...assuming 30-day Feb
15        12)
16       (if (<= month 2) ; Correct for 28- or 29-day Feb
17         0
18         (if (julian-leap-year? year)
19             -1
20             -2))
21       day))) ; Days so far this month.

```

```

1 (defun julian-from-fixed (date)
2   ;; TYPE fixed-date -> julian-date
3   ;; Julian (year month day) corresponding to fixed date.
4   (let* ((approx ; Nominal year.
5          (quotient (+ (* 4 (- date julian-epoch)) 1464)
6                    1461))
7          (year (if (<= approx 0)

```

```

(3.2) 8 (1- approx) ; No year 0.
9 approx))
10 (prior-days; This year
11 (- date (fixed-from-julian
12          (julian-date year january 1))))
13 (correction; To simulate a 30-day Feb
14 (if (< date (fixed-from-julian
15          (julian-date year march 1))))
16 0
17 (if (julian-leap-year? year)
18     1
19     2)))
20 (month ; Assuming a 30-day Feb
21 quotient
22 (+ (* 12 (+ prior-days correction)) 373)
23 367))
24 (day ; Calculate the day by subtraction.
25 (1+ (- date
26        (fixed-from-julian
27         (julian-date year month 1)))))
28 (julian-date year month day)))

```

```

1 (defconstant kalends
2   ;; TYPE roman-event
3   ;; Class of Kalends.
4   1)
(3.5)

```

```

1 (defconstant nones
2   ;; TYPE roman-event
3   ;; Class of Nones.
4   2)
(3.6)

```

```

1 (defconstant ides
2   ;; TYPE roman-event
3   ;; Class of Ides.
4   3)
(3.7)

```

```

1 (defun roman-date (year month event count leap)
2   ;; TYPE (roman-year roman-month roman-event roman-count)
3   ;; TYPE roman-leap -> roman-date
4   (list year month event count leap))

```

```

1 (defun roman-year (date)
2   ;; TYPE roman-date -> roman-year
3   (first date))

```

```

1 (defun roman-month (date)
2   ;; TYPE roman-date -> roman-month
3   (second date))

```

```

1 (defun roman-event (date)
2   ;; TYPE roman-date -> roman-event
3   (third date))

```

```

1 (defun roman-count (date)
2   ;; TYPE roman-date -> roman-count
3   (fourth date))

```

```

1 (defun roman-leap (date)
2   ;; TYPE roman-date -> roman-leap
3   (fifth date))

```

```

1 (defun ides-of-month (month)
2   ;; TYPE roman-month -> ides
3   ;; Date of Ides in Roman month.
4   (if (member month (list march may july october))
5       15
6       13))

```

(3.8)

```

1 (defun nones-of-month (month) (3.9)

```

```

2   ;; TYPE roman-month -> nones
3   ;; Date of Nones in Roman month.
4   (- (ides-of-month month) 8))

```

```

1 (defun fixed-from-roman (r-date) (3.10)

```

```

2   ;; TYPE roman-date -> fixed-date
3   ;; Fixed date for Roman name r-date.
4   (let* ((leap (roman-leap r-date))
5          (count (roman-count r-date))
6          (event (roman-event r-date))
7          (month (roman-month r-date))
8          (year (roman-year r-date)))
9     (+ (cond
10        ((= event kalends)
11         (fixed-from-julian (julian-date year month 1)))
12        ((= event nones)
13         (fixed-from-julian
14          (julian-date year month (nones-of-month month))))
15        ((= event ides)
16         (fixed-from-julian
17          (julian-date year month (ides-of-month month))))
18        (- count)
19        (if (and (julian-leap-year? year)
20                (= month march)
21                (= event kalends)
22                (>= 16 count 6)))
23          0 ; After Ides until leap day
24          1) ; Otherwise
25        (if leap
26          1 ; Leap day
27          0)))) ; Non-leap day

```

```

1  (defun roman-from-fixed (date)                                (3.11) 36      (roman-date year march kalends
2    ;; TYPE fixed-date -> roman-date                          37      (- 31 day) (= day 25))))))
3    ;; Roman name for fixed date.
4    (let* ((j-date (julian-from-fixed date))
5           (month (standard-month j-date))
6           (day (standard-day j-date))
7           (year (standard-year j-date))
8           (month-prime (amod (1+ month) 12))
9           (year-prime (if (/= month-prime 1)
10                          year
11                          (if (/= year -1)
12                              (1+ year)
13                              1))))
14      (kalends1 (fixed-from-roman
15                 (roman-date year-prime month-prime
16                             kalends 1 false))))
17  (cond
18    ((= day 1) (roman-date year month kalends 1 false))
19    ((<= day (nones-of-month month))
20     (roman-date year month nones
21                 (1+ (- (nones-of-month month) day)) false))
22    ((<= day (ides-of-month month))
23     (roman-date year month ides
24                 (1+ (- (ides-of-month month) day)) false))
25    ((or (/= month february)
26         (not (julian-leap-year? year))))
27     ;; After the Ides, in a month that is not February of a
28     ;; leap year
29     (roman-date year-prime month-prime kalends
30                 (1+ (- kalends1 date)) false))
31    ((< day 25)
32     ;; February of a leap year, before leap day
33     (roman-date year march kalends (- 30 day) false))
34    (true
35     ;; February of a leap year, on or after leap day

```

```

1  (defconstant year-rome-founded                                (3.12)
2    ;; TYPE julian-year
3    ;; Year on the Julian calendar of the founding of Rome.
4    (bce 753))
5
1  (defun julian-year-from-auc (year)                            (3.13)
2    ;; TYPE auc-year -> julian-year
3    ;; Julian year equivalent to AUC year
4    (if (<= 1 year (- year-rome-founded))
5        (+ year year-rome-founded -1)
6        (+ year year-rome-founded)))
7
1  (defun auc-year-from-julian (year)                            (3.14)
2    ;; TYPE julian-year -> auc-year
3    ;; Year AUC equivalent to Julian year
4    (if (<= year-rome-founded year -1)
5        (- year year-rome-founded -1)
6        (- year year-rome-founded)))
7
1  (defun olympiad (cycle year)
2    ;; TYPE (olympiad-cycle olympiad-year) -> olympiad
3    (list cycle year))
4
1  (defun olympiad-cycle (o-date)
2    ;; TYPE olympiad -> olympiad-cycle
3    (first o-date))
4
1  (defun olympiad-year (o-date)
2    ;; TYPE olympiad -> olympiad-year
3    (second o-date))

```

```

1 (defconstant olympiad-start (3.15) 1 (defconstant autumn (3.20)
2 ;; TYPE julian-year 2 ;; TYPE season
3 ;; Start of the Olympiads. 3 ;; Longitude of sun at autumnal equinox.
4 (bce 776)) 4 (deg 180))

1 (defun julian-year-from-olympiad (o-date) (3.16) 1 (defconstant winter (3.21)
2 ;; TYPE olympiad -> julian-year 2 ;; TYPE season
3 ;; Julian year corresponding to Olympian o-date. 3 ;; Longitude of sun at winter solstice.
4 (let* ((cycle (olympiad-cycle o-date)) 4 (deg 270))
5 (year (olympiad-year o-date))
6 (years (+ olympiad-start
7 (* 4 (1- cycle))
8 year -1)))
9 (if (< years 0)
10 years
11 (1+ years))))

1 (defun olympiad-from-julian-year (j-year) (3.17) 6 ;; longitudes and are of equal length,
2 ;; TYPE julian-year -> olympiad 7 ;; and a seasonal year started at moment start.
3 ;; Olympiad corresponding to Julian year j-year. 8 (let* ((year (gregorian-year-range g-year))
4 (let* ((years (- j-year olympiad-start 9 (pos (* (/ season (deg 360)) cap-L))
5 (if (< j-year 0) 0 1)))) 10 (cap-Delta (- pos (mod start cap-L))))
6 (olympiad (1+ (quotient years 4)) 11 (positions-in-range pos cap-L cap-Delta year)))
7 (1+ (mod years 4))))

1 (defconstant spring (3.18) 1 (defun julian-season-in-gregorian (season g-year) (3.23)
2 ;; TYPE season 2 ;; TYPE (season gregorian-year) -> list-of-moments
3 ;; Longitude of sun at vernal equinox. 3 ;; Moment(s) of Julian season in Gregorian year g-year.
4 (deg 0)) 4 (let* ((cap-Y (+ 365 (hr 6)))
5 (offset ; season start
6 (* (/ season (deg 360)) cap-Y)))
7 (cycle-in-gregorian season g-year cap-Y
8 (+ (fixed-from-julian
9 (julian-date (bce 1) march 23))
10 offset))))

```

```

1 (defun julian-in-gregorian (j-month j-day g-year) (3.24)
2   ;; TYPE (julian-month julian-day gregorian-year)
3   ;; TYPE -> list-of-fixed-dates
4   ;; List of the fixed dates of Julian month j-month, day
5   ;; j-day that occur in Gregorian year g-year.
6   (let* ((jan1 (gregorian-new-year g-year))
7          (y (standard-year (julian-from-fixed jan1)))
8          (y-prime (if (= y -1)
9                       1
10                      (1+ y)))
11          ;; The possible occurrences in one year are
12          (date0 (fixed-from-julian
13                  (julian-date y j-month j-day)))
14          (date1 (fixed-from-julian
15                  (julian-date y-prime j-month j-day))))
16     (list-range (list date0 date1)
17                 (gregorian-year-range g-year))))

1 (defun eastern-orthodox-christmas (g-year) (3.25)
2   ;; TYPE gregorian-year -> list-of-fixed-dates
3   ;; List of zero or one fixed dates of Eastern Orthodox
4   ;; Christmas in Gregorian year g-year.
5   (julian-in-gregorian december 25 g-year))

```

In languages like Lisp that allow functions as parameters, one could write a generic version of this function to collect the holidays of any given calendar and pass *fixed-from-julian* to it as an additional parameter. We have deliberately avoided this and similar advanced language features in the interests of portability.

D.4 The Coptic and Ethiopian Calendars

```

1 (defun coptic-date (year month day)
2   ;; TYPE (coptic-year coptic-month coptic-day) -> coptic-date
3   (list year month day))

```

```

1 (defconstant coptic-epoch (4.1)
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Coptic calendar.
4   (fixed-from-julian (julian-date (ce 284) august 29)))

```

```

1 (defun coptic-leap-year? (c-year) (4.2)
2   ;; TYPE coptic-year -> boolean
3   ;; True if c-year is a leap year on the Coptic calendar.
4   (= (mod c-year 4) 3))

```

```

1 (defun fixed-from-coptic (c-date) (4.3)
2   ;; TYPE coptic-date -> fixed-date
3   ;; Fixed date of Coptic date c-date.
4   (let* ((month (standard-month c-date))
5          (day (standard-day c-date))
6          (year (standard-year c-date)))
7     (+ coptic-epoch -1 ; Days before start of calendar
8        (* 365 (1- year)); Ordinary days in prior years
9        (quotient year 4); Leap days in prior years
10       (* 30 (1- month)); Days in prior months this year
11       day))) ; Days so far this month

```

```

1 (defun coptic-from-fixed (date) (4.4)
2   ;; TYPE fixed-date -> coptic-date
3   ;; Coptic equivalent of fixed date.
4   (let* ((year ; Calculate the year by cycle-of-years formula
5           (quotient (+ (* 4 (- date coptic-epoch)) 1463)
6                     1461))
7          (month; Calculate the month by division.
8           (1+ (quotient
9                (- date (fixed-from-coptic
10                        (coptic-date year 1 1))))
11              30)))

```

```

12      (day ; Calculate the day by subtraction.
13      (- date -1
14      (fixed-from-coptic
15      (coptic-date year month 1))))
16      (coptic-date year month day)))

```

```

1 (defun ethiopic-date (year month day)
2   ;; TYPE (ethiopic-year ethiopic-month ethiopic-day)
3   ;; TYPE -> ethiopic-date
4   (list year month day))

```

```

1 (defconstant ethiopic-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Ethiopic calendar.
4   (fixed-from-julian (julian-date (ce 8) august 29)))

```

```

1 (defun fixed-from-ethiopic (e-date)
2   ;; TYPE ethiopic-date -> fixed-date
3   ;; Fixed date of Ethiopic date e-date.
4   (let* ((month (standard-month e-date))
5          (day (standard-day e-date))
6          (year (standard-year e-date)))
7     (+ ethiopic-epoch
8        (- (fixed-from-coptic
9            (coptic-date year month day))
10          coptic-epoch))))

```

```

1 (defun ethiopic-from-fixed (date)
2   ;; TYPE fixed-date -> ethiopic-date
3   ;; Ethiopic equivalent of fixed date.
4   (coptic-from-fixed
5    (+ date (- coptic-epoch ethiopic-epoch))))

```

```

1 (defun coptic-in-gregorian (c-month c-day g-year)
2   ;; TYPE (coptic-month coptic-day gregorian-year)
3   ;; TYPE -> list-of-fixed-dates
4   ;; List of the fixed dates of Coptic month c-month, day
5   ;; c-day that occur in Gregorian year g-year.
6   (let* ((jan1 (gregorian-new-year g-year))
7          (y (standard-year (coptic-from-fixed jan1)))
8          ;; The possible occurrences in one year are
9          (date0 (fixed-from-coptic
10                 (coptic-date y c-month c-day)))
11          (date1 (fixed-from-coptic
12                 (coptic-date (1+ y) c-month c-day))))
13          (list-range (list date0 date1)
14                      (gregorian-year-range g-year))))

```

(4.5)

```

1 (defun coptic-christmas (g-year)
2   ;; TYPE gregorian-year -> list-of-fixed-dates
3   ;; List of zero or one fixed dates of Coptic Christmas
4   ;; in Gregorian year g-year.
5   (coptic-in-gregorian 4 29 g-year))

```

(4.6)

D.5 The ISO Calendar

```

1 (defun iso-date (year week day)
2   ;; TYPE (iso-year iso-week iso-day) -> iso-date
3   (list year week day))

```

```

1 (defun iso-week (date)
2   ;; TYPE iso-date -> iso-week
3   (second date))

```

```

1 (defun iso-day (date)
2   ;; TYPE iso-date -> day-of-week
3   (third date))

```

(4.7)

```

1 (defun iso-year (date)
2   ;; TYPE iso-date -> iso-year
3   (first date))

1 (defun fixed-from-iso (i-date)
2   ;; TYPE iso-date -> fixed-date
3   ;; Fixed date equivalent to ISO i-date.
4   (let* ((week (iso-week i-date))
5          (day (iso-day i-date))
6          (year (iso-year i-date)))
7     ;; Add fixed date of Sunday preceding date plus day
8     ;; in week.
9     (+ (nth-kday
10        week sunday
11        (gregorian-date (1- year) december 28)) day)))

```

(5.1)

```

1 (defun iso-from-fixed (date)
2   ;; TYPE fixed-date -> iso-date
3   ;; ISO (year week day) corresponding to the fixed date.
4   (let* ((approx ; Year may be one too small.
5          (gregorian-year-from-fixed (- date 3)))
6          (year (if (>= date
7                    (fixed-from-iso
8                     (iso-date (1+ approx) 1 1)))
9                    (1+ approx)
10                     approx))
11          (week (1+ (quotient
12                     (- date
13                      (fixed-from-iso (iso-date year 1 1)))
14                      7)))
15          (day (amod (- date (rd 0)) 7)))
16          (iso-date year week day)))

```

(5.2)

```

1 (defun iso-long-year? (i-year)
2   ;; TYPE iso-year -> boolean
3   ;; True if i-year is a long (53-week) year.
4   (let* ((jan1 (day-of-week-from-fixed
5                (gregorian-new-year i-year)))
6          (dec31 (day-of-week-from-fixed
7                  (gregorian-year-end i-year))))
8     (or (= jan1 thursday)
9         (= dec31 thursday))))

```

(5.3)

D.6 The Icelandic Calendar

```

1 (defun icelandic-date (year season week weekday)
2   ;; TYPE (icelandic-year icelandic-season
3   ;; TYPE icelandic-week icelandic-weekday) -> icelandic-date
4   (list year season week weekday))

1 (defun icelandic-year (i-date)
2   ;; TYPE icelandic-date -> icelandic-year
3   (first i-date))

1 (defun icelandic-season (i-date)
2   ;; TYPE icelandic-date -> icelandic-season
3   (second i-date))

1 (defun icelandic-week (i-date)
2   ;; TYPE icelandic-date -> icelandic-week
3   (third i-date))

1 (defun icelandic-weekday (i-date)
2   ;; TYPE icelandic-date -> icelandic-weekday
3   (fourth i-date))

```

```

1  (defconstant icelandic-epoch
2    ;; TYPE fixed-date
3    ;; Fixed date of start of the Icelandic calendar.
4    (fixed-from-gregorian (gregorian-date 1 april 19)))

```

```

(6.1) 14      (+ start
15             (* 7 (1- week)) ; Elapsed weeks.
16             (mod (- weekday shift) 7)))

```

```

1  (defun icelandic-summer (i-year)
2    ;; TYPE icelandic-year -> fixed-date
3    ;; Fixed date of start of Icelandic year i-year.
4    (let* ((april9 (+ icelandic-epoch (* 365 (1- i-year))
5                      (sigma ((y (to-radix i-year (list 4 25 4)))
6                                (a (list 97 24 1 0)))
7                                (* y a))))))
8      (kday-on-or-after thursday april9)))

```

```

1  (defun icelandic-from-fixed (date) (6.5)

```

```

2    ;; TYPE fixed-date -> icelandic-date
3    ;; Icelandic (year season week weekday) corresponding to
4    ;; the fixed date.
5    (let* ((approx ; approximate year
6            (quotient (- date icelandic-epoch -369)
7                      146097/400))
8            (year (if (>= date icelandic-summer approx))
9                    approx
10                   (1- approx)))
11      (season (if (< date (icelandic-winter year))
12                summer
13                winter)))
14      (start ; Start of current season.
15            (if (= season summer)
16                (icelandic-summer year)
17                (icelandic-winter year)))
18      (week ; Weeks since start of season.
19            (1+ (quotient (- date start) 7)))
20      (weekday (day-of-week-from-fixed date)))
21      (icelandic-date year season week weekday)))

```

```

1  (defun icelandic-winter (i-year)
2    ;; TYPE icelandic-year -> fixed-date
3    ;; Fixed date of start of Icelandic winter season
4    ;; in Icelandic year i-year.
5    (- (icelandic-summer (1+ i-year)) 180))

```

```

(6.3)

```

```

1  (defun fixed-from-icelandic (i-date)
2    ;; TYPE icelandic-date -> fixed-date
3    ;; Fixed date equivalent to Icelandic i-date.
4    (let* ((year (icelandic-year i-date))
5            (season (icelandic-season i-date))
6            (week (icelandic-week i-date))
7            (weekday (icelandic-weekday i-date))
8            (start ; Start of season.
9                  (if (= season summer)
10                      (icelandic-summer year)
11                      (icelandic-winter year)))
12            (shift ; First day of week in prior season.
13                  (if (= season summer) thursday saturday)))

```

```

(6.4)

```

```

1  (defun icelandic-leap-year? (i-year) (6.6)
2    ;; TYPE icelandic-year -> boolean
3    ;; True if Icelandic i-year is a leap year (53 weeks)
4    ;; on the Icelandic calendar.
5    ( /= (- (icelandic-summer (1+ i-year))
6            (icelandic-summer i-year))
7          364))

```

```

1 (defun icelandic-month (i-date)
2   ;; TYPE icelandic-date -> icelandic-month
3   ;; Month of i-date on the Icelandic calendar.
4   ;; Epagomenae are "month" 0.
5   (let* ((date (fixed-from-icelandic i-date))
6          (year (icelandic-year i-date))
7          (season (icelandic-season i-date))
8          (midsummer (- (icelandic-winter year) 90))
9          (start (cond ((= season winter)
10                       (icelandic-winter year))
11                      ((>= date midsummer)
12                       (- midsummer 90))
13                      ((< date (+ (icelandic-summer year) 90))
14                       (icelandic-summer year))
15                      (t ; Epagomenae.
16                       midsummer))))
17     (1+ (quotient (- date start) 30))))

```

D.7 The Islamic Calendar

```

1 (defun islamic-date (year month day)
2   ;; TYPE (islamic-year islamic-month islamic-day)
3   ;; TYPE -> islamic-date
4   (list year month day))

1 (defconstant islamic-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Islamic calendar.
4   (fixed-from-julian (julian-date (ce 622) july 16)))

1 (defun islamic-leap-year? (i-year)
2   ;; TYPE islamic-year -> boolean
3   ;; True if i-year is an Islamic leap year.
4   (< (mod (+ 14 (* 11 i-year)) 30) 11))

```

```

(6.7) 1 (defun fixed-from-islamic (i-date)
2   ;; TYPE islamic-date -> fixed-date
3   ;; Fixed date equivalent to Islamic date i-date.
4   (let* ((month (standard-month i-date))
5          (day (standard-day i-date))
6          (year (standard-year i-date)))
7     (+ (1- islamic-epoch) ; Days before start of calendar
8        (* (1- year) 354) ; Ordinary days since epoch.
9        (quotient ; Leap days since epoch.
10          (+ 3 (* 11 year)) 30)
11        (* 29 (1- month)) ; Days in prior months this year
12        (quotient month 2)
13        day))) ; Days so far this month.

```

```

1 (defun islamic-from-fixed (date)
2   ;; TYPE fixed-date -> islamic-date
3   ;; Islamic date (year month day) corresponding to fixed
4   ;; date.
5   (let* ((year
6          (quotient
7            (+ (* 30 (- date islamic-epoch)) 10646)
8            10631))
9          (prior-days
10           (- date (fixed-from-islamic
11                   (islamic-date year 1 1))))
12          (month
13           (quotient
14             (+ (* 11 prior-days) 330)
15             325))
16          (day
17           (1+ (- date (fixed-from-islamic
18                       (islamic-date year month 1))))))
19          (islamic-date year month day)))

```

```

1 (defun islamic-in-gregorian (i-month i-day g-year)
2   ;; TYPE (islamic-month islamic-day gregorian-year)
3   ;; TYPE -> list-of-fixed-dates
4   ;; List of the fixed dates of Islamic month i-month, day
5   ;; i-day that occur in Gregorian year g-year.
6   (let* ((jan1 (gregorian-new-year g-year))
7          (y (standard-year (islamic-from-fixed jan1)))
8          ;; The possible occurrences in one year are
9          (date0 (fixed-from-islamic
10                (islamic-date y i-month i-day)))
11          (date1 (fixed-from-islamic
12                (islamic-date (1+ y) i-month i-day)))
13          (date2 (fixed-from-islamic
14                (islamic-date (+ y 2) i-month i-day))))
15   ;; Combine in one list those that occur in current year
16   (list-range (list date0 date1 date2)
17     (gregorian-year-range g-year))))

```

(7.5)

```

1 (defconstant iyyar
2   ;; TYPE hebrew-month
3   ;; Iyyar is month number 2.
4   2)

```

(8.2)

```

1 (defconstant sivan
2   ;; TYPE hebrew-month
3   ;; Sivan is month number 3.
4   3)

```

(8.3)

```

1 (defconstant tammuz
2   ;; TYPE hebrew-month
3   ;; Tammuz is month number 4.
4   4)

```

(8.4)

```

1 (defun mawlid (g-year)
2   ;; TYPE gregorian-year -> list-of-fixed-dates
3   ;; List of fixed dates of Mawlid an-Nabi occurring in
4   ;; Gregorian year g-year.
5   (islamic-in-gregorian 3 12 g-year))

```

(7.6)

```

1 (defconstant av
2   ;; TYPE hebrew-month
3   ;; Av is month number 5.
4   5)

```

(8.5)

D.8 The Hebrew Calendar

```

1 (defun hebrew-date (year month day)
2   ;; TYPE (hebrew-year hebrew-month hebrew-day) -> hebrew-date
3   (list year month day))

```

```

1 (defconstant elul
2   ;; TYPE hebrew-month
3   ;; Elul is month number 6.
4   6)

```

(8.6)

```

1 (defconstant nisan
2   ;; TYPE hebrew-month
3   ;; Nisan is month number 1.
4   1)

```

(8.1)

```

1 (defconstant tishri
2   ;; TYPE hebrew-month
3   ;; Tishri is month number 7.
4   7)

```

(8.7)

1 (defconstant marheshvan	(8.8)	1 (defun hebrew-leap-year? (h-year)	(8.14)
2 ;; TYPE hebrew-month		2 ;; TYPE hebrew-year -> boolean	
3 ;; Marheshvan is month number 8.		3 ;; True if <i>h-year</i> is a leap year on Hebrew calendar.	
4 8)		4 (< (mod (1+ (* 7 h-year)) 19) 7))	
1 (defconstant kislev	(8.9)	1 (defun last-month-of-hebrew-year (h-year)	(8.15)
2 ;; TYPE hebrew-month		2 ;; TYPE hebrew-year -> hebrew-month	
3 ;; Kislev is month number 9.		3 ;; Last month of Hebrew year <i>h-year</i> .	
4 9)		4 (if (hebrew-leap-year? h-year)	
		5 adarii	
		6 adar))	
1 (defconstant tevet	(8.10)		
2 ;; TYPE hebrew-month		1 (defun hebrew-sabbatical-year? (h-year)	(8.16)
3 ;; Tevet is month number 10.		2 ;; TYPE hebrew-year -> boolean	
4 10)		3 ;; True if <i>h-year</i> is a sabbatical year on the Hebrew	
		4 ;; calendar.	
		5 (= (mod h-year 7) 0))	
1 (defconstant shevat	(8.11)		
2 ;; TYPE hebrew-month		1 (defconstant hebrew-epoch	(8.17)
3 ;; Shevat is month number 11.		2 ;; TYPE fixed-date	
4 11)		3 ;; Fixed date of start of the Hebrew calendar, that is,	
		4 ;; Tishri 1, 1 AM.	
		5 (fixed-from-julian (julian-date (bce 3761) october 7)))	
1 (defconstant adar	(8.12)		
2 ;; TYPE hebrew-month		1 (defun molad (h-year h-month)	(8.19)
3 ;; Adar is month number 12.		2 ;; TYPE (hebrew-year hebrew-month) -> rational-moment	
4 12)		3 ;; Moment of mean conjunction of <i>h-month</i> in Hebrew	
		4 ;; <i>h-year</i> .	
1 (defconstant adarii	(8.13)	5 (let* ((y ;; Treat Nisan as start of year.	
2 ;; TYPE hebrew-month		6 (if (< h-month tishri)	
3 ;; Adar II is month number 13.		7 (1+ h-year)	
4 13)		8 h-year))	

```

9      (months-elapsed
10      (+ (- h-month tishri) ;; Months this year.
11      (quotient ;; Months until New Year.
12      (- (* 235 y) 234)
13      19)))
14      (+ hebrew-epoch
15      -876/25920
16      (* months-elapsed (+ 29 (hr 12) 793/25920))))))

1      (defun hebrew-calendar-elapsed-days (h-year) (8.20)
2      ;; TYPE hebrew-year -> integer
3      ;; Number of days elapsed from the (Sunday) noon prior
4      ;; to the epoch of the Hebrew calendar to the mean
5      ;; conjunction (molad) of Tishri of Hebrew year h-year,
6      ;; or one day later.
7      (let* ((months-elapsed ; Since start of Hebrew calendar.
8      (quotient (- (* 235 h-year) 234) 19))
9      (parts-elapsed; Fractions of days since prior noon.
10      (+ 12084 (* 13753 months-elapsed)))
11      (days ; Whole days since prior noon.
12      (+ (* 29 months-elapsed)
13      (quotient parts-elapsed 25920)))
14      ;; If (* 13753 months-elapsed) causes integers that
15      ;; are too large, use instead:
16      ;; (parts-elapsed
17      ;; (+ 204 (* 793 (mod months-elapsed 1080))))
18      ;; (hours-elapsed
19      ;; (+ 11 (* 12 months-elapsed)
20      ;; (* 793 (quotient months-elapsed 1080)))
21      ;; (quotient parts-elapsed 1080)))
22      ;; (days
23      ;; (+ (* 29 months-elapsed)
24      ;; (quotient hours-elapsed 24)))
25      ;; If even larger integers aren't a problem, use just:
26      ;; (days

```

```

27      ;; (quotient (+ 12084 (* months-elapsed 765433))
28      ;; 25920)))
29      )
30      (if (< (mod (* 3 (1+ days)) 7) 3); Sun, Wed, or Fri
31      (+ days 1) ; Delay one day.
32      days)))

```

```

1      (defun hebrew-year-length-correction (h-year) (8.21)
2      ;; TYPE hebrew-year -> 0-2
3      ;; Delays to start of Hebrew year h-year to keep ordinary
4      ;; year in range 353-356 and leap year in range 383-386.
5      (let* ((ny0 (hebrew-calendar-elapsed-days (1- h-year)))
6      (ny1 (hebrew-calendar-elapsed-days h-year))
7      (ny2 (hebrew-calendar-elapsed-days (1+ h-year))))
8      (cond
9      ((= (- ny2 ny1) 356) ; Next year would be too long.
10      2)
11      ((= (- ny1 ny0) 382) ; Previous year too short.
12      1)
13      (t 0))))

```

```

1      (defun hebrew-new-year (h-year) (8.22)
2      ;; TYPE hebrew-year -> fixed-date
3      ;; Fixed date of Hebrew new year h-year.
4      (+ hebrew-epoch
5      (hebrew-calendar-elapsed-days h-year)
6      (hebrew-year-length-correction h-year)))

```

```

1      (defun last-day-of-hebrew-month (h-year h-month) (8.23)
2      ;; TYPE (hebrew-year hebrew-month) -> hebrew-day
3      ;; Last day of month h-month in Hebrew year h-year.
4      (if (or (member h-month
5      (list iyyar tammuz elul tevet adarii))
6      (and (= h-month adar)
7      (not (hebrew-leap-year? h-year))))

```

```

8      (and (= h-month marheshvan)
9          (not (long-marheshvan? h-year)))
10     (and (= h-month kislev)
11         (short-kislev? h-year)))
12     29
13     30))

```

```

1 (defun long-marheshvan? (h-year)
2   ;; TYPE hebrew-year -> boolean
3   ;; True if Marheshvan is long in Hebrew year h-year.
4   (member (days-in-hebrew-year h-year) (list 355 385)))

```

```

1 (defun short-kislev? (h-year)
2   ;; TYPE hebrew-year -> boolean
3   ;; True if Kislev is short in Hebrew year h-year.
4   (member (days-in-hebrew-year h-year) (list 353 383)))

```

```

1 (defun days-in-hebrew-year (h-year)
2   ;; TYPE hebrew-year -> {353,354,355,383,384,385}
3   ;; Number of days in Hebrew year h-year.
4   (- (hebrew-new-year (1+ h-year))
5      (hebrew-new-year h-year))

```

```

1 (defun fixed-from-hebrew (h-date)
2   ;; TYPE hebrew-date -> fixed-date
3   ;; Fixed date of Hebrew date h-date.
4   (let* ((month (standard-month h-date))
5         (day (standard-day h-date))
6         (year (standard-year h-date)))
7     (+ (hebrew-new-year year)
8        day -1 ; Days so far this month.
9        (if ;; before Tishri

```

```

10      (< month tishri)
11      ;; Then add days in prior months this year before
12      ;; and after Nisan.
13      (+ (sum (last-day-of-hebrew-month year m)
14          m tishri
15          (<= m (last-month-of-hebrew-year year)))
16         (sum (last-day-of-hebrew-month year m)
17             m nisan (< m month))))
18      ;; Else add days in prior months this year
19      (sum (last-day-of-hebrew-month year m)
20          m tishri (< m month))))))

```

```

1 (defun hebrew-from-fixed (date)
2   ;; TYPE fixed-date -> hebrew-date
3   ;; Hebrew (year month day) corresponding to fixed date.
4   ;; The fraction can be approximated by 365.25.
5   (let* ((approx ; Approximate year
6         (1+
7          (quotient (- date hebrew-epoch) 35975351/98496)))
8         ;; The value 35975351/98496, the average length of
9         ;; a Hebrew year, can be approximated by 365.25
10        (year ; Search forward.
11        (final y (1- approx)
12                (<= (hebrew-new-year y) date)))
13        (start ; Starting month for search for month.
14        (if (< date (fixed-from-hebrew
15                (hebrew-date year nisan 1)))
16            tishri
17            nisan))
18        (month ; Search forward from either Tishri or Nisan.
19        (next m start
20              (<= date
21                (fixed-from-hebrew
22                 (hebrew-date
23                  year

```

```

24             m
25             (last-day-of-hebrew-month year m))))))
26         (day ; Calculate the day by subtraction.
27         (1+ (- date (fixed-from-hebrew
28                 (hebrew-date year month 1))))))
29         (hebrew-date year month day)))

```

We are using Common Lisp exact arithmetic for rationals here (and elsewhere). Without that facility, one must rephrase all quotient operations to work with integers only.

The function `hebrew-calendar-elapsed-days` is called repeatedly during the calculations, often several times for the same year. A more efficient algorithm could avoid such repetition.

```

1  (defun fixed-from-molad (moon)                                (8.29)
2    ;; TYPE duration -> fixed-date
3    ;; Fixed date of the molad that occurs moon days
4    ;; and fractional days into the week.
5    (let* ((r (mod (- (* 74377 moon) 2879/2160) 7)))
6      (fixed-from-moment
7        (+ (molad 1 tishri) (* r 765433)))))

```

(This latter function requires 64-bit integers.)

```

1  (defun yom-kippur (g-year)                                    (8.30)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Yom Kippur occurring in Gregorian year
4    ;; g-year.
5    (let* ((h-year
6            (1+ (- g-year
7                  (gregorian-year-from-fixed
8                    hebrew-epoch)))))
9      (fixed-from-hebrew (hebrew-date h-year tishri 10)))

```

```

1  (defun passover (g-year)                                     (8.31)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Passover occurring in Gregorian year
4    ;; g-year.

```

```

5    (let* ((h-year
6            (- g-year
7              (gregorian-year-from-fixed hebrew-epoch))))
8      (fixed-from-hebrew (hebrew-date h-year nisan 15)))

```

```

1  (defun omer (date)                                           (8.32)
2    ;; TYPE fixed-date -> omer-count
3    ;; Number of elapsed weeks and days in the omer at date.
4    ;; Returns bogus if that date does not fall during the
5    ;; omer.
6    (let* ((c (- date
7                  (passover
8                    (gregorian-year-from-fixed date)))))
9      (if (<= 1 c 49)
10         (list (quotient c 7) (mod c 7))
11         bogus)))

```

```

1  (defun purim (g-year)                                         (8.33)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Purim occurring in Gregorian year g-year.
4    (let* ((h-year
5            (- g-year
6              (gregorian-year-from-fixed hebrew-epoch)))
7          (last-month ; Adar or Adar II
8            (last-month-of-hebrew-year h-year)))
9      (fixed-from-hebrew
10        (hebrew-date h-year last-month 14)))

```

```

1  (defun ta-anit-esther (g-year)                                (8.34)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Ta'anit Esther occurring in
4    ;; Gregorian year g-year.
5    (let* ((purim-date (purim g-year)))

```

```

6      (if ; Purim is on Sunday
7        (= (day-of-week-from-fixed purim-date) sunday)
8          ;; Then prior Thursday
9          (- purim-date 3)
10         ;; Else previous day
11         (1- purim-date))))

```

```

1  (defun tishah-be-av (g-year)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Tishah be-Av occurring in
4    ;; Gregorian year g-year.
5    (let* ((h-year ; Hebrew year
6            (- g-year
7              (gregorian-year-from-fixed hebrew-epoch)))
8           (av9
9             (fixed-from-hebrew
10              (hebrew-date h-year av 9))))
11      (if ; Ninth of Av is Saturday
12        (= (day-of-week-from-fixed av9) saturday)
13          ;; Then the next day
14          (1+ av9)
15          av9)))

```

```

1  (defun yom-ha-zikkaron (g-year)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Yom ha-Zikkaron occurring in Gregorian
4    ;; year g-year.
5    (let* ((h-year ; Hebrew year
6            (- g-year
7              (gregorian-year-from-fixed hebrew-epoch)))
8           (iyyar4; Ordinarily Iyyar 4
9             (fixed-from-hebrew
10              (hebrew-date h-year iyyar 4))))
11      (cond ((member (day-of-week-from-fixed iyyar4)

```

(8.35)

```

12      (list thursday friday))
13      ;; If Iyyar 4 is Thursday or Friday, then Wednesday
14      (kday-before wednesday iyyar4))
15      ;; If it's on Sunday, then Monday
16      ((= sunday (day-of-week-from-fixed iyyar4))
17       (1+ iyyar4))
18      (t iyyar4))))

```

```

1  (defun sh-ela (g-year)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of fixed dates of Sh'ela occurring in
4    ;; Gregorian year g-year.
5    (coptic-in-gregorian 3 26 g-year))

```

(8.37)

```

1  (defun birkath-ha-hama (g-year)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of fixed date of Birkath ha-Hama occurring in
4    ;; Gregorian year g-year, if it occurs.
5    (let* ((dates (coptic-in-gregorian 7 30 g-year)))
6      (if (and (not (equal dates nil))
7              (= (mod (standard-year
8                      (coptic-from-fixed (first dates)))
9                    28)
10                17)))
11          dates
12          nil)))

```

(8.38)

```

1  (defun samuel-season-in-gregorian (season g-year)
2    ;; TYPE (season gregorian-year) -> list-of-moments
3    ;; Moment(s) of season in Gregorian year g-year
4    ;; per Samuel.
5    (let* ((cap-Y (+ 365 (hr 6)))
6           (offset ; season start

```

(8.39)

```

7      (* (/ season (deg 360)) cap-Y)))
8      (cycle-in-gregorian season g-year cap-Y
9        (+ (fixed-from-hebrew
10           (hebrew-date 1 adar 21))
11           (hr 18)
12           offset))))

```

```

1  (defun alt-birkath-ha-hama (g-year)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of fixed date of Birkath ha-Hama occurring in
4    ;; Gregorian year g-year, if it occurs.
5    (let* ((cap-Y (+ 365 (hr 6))) ; year
6           (season (+ spring (* (hr 6) (/ (deg 360) cap-Y))))
7           (moments (samuel-season-in-gregorian season g-year)))
8      (if (and (not (equal moments nil))
9              (= (day-of-week-from-fixed (first moments))
10                wednesday)
11              (= (time-from-moment (first moments))
12                (hr 0))) ; midnight
13          (list (fixed-from-moment (first moments)))
14          nil)))

```

(8.40)

```

1  (defun adda-season-in-gregorian (season g-year)
2    ;; TYPE (season gregorian-year) -> list-of-moments
3    ;; Moment(s) of season in Gregorian year g-year
4    ;; per R. Adda bar Ahava.
5    (let* ((cap-Y (+ 365 (hr (+ 5 3791/4104))))
6           (offset ; season start
7                 (* (/ season (deg 360)) cap-Y)))
8      (cycle-in-gregorian season g-year cap-Y
9        (+ (fixed-from-hebrew
10           (hebrew-date 1 adar 28))
11           (hr 18)
12           offset))))

```

(8.41)

```

1  (defun hebrew-in-gregorian (h-month h-day g-year)
2    ;; TYPE (hebrew-month hebrew-day gregorian-year)
3    ;; TYPE -> list-of-fixed-dates
4    ;; List of the fixed dates of Hebrew month h-month, day
5    ;; h-day that occur in Gregorian year g-year.
6    (let* ((jan1 (gregorian-new-year g-year))
7           (y (standard-year (hebrew-from-fixed jan1)))
8           ;; The possible occurrences in one year are
9           (date0 (fixed-from-hebrew
10                  (hebrew-date y h-month h-day)))
11          (date1 (fixed-from-hebrew
12                  (hebrew-date (1+ y) h-month h-day)))
13          (date2 (fixed-from-hebrew
14                  (hebrew-date (+ y 2) h-month h-day))))
15      (list-range (list date0 date1 date2)
16                  (gregorian-year-range g-year)))

```

(8.42)

```

1  (defun hanukkah (g-year)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; Fixed date(s) of first day of Hanukkah
4    ;; occurring in Gregorian year g-year.
5    (hebrew-in-gregorian kislev 25 g-year))

```

(8.43)

```

1  (defun hebrew-birthday (birthdate h-year)
2    ;; TYPE (hebrew-date hebrew-year) -> fixed-date
3    ;; Fixed date of the anniversary of Hebrew birthdate
4    ;; occurring in Hebrew h-year.
5    (let* ((birth-day (standard-day birthdate))
6           (birth-month (standard-month birthdate))
7           (birth-year (standard-year birthdate)))
8      (if ; It's Adar in a normal Hebrew year or Adar II
9          ; in a Hebrew leap year,
10          (= birth-month (last-month-of-hebrew-year birth-year))
11          ;; Then use the same day in last month of Hebrew year.

```

(8.44)

```

12      (fixed-from-hebrew
13      (hebrew-date h-year (last-month-of-hebrew-year h-year)
14      birth-day))
15      ;; Else use the normal anniversary of the birth date,
16      ;; or the corresponding day in years without that date
17      (+ (fixed-from-hebrew
18      (hebrew-date h-year birth-month 1))
19      birth-day -1)))

```

```

1  (defun hebrew-birthday-in-gregorian (birthdate g-year)      (8.45)
2      ;; TYPE (hebrew-date gregorian-year)
3      ;; TYPE -> list-of-fixed-dates
4      ;; List of the fixed dates of Hebrew birthday
5      ;; that occur in Gregorian g-year.
6      (let* ((jan1 (gregorian-new-year g-year))
7      (y (standard-year (hebrew-from-fixed jan1)))
8      ;; The possible occurrences in one year are
9      (date0 (hebrew-birthday birthdate y))
10     (date1 (hebrew-birthday birthdate (1+ y)))
11     (date2 (hebrew-birthday birthdate (+ y 2))))
12      ;; Combine in one list those that occur in current year.
13      (list-range (list date0 date1 date2)
14      (gregorian-year-range g-year))))

```

```

1  (defun jahrzeit (death-date h-year)      (8.46)
2      ;; TYPE (hebrew-date hebrew-year) -> fixed-date
3      ;; Fixed date of the anniversary of Hebrew death-date
4      ;; occurring in Hebrew h-year.
5      (let* ((death-day (standard-day death-date))
6      (death-month (standard-month death-date))
7      (death-year (standard-year death-date)))
8      (cond
9      ;; If it's Marheshvan 30 it depends on the first
10     ;; anniversary; if that was not Marheshvan 30, use

```

```

11     ;; the day before Kislev 1.
12     ((and (= death-month marheshvan)
13     (= death-day 30)
14     (not (long-marheshvan? (1+ death-year)))))
15     (1- (fixed-from-hebrew
16     (hebrew-date h-year kislev 1))))
17     ;; If it's Kislev 30 it depends on the first
18     ;; anniversary; if that was not Kislev 30, use
19     ;; the day before Tevet 1.
20     ((and (= death-month kislev)
21     (= death-day 30)
22     (short-kislev? (1+ death-year)))))
23     (1- (fixed-from-hebrew
24     (hebrew-date h-year tevet 1))))
25     ;; If it's Adar II, use the same day in last
26     ;; month of Hebrew year (Adar or Adar II).
27     ((= death-month adarii)
28     (fixed-from-hebrew
29     (hebrew-date
30     h-year (last-month-of-hebrew-year h-year)
31     death-day))))
32     ;; If it's the 30th in Adar I and Hebrew year is not a
33     ;; Hebrew leap year (so Adar has only 29 days), use the
34     ;; last day in Shevat.
35     ((and (= death-day 30)
36     (= death-month adar)
37     (not (hebrew-leap-year? h-year)))))
38     (fixed-from-hebrew (hebrew-date h-year shevat 30)))
39     ;; In all other cases, use the normal anniversary of
40     ;; the date of death.
41     (t (+ (fixed-from-hebrew
42     (hebrew-date h-year death-month 1))
43     death-day -1))))

```

```

1  (defun jahrzeit-in-gregorian (death-date g-year)      (8.47)
2      ;; TYPE (hebrew-date gregorian-year)

```

```

3  ;; TYPE -> list-of-fixed-dates
4  ;; List of the fixed dates of death-date (yahrzeit)
5  ;; that occur in Gregorian year g-year.
6  (let* ((jan1 (gregorian-new-year g-year))
7         (y (standard-year (hebrew-from-fixed jan1)))
8         ;; The possible occurrences in one year are
9         (date0 (yahrzeit death-date y))
10        (date1 (yahrzeit death-date (1+ y)))
11        (date2 (yahrzeit death-date (+ y 2))))
12    ;; Combine in one list those that occur in current year
13    (list-range (list date0 date1 date2)
14               (gregorian-year-range g-year))))

```

```

1  (defun shift-days (l cap-Delta)
2    ;; TYPE (list-of-weekdays integer) -> list-of-weekdays
3    ;; Shift each weekday on list l by cap-Delta days
4    (if (equal 1 nil)
5        nil
6        (append (list (mod (+ (first l) cap-Delta) 7))
7                  (shift-days (rest l) cap-Delta))))

```

```

1  (defun possible-hebrew-days (h-month h-day)
2    ;; TYPE (hebrew-month hebrew-day) -> list-of-weekdays
3    ;; Possible days of week
4    (let* ((h-date0 (hebrew-date 5 nisan 1))
5           ;; leap year with full pattern
6           (h-year (if (> h-month elul) 6 5))
7           (h-date (hebrew-date h-year h-month h-day))
8           (n (- (fixed-from-hebrew h-date)
9                 (fixed-from-hebrew h-date0)))
10          (basic (list tuesday thursday saturday))
11          (extra
12            (cond
13              ((and (= h-month marheshvan) (= h-day 30))

```

```

14              nil)
15              ((and (= h-month kislev) (< h-day 30))
16                (list monday wednesday friday))
17              ((and (= h-month kislev) (= h-day 30))
18                (list monday))
19              ((member h-month (list tevet shevat))
20                (list sunday monday))
21              ((and (= h-month adar) (< h-day 30))
22                (list sunday monday))
23              (t (list sunday))))))
24    (shift-days (append basic extra) n)))

```

D.9 The Ecclesiastical Calendars

(8.49) (9.1)

```

1  (defun orthodox-easter (g-year)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Orthodox Easter in Gregorian year g-year.
4    (let* ((shifted-epact ; Age of moon for April 5.
5           (mod (+ 14 (* 11 (mod g-year 19)))
6                30))
7           (j-year (if (> g-year 0); Julian year number.
8                       g-year
9                       (1- g-year))))
10      (paschal-moon ; Day after full moon on
11                    ; or after March 21.
12                    (- (fixed-from-julian (julian-date j-year april 19))
13                       shifted-epact)))
14      ;; Return the Sunday following the Paschal moon.
15      (kday-after sunday paschal-moon)))

```

(8.50) (9.2)

```

1  (defun alt-orthodox-easter (g-year)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Alternative calculation of fixed date of Orthodox Easter
4    ;; in Gregorian year g-year.
5    (let* ((paschal-moon ; Day after full moon on

```

```

6                                     ; or after March 21.
7      (+ (* 354 g-year)
8        (* 30 (quotient (+ (* 7 g-year) 8) 19))
9        (quotient g-year 4)
10       (- (quotient g-year 19))
11       -273
12       gregorian-epoch)))
13 ;; Return the Sunday following the Paschal moon.
14 (kday-after sunday paschal-moon)))

1 (defun easter (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of Easter in Gregorian year g-year.
4   (let* ((century (1+ (quotient g-year 100)))
5          (shifted-epact      ; Age of moon for April 5...
6            (mod
7              (+ 14 (* 11 (mod g-year 19))); ...by Nicaean rule
8              (- ;...corrected for the Gregorian century rule
9                (quotient (* 3 century) 4))
10              (quotient; ...corrected for Metonic
11                ; cycle inaccuracy.
12                (+ 5 (* 8 century)) 25))
13              30))
14   (adjusted-epact      ; Adjust for 29.5 day month.
15     (if (or (= shifted-epact 0)
16             (and (= shifted-epact 1)
17                   (< 10 (mod g-year 19)))))
18       (1+ shifted-epact)
19       shifted-epact))
20   (paschal-moon; Day after full moon on
21     ; or after March 21.
22     (- (fixed-from-gregorian
23         (gregorian-date g-year april 19))
24        adjusted-epact)))
25 ;; Return the Sunday following the Paschal moon.
26 (kday-after sunday paschal-moon)))

```

(9.3)

```

1 (defun pentecost (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of Pentecost in Gregorian year g-year.
4   (+ (easter g-year) 49))

```

(9.4)

D.10 The Old Hindu Calendars

```

1 (defconstant hindu-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Hindu calendar (Kali Yuga).
4   (fixed-from-julian (julian-date (bce 3102) february 18)))

```

(10.1)

```

1 (defun hindu-day-count (date)
2   ;; TYPE fixed-date -> integer
3   ;; Elapsed days (Ahargana) to date since Hindu epoch (KY).
4   (- date hindu-epoch))

```

(10.2)

```

1 (defconstant arya-solar-year
2   ;; TYPE rational
3   ;; Length of Old Hindu solar year.
4   1577917500/4320000)

```

(10.3)

```

1 (defconstant arya-jovian-period
2   ;; TYPE rational
3   ;; Number of days in one revolution of Jupiter around the
4   ;; Sun.
5   1577917500/364224)

```

(10.4)

```

1 (defun jovian-year (date)
2   ;; TYPE fixed-date -> 1-60
3   ;; Year of Jupiter cycle at fixed date.
4   (amod (+ 27 (quotient (hindu-day-count date)
5                          (/ arya-jovian-period 12))))
6   60))

```

(10.5)

```

1 (defconstant arya-solar-month (10.6) 1 (defun old-hindu-lunar-month (date)
2 ;; TYPE rational 2 ;; TYPE old-hindu-lunar-date -> old-hindu-lunar-month
3 ;; Length of Old Hindu solar month. 3 (second date))
4 (/ arya-solar-year 12))

1 (defun fixed-from-old-hindu-solar (s-date) (10.7) 1 (defun old-hindu-lunar-leap (date)
2 ;; TYPE hindu-solar-date -> fixed-date 2 ;; TYPE old-hindu-lunar-date -> old-hindu-lunar-leap
3 ;; Fixed date corresponding to Old Hindu solar date s-date. 3 (third date))
4 (let* ((month (standard-month s-date))
5 (day (standard-day s-date))
6 (year (standard-year s-date)))
7 (ceiling
8 (+ hindu-epoch ; Since start of era.
9 (* year arya-solar-year) ; Days in elapsed years
10 (* (1- month) arya-solar-month) ; ...in months.
11 day (hr -30)))) ; Midnight of day.

1 (defun old-hindu-solar-from-fixed (date) (10.8) 1 (defun old-hindu-lunar-year (date)
2 ;; TYPE fixed-date -> hindu-solar-date 2 ;; TYPE old-hindu-lunar-date -> old-hindu-lunar-year
3 ;; Old Hindu solar date equivalent to fixed date. 3 (first date))
4 (let* ((sun ; Sunrise on Hindu date.
5 (+ (hindu-day-count date) (hr 6)))
6 (year ; Elapsed years.
7 (quotient sun arya-solar-year))
8 (month (1+ (mod (quotient sun arya-solar-month)
9 12)))
10 (day (1+ (floor (mod sun arya-solar-month)))))
11 (hindu-solar-date year month day))

1 (defun old-hindu-lunar-date (year month leap day)
2 ;; TYPE (old-hindu-lunar-year old-hindu-lunar-month
3 ;; TYPE old-hindu-lunar-leap old-hindu-lunar-day)
4 ;; TYPE -> old-hindu-lunar-date
5 (list year month leap day))

```

(10.9)

(10.10)

```

1 (defun old-hindu-lunar-leap-year? (l-year)
2   ;; TYPE old-hindu-lunar-year -> boolean
3   ;; True if l-year is a leap year on the
4   ;; old Hindu calendar.
5   (>= (mod (- (* l-year arya-solar-year)
6             arya-solar-month)
7         arya-lunar-month)
8       23902504679/1282400064))

1 (defun old-hindu-lunar-from-fixed (date)
2   ;; TYPE fixed-date -> old-hindu-lunar-date
3   ;; Old Hindu lunar date equivalent to fixed date.
4   (let* ((sun ; Sunrise on Hindu date.
5          (+ (hindu-day-count date) (hr 6)))
6          (new-moon ; Beginning of lunar month.
7            (- sun (mod sun arya-lunar-month)))
8          (leap ; If lunar contained in solar.
9              (and (>= (- arya-solar-month arya-lunar-month)
10                     (mod new-moon arya-solar-month))
11                 (> (mod new-moon arya-solar-month) 0)))
12          (month ; Next solar month's name.
13              (1+ (mod (ceiling (/ new-moon
14                                arya-solar-month))
15                      12)))
16          (day ; Lunar days since beginning of lunar month.
17              (1+ (mod (quotient sun arya-lunar-day) 30)))
18          (year ; Solar year at end of lunar month(s).
19              (1- (ceiling (/ (+ new-moon arya-solar-month)
20                             arya-solar-year))))
21          (old-hindu-lunar-date year month leap day)))

1 (defun fixed-from-old-hindu-lunar (l-date)
2   ;; TYPE old-hindu-lunar-date -> fixed-date
3   ;; Fixed date corresponding to Old Hindu lunar date

```

```

(10.11) 4 ;; l-date.
5 (let* ((year (old-hindu-lunar-year l-date))
6        (month (old-hindu-lunar-month l-date))
7        (leap (old-hindu-lunar-leap l-date))
8        (day (old-hindu-lunar-day l-date))
9        (mina ; One solar month before solar new year.
10             (* (1- (* 12 year)) arya-solar-month))
11        (lunar-new-year ; New moon after mina.
12                       (* arya-lunar-month
13                          (1+ (quotient mina arya-lunar-month)))))
14 (ceiling
15  (+ hindu-epoch
16     lunar-new-year
17     (* arya-lunar-month
18        (if ; If there was a leap month this year.
19            (and (not leap)
20                 (<= (ceiling (/ (- lunar-new-year mina)
21                                (- arya-solar-month
22                                   arya-lunar-month)))
23                    month))
24         month
25         (1- month))))
26  (* (1- day) arya-lunar-day) ; Lunar days.
27  (hr -6)))) ; Subtract 1 if phase begins before
28              ; sunrise.

```

D.11 The Mayan Calendars

```

1 (defun mayan-long-count-date (baktun katun tun uinal kin)
2   ;; TYPE (mayan-baktun mayan-katun mayan-tun mayan-uinal
3   ;; TYPE mayan-kin) -> mayan-long-count-date
4   (list baktun katun tun uinal kin))

(10.14) 1 (defun mayan-baktun (date)
2         ;; TYPE mayan-long-count-date -> mayan-baktun
3         (first date))

```

```

1 (defun mayan-katun (date)
2   ;; TYPE mayan-long-count-date -> mayan-katun
3   (second date))

```

```

1 (defun mayan-tun (date)
2   ;; TYPE mayan-long-count-date -> mayan-tun
3   (third date))

```

```

1 (defun mayan-uinal (date)
2   ;; TYPE mayan-long-count-date -> mayan-uinal
3   (fourth date))

```

```

1 (defun mayan-kin (date)
2   ;; TYPE mayan-long-count-date -> mayan-kin
3   (fifth date))

```

```

1 (defconstant mayan-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Mayan calendar, according
4   ;; to the Goodman-Martinez-Thompson correlation.
5   ;; That is, August 11, -3113.
6   (fixed-from-jd 584283))

```

(11.1)

```

1 (defun fixed-from-mayan-long-count (count)
2   ;; TYPE mayan-long-count-date -> fixed-date
3   ;; Fixed date corresponding to the Mayan long count,
4   ;; which is a list (baktun katun tun uinal kin).
5   (+ mayan-epoch      ; Fixed date at Mayan 0.0.0.0.0
6      (from-radix count (list 20 20 18 20))))

```

(11.2)

```

1 (defun mayan-long-count-from-fixed (date)
2   ;; TYPE fixed-date -> mayan-long-count-date
3   ;; Mayan long count date of fixed date.
4   (to-radix (- date mayan-epoch) (list 20 20 18 20)))

```

(11.3)

```

1 (defun mayan-haab-date (month day)
2   ;; TYPE (mayan-haab-month mayan-haab-day) -> mayan-haab-date
3   (list month day))

```

```

1 (defun mayan-haab-day (date)
2   ;; TYPE mayan-haab-date -> mayan-haab-day
3   (second date))

```

```

1 (defun mayan-haab-month (date)
2   ;; TYPE mayan-haab-date -> mayan-haab-month
3   (first date))

```

```

1 (defun mayan-haab-ordinal (h-date)
2   ;; TYPE mayan-haab-date -> nonnegative-integer
3   ;; Number of days into cycle of Mayan haab date h-date.
4   (let* ((day (mayan-haab-day h-date))
5          (month (mayan-haab-month h-date)))
6     (+ (* (1- month) 20) day)))

```

(11.4)

```

1 (defconstant mayan-haab-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of haab cycle.
4   (- mayan-epoch
5      (mayan-haab-ordinal (mayan-haab-date 18 8))))

```

(11.5)

```

1 (defun mayan-haab-from-fixed (date)
2   ;; TYPE fixed-date -> mayan-haab-date
3   ;; Mayan haab date of fixed date.
4   (let* ((count
5           (mod (- date mayan-haab-epoch) 365))
6           (day (mod count 20))
7           (month (1+ (quotient count 20))))
8     (mayan-haab-date month day)))

1 (defun mayan-haab-on-or-before (haab date)
2   ;; TYPE (mayan-haab-date fixed-date) -> fixed-date
3   ;; Fixed date of latest date on or before fixed date
4   ;; that is Mayan haab date haab.
5   (mod3 (+ (mayan-haab-ordinal haab) mayan-haab-epoch)
6     date (- date 365)))

1 (defun mayan-tzolkin-date (number name)
2   ;; TYPE (mayan-tzolkin-number mayan-tzolkin-name)
3   ;; TYPE -> mayan-tzolkin-date
4   (list number name))

1 (defun mayan-tzolkin-number (date)
2   ;; TYPE mayan-tzolkin-date -> mayan-tzolkin-number
3   (first date))

1 (defun mayan-tzolkin-name (date)
2   ;; TYPE mayan-tzolkin-date -> mayan-tzolkin-name
3   (second date))

1 (defconstant mayan-tzolkin-epoch
2   ;; TYPE fixed-date
3   ;; Start of tzolkin date cycle.
4   (- mayan-epoch
5     (mayan-tzolkin-ordinal (mayan-tzolkin-date 4 20))))

```

(11.6)

(11.7)

(11.8)

```

1 (defun mayan-tzolkin-from-fixed (date)
2   ;; TYPE fixed-date -> mayan-tzolkin-date
3   ;; Mayan tzolkin date of fixed date.
4   (let* ((count (- date mayan-tzolkin-epoch -1))
5           (number (amod count 13))
6           (name (amod count 20)))
7     (mayan-tzolkin-date number name)))

1 (defun mayan-tzolkin-ordinal (t-date)
2   ;; TYPE mayan-tzolkin-date -> nonnegative-integer
3   ;; Number of days into Mayan tzolkin cycle of t-date.
4   (let* ((number (mayan-tzolkin-number t-date))
5           (name (mayan-tzolkin-name t-date)))
6     (mod (+ number -1
7             (* 39 (- number name)))
8       260)))

1 (defun mayan-tzolkin-on-or-before (tzolkin date)
2   ;; TYPE (mayan-tzolkin-date fixed-date) -> fixed-date
3   ;; Fixed date of latest date on or before fixed date
4   ;; that is Mayan tzolkin date tzolkin.
5   (mod3 (+ (mayan-tzolkin-ordinal tzolkin) mayan-tzolkin-epoch)
6     date (- date 260)))

1 (defun mayan-year-bearer-from-fixed (date)
2   ;; TYPE fixed-date -> mayan-tzolkin-name
3   ;; Year bearer of year containing fixed date.
4   ;; Returns bogus for uayeb.
5   (let* ((x (mayan-haab-on-or-before
6             (mayan-haab-date 1 0)
7             date)))
8     (if (= (mayan-haab-month (mayan-haab-from-fixed date))
9           19)
10        bogus
11        (mayan-tzolkin-name (mayan-tzolkin-from-fixed x)))))

```

(11.9)

(11.10)

(11.11)

(11.12)

```

1 (defun mayan-calendar-round-on-or-before (haab tzolkin date) (11.13)
2   ;; TYPE (mayan-haab-date mayan-tzolkin-date fixed-date)
3   ;; TYPE -> fixed-date
4   ;; Fixed date of latest date on or before date, that is
5   ;; Mayan haab date haab and tzolkin date tzolkin.
6   ;; Returns bogus for impossible combinations.
7   (let* ((haab-count
8          (+ (mayan-haab-ordinal haab) mayan-haab-epoch))
9          (tzolkin-count
10         (+ (mayan-tzolkin-ordinal tzolkin)
11            mayan-tzolkin-epoch))
12         (diff (- tzolkin-count haab-count)))
13     (if (= (mod diff 5) 0)
14         (mod3 (+ haab-count (* 365 diff))
15              date (- date 18980))
16         bogus)); haab-tzolkin combination is impossible.

1 (defconstant aztec-correlation (11.14)
2   ;; TYPE fixed-date
3   ;; Known date of Aztec cycles (Caso's correlation)
4   (fixed-from-julian (julian-date 1521 August 13)))

1 (defun aztec-xihuitl-date (month day)
2   ;; TYPE (aztec-xihuitl-month aztec-xihuitl-day) ->
3   ;; TYPE aztec-xihuitl-date
4   (list month day))

1 (defun aztec-xihuitl-month (date)
2   ;; TYPE aztec-xihuitl-date -> aztec-xihuitl-month
3   (first date))

1 (defun aztec-xihuitl-day (date)
2   ;; TYPE aztec-xihuitl-date -> aztec-xihuitl-day
3   (second date))

1 (defun aztec-xihuitl-ordinal (x-date) (11.15)
2   ;; TYPE aztec-xihuitl-date -> nonnegative-integer
3   ;; Number of elapsed days into cycle of Aztec xihuitl x-date.
4   (let* ((day (aztec-xihuitl-day x-date))
5          (month (aztec-xihuitl-month x-date)))
6     (+ (* (1- month) 20) (1- day))))

1 (defconstant aztec-xihuitl-correlation (11.16)
2   ;; TYPE fixed-date
3   ;; Start of a xihuitl cycle.
4   (- aztec-correlation
5      (aztec-xihuitl-ordinal (aztec-xihuitl-date 11 2))))

1 (defun aztec-xihuitl-from-fixed (date) (11.17)
2   ;; TYPE fixed-date -> aztec-xihuitl-date
3   ;; Aztec xihuitl date of fixed date.
4   (let* ((count (mod (- date aztec-xihuitl-correlation) 365))
5          (day (1+ (mod count 20)))
6          (month (1+ (quotient count 20))))
7     (aztec-xihuitl-date month day))

1 (defun aztec-xihuitl-on-or-before (xihuitl date) (11.18)
2   ;; TYPE (aztec-xihuitl-date fixed-date) -> fixed-date
3   ;; Fixed date of latest date on or before fixed date
4   ;; that is Aztec xihuitl date xihuitl.
5   (mod3 (+ aztec-xihuitl-correlation
6            (aztec-xihuitl-ordinal xihuitl))
7          date (- date 365)))

```

```

1 (defun aztec-tonalpohualli-date (number name)
2   ;; TYPE (aztec-tonalpohualli-number aztec-tonalpohualli-name)
3   ;; TYPE -> aztec-tonalpohualli-date
4   (list number name))

1 (defun aztec-tonalpohualli-number (date)
2   ;; TYPE aztec-tonalpohualli-date -> aztec-tonalpohualli-number
3   (first date))

1 (defun aztec-tonalpohualli-name (date)
2   ;; TYPE aztec-tonalpohualli-date -> aztec-tonalpohualli-name
3   (second date))

1 (defun aztec-tonalpohualli-ordinal (t-date) (11.19)
2   ;; TYPE aztec-tonalpohualli-date -> nonnegative-integer
3   ;; Number of days into Aztec tonalpohualli cycle of t-date.
4   (let* ((number (aztec-tonalpohualli-number t-date))
5          (name (aztec-tonalpohualli-name t-date)))
6     (mod (+ number -1
7            (* 39 (- number name)))
8          260)))

1 (defconstant aztec-tonalpohualli-correlation (11.20)
2   ;; TYPE fixed-date
3   ;; Start of a tonalpohualli date cycle.
4   (- aztec-correlation
5      (aztec-tonalpohualli-ordinal
6       (aztec-tonalpohualli-date 1 5))))

1 (defun aztec-tonalpohualli-from-fixed (date) (11.21)
2   ;; TYPE fixed-date -> aztec-tonalpohualli-date
3   ;; Aztec tonalpohualli date of fixed date.
4   (let* ((count (- date aztec-tonalpohualli-correlation -1))
5          (number (amod count 13))
6          (name (amod count 20)))
7     (aztec-tonalpohualli-date number name)))

1 (defun aztec-tonalpohualli-on-or-before (tonalpohualli date) (11.22)
2   ;; TYPE (aztec-tonalpohualli-date fixed-date) -> fixed-date
3   ;; Fixed date of latest date on or before fixed date
4   ;; that is Aztec tonalpohualli date tonalpohualli.
5   (mod3 (+ aztec-tonalpohualli-correlation
6            (aztec-tonalpohualli-ordinal tonalpohualli))
7         date (- date 260)))

1 (defun aztec-xiuhmolpilli-designation (number name)
2   ;; TYPE (aztec-xiuhmolpilli-number aztec-xiuhmolpilli-name)
3   ;; TYPE -> aztec-xiuhmolpilli-designation
4   (list number name))

1 (defun aztec-xiuhmolpilli-number (date)
2   ;; TYPE aztec-xiuhmolpilli-designation -> aztec-xiuhmolpilli-number
3   (first date))

1 (defun aztec-xiuhmolpilli-name (date)
2   ;; TYPE aztec-xiuhmolpilli-designation -> aztec-xiuhmolpilli-name
3   (second date))

1 (defun aztec-xiuhmolpilli-from-fixed (date) (11.23)
2   ;; TYPE fixed-date -> aztec-xiuhmolpilli-designation
3   ;; Designation of year containing fixed date.
4   ;; Returns bogus for nemontemi.

```

```

5      (let* ((x (aztec-xihuitl-on-or-before
6                (aztec-xihuitl-date 18 20)
7                (+ date 364)))
8              (month (aztec-xihuitl-month
9                      (aztec-xihuitl-from-fixed date))))
10      (if (= month 19)
11          bogus
12          (aztec-tonalpohualli-from-fixed x))))

1      (defun aztec-xihuitl-tonalpohualli-on-or-before (11.24)
2        (xihuitl tonalpohualli date)
3        ;; TYPE (aztec-xihuitl-date aztec-tonalpohualli-date
4        ;; TYPE fixed-date) -> fixed-date
5        ;; Fixed date of latest xihuitl-tonalpohualli combination
6        ;; on or before date. That is the date on or before
7        ;; date that is Aztec xihuitl date xihuitl and
8        ;; tonalpohualli date tonalpohualli.
9        ;; Returns bogus for impossible combinations.
10      (let* ((xihuitl-count
11              (+ (aztec-xihuitl-ordinal xihuitl)
12                aztec-xihuitl-correlation))
13              (tonalpohualli-count
14              (+ (aztec-tonalpohualli-ordinal tonalpohualli)
15                aztec-tonalpohualli-correlation))
16              (diff (- tonalpohualli-count xihuitl-count)))
17      (if (= (mod diff 5) 0)
18          (mod3 (+ xihuitl-count (* 365 diff))
19              date (- date 18980))
20          bogus)); xihuitl-tonalpohualli combination is impossible.

1      (defun bali-luang (b-date)
2        ;; TYPE balinese-date -> boolean
3        (first b-date))

1      (defun bali-dwiwara (b-date)
2        ;; TYPE balinese-date -> 1-2
3        (second b-date))

1      (defun bali-triwara (b-date)
2        ;; TYPE balinese-date -> 1-3
3        (third b-date))

1      (defun bali-caturwara (b-date)
2        ;; TYPE balinese-date -> 1-4
3        (fourth b-date))

1      (defun bali-pancawara (b-date)
2        ;; TYPE balinese-date -> 1-5
3        (fifth b-date))

1      (defun bali-sadwara (b-date)
2        ;; TYPE balinese-date -> 1-6
3        (sixth b-date))

```

D.12 The Balinese Pawukon Calendar

```

1      (defun balinese-date (b1 b2 b3 b4 b5 b6 b7 b8 b9 b0)
2        ;; TYPE (boolean 1-2 1-3 1-4 1-5 1-6 1-7 1-8 1-9 0-9)
3        ;; TYPE -> balinese-date
4        (list b1 b2 b3 b4 b5 b6 b7 b8 b9 b0))

1      (defun bali-saptawara (b-date)
2        ;; TYPE balinese-date -> 1-7
3        (seventh b-date))

```

```

1 (defun bali-asatawara (b-date)
2   ;; TYPE balinese-date -> 1-8
3   (eighth b-date))

1 (defun bali-sangawara (b-date)
2   ;; TYPE balinese-date -> 1-9
3   (ninth b-date))

1 (defun bali-dasawara (b-date)
2   ;; TYPE balinese-date -> 0-9
3   (tenth b-date))

1 (defun bali-pawukon-from-fixed (date)
2   ;; TYPE fixed-date -> balinese-date
3   ;; Positions of date in ten cycles of Balinese Pawukon
4   ;; calendar.
5   (balinese-date (bali-luang-from-fixed date)
6     (bali-dwiwara-from-fixed date)
7     (bali-triwara-from-fixed date)
8     (bali-caturwara-from-fixed date)
9     (bali-pancawara-from-fixed date)
10    (bali-sadwara-from-fixed date)
11    (bali-saptawara-from-fixed date)
12    (bali-asatawara-from-fixed date)
13    (bali-sangawara-from-fixed date)
14    (bali-dasawara-from-fixed date)))

1 (defconstant bali-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of a Balinese Pawukon cycle.
4   (fixed-from-jd 146))

1 (defun bali-day-from-fixed (date)
2   ;; TYPE fixed-date -> 0-209
3   ;; Position of date in 210-day Pawukon cycle.
4   (mod (- date bali-epoch) 210))

1 (defun bali-triwara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-3
3   ;; Position of date in 3-day Balinese cycle.
4   (1+ (mod (bali-day-from-fixed date) 3)))

1 (defun bali-sadwara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-6
3   ;; Position of date in 6-day Balinese cycle.
4   (1+ (mod (bali-day-from-fixed date) 6)))

1 (defun bali-saptawara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-7
3   ;; Position of date in Balinese week.
4   (1+ (mod (bali-day-from-fixed date) 7)))

1 (defun bali-pancawara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-5
3   ;; Position of date in 5-day Balinese cycle.
4   (amod (+ (bali-day-from-fixed date) 2) 5))

1 (defun bali-week-from-fixed (date)
2   ;; TYPE fixed-date -> 1-30
3   ;; Week number of date in Balinese cycle.
4   (1+ (quotient (bali-day-from-fixed date) 7)))

```

```

1 (defun bali-dasawara-from-fixed (date)
2   ;; TYPE fixed-date -> 0-9
3   ;; Position of date in 10-day Balinese cycle.
4   (let* ((i ; Position in 5-day cycle.
5          (1- (bali-pancawara-from-fixed date)))
6          (j ; Weekday.
7          (1- (bali-saptawara-from-fixed date))))
8     (mod (+ 1 (nth i (list 5 9 7 4 8)))
9          (nth j (list 5 4 3 7 8 6 9)))
10    10)))

```

```

1 (defun bali-dwiwara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-2
3   ;; Position of date in 2-day Balinese cycle.
4   (amod (bali-dasawara-from-fixed date) 2))

1 (defun bali-luang-from-fixed (date)
2   ;; TYPE fixed-date -> boolean
3   ;; Membership of date in "1-day" Balinese cycle.
4   (evenp (bali-dasawara-from-fixed date)))

```

```

1 (defun bali-sangawara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-9
3   ;; Position of date in 9-day Balinese cycle.
4   (1+ (mod (max 0
5              (- (bali-day-from-fixed date) 3))
6            9)))

```

```

1 (defun bali-asatawara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-8
3   ;; Position of date in 8-day Balinese cycle.
4   (let* ((day (bali-day-from-fixed date)))

```

```

(12.9) 5 (1+ (mod
6         (max 6
7             (+ 4 (mod (- day 70)
8                       210))))
9         8))))

```

```

1 (defun bali-caturwara-from-fixed (date)
2   ;; TYPE fixed-date -> 1-4
3   ;; Position of date in 4-day Balinese cycle.
4   (amod (bali-asatawara-from-fixed date) 4))

```

```

(12.10) 1 (defun bali-on-or-before (b-date date)
2         ;; TYPE (balinese-date fixed-date) -> fixed-date
3         ;; Last fixed date on or before date with Pawukon b-date.
4         (let* ((luang (bali-luang b-date))

```

```

5             (dwiwara (bali-dwiwara b-date))
6             (triwara (bali-triwara b-date))
7             (caturwara (bali-caturwara b-date))
8             (pancawara (bali-pancawara b-date))
9             (sadwara (bali-sadwara b-date))
10            (saptawara (bali-saptawara b-date))
11            (asatawara (bali-asatawara b-date))
12            (sangawara (bali-sangawara b-date))
13            (dasawara (bali-dasawara b-date))
14            (a5 ; Position in 5-day subcycle.
15              (1- pancawara))
16            (a6 ; Position in 6-day subcycle.
17              (1- sadwara))
18            (b7 ; Position in 7-day subcycle.
19              (1- saptawara))
20            (b35 ; Position in 35-day subcycle.
21              (mod (+ a5 14 (* 15 (- b7 a5))) 35))
22            (days ; Position in full cycle.
23              (+ a6 (* 36 (- b35 a6))))
24            (cap-Delta (bali-day-from-fixed (rd 0))))
25            (- date (mod (- (+ date cap-Delta) days) 210))))

```

```

1 (defun kajeng-keliwon (g-year)
2   ;; TYPE gregorian-year -> list-of-fixed-dates
3   ;; Occurrences of Kajeng Keliwon (9th day of each
4   ;; 15-day subcycle of Pawukon) in Gregorian year g-year.
5   (let* ((year (gregorian-year-range g-year))
6          (cap-Delta (bali-day-from-fixed (rd 0))))
7     (positions-in-range 8 15 cap-Delta year)))

```

```

1 (defun tumpek (g-year)
2   ;; TYPE gregorian-year -> list-of-fixed-dates
3   ;; Occurrences of Tumpek (14th day of Pawukon and every
4   ;; 35th subsequent day) within Gregorian year g-year.
5   (let* ((year (gregorian-year-range g-year))
6          (cap-Delta (bali-day-from-fixed (rd 0))))
7     (positions-in-range 13 35 cap-Delta year)))

```

D.13 General Cyclical Calendars

No Lisp code is included for this chapter.

D.14 Time and Astronomy

Common Lisp's built-in trigonometric functions work with radians, whereas we have used degrees. The following functions do the necessary normalization and conversions:

```

1 (defun radians-from-degrees (theta)
2   ;; TYPE real -> radian
3   ;; Convert angle theta from degrees to radians.
4   (* (mod theta 360) pi 1/180))

1 (defun degrees-from-radians (theta)
2   ;; TYPE radian -> angle
3   ;; Convert angle theta from radians to degrees.
4   (mod (/ theta pi 1/180) 360))

```

```

(12.16) 1 (defun sin-degrees (theta)
2         ;; TYPE angle -> amplitude
3         ;; Sine of theta (given in degrees).
4         (sin (radians-from-degrees theta)))

```

```

1 (defun cos-degrees (theta)
2   ;; TYPE angle -> amplitude
3   ;; Cosine of theta (given in degrees).
4   (cos (radians-from-degrees theta)))

```

```

(12.17) 1 (defun tan-degrees (theta)
2         ;; TYPE angle -> real
3         ;; Tangent of theta (given in degrees).
4         (tan (radians-from-degrees theta)))

```

```

1 (defun arctan-degrees (y x)                                     (14.7)
2   ;; TYPE (real real) -> angle
3   ;; Arctangent of y/x in degrees.
4   ;; Returns bogus if x and y are both 0.
5   (if (and (= x 0) (= y 0))
6       bogus
7       (mod
8         (if (= x 0)
9             (* (sign y) (deg 90L0))
10            (let* ((alpha (degrees-from-radians
11                        (atan (/ y x)))))
12              (if (>= x 0)
13                  alpha
14                  (+ alpha (deg 180L0))))))
15         360)))

```

```

1 (defun arcsin-degrees (x)
2   ;; TYPE amplitude -> angle
3   ;; Arcsine of x in degrees.
4   (degrees-from-radians (asin x)))

```

```

1  (defun arccos-degrees (x)
2    ;; TYPE amplitude -> angle
3    ;; Arccosine of x in degrees.
4    (degrees-from-radians (acos x)))

```

We also use the following functions to indicate units; they are also used for typesetting:

```

1  (defun hr (x)
2    ;; TYPE real -> duration
3    ;; x hours.
4    (/ x 24))

```

```

1  (defun mn (x)
2    ;; TYPE real -> duration
3    ;; x minutes.
4    (/ x 24 60))

```

```

1  (defun sec (x)
2    ;; TYPE real -> duration
3    ;; x seconds.
4    (/ x 24 60 60))

```

```

1  (defun mt (x)
2    ;; TYPE real -> distance
3    ;; x meters.
4    ;; For typesetting purposes.
5    x)

```

```

1  (defun deg (x)
2    ;; TYPE real -> angle
3    ;; TYPE list-of-reals -> list-of-angles
4    ;; x degrees.
5    ;; For typesetting purposes.
6    x)

```

```

1  (defun mins (x)
2    ;; TYPE real -> angle
3    ;; x arcminutes
4    (/ x 60))

```

```

1  (defun secs (x)
2    ;; TYPE real -> angle
3    ;; x arcseconds
4    (/ x 3600))

```

```

1  (defun angle (d m s)
2    ;; TYPE (integer integer real) -> angle
3    ;; d degrees, m arcminutes, s arcseconds.
4    (+ d (/ (+ m (/ s 60)) 60)))

```

```

1  (defun degrees-minutes-seconds (d m s)
2    ;; TYPE (degree minute real) -> angle
3    (list d m s))

```

The `deg` function is also applied to lists, to indicate that it is a list of angles.

The following allow us to specify locations and directions:

```

1  (defun location (latitude longitude elevation zone)
2    ;; TYPE (half-circle circle distance real) -> location
3    (list latitude longitude elevation zone))

```

```

1  (defun latitude (location)
2    ;; TYPE location -> half-circle
3    (first location))

```

```

1 (defun longitude (location)
2   ;; TYPE location -> circle
3   (second location))

1 (defun elevation (location)
2   ;; TYPE location -> distance
3   (third location))

1 (defun zone (location)
2   ;; TYPE location -> real
3   (fourth location))

1 (defconstant mecca
2   ;; TYPE location
3   ;; Location of Mecca.
4   (location (angle 21 25 24) (angle 39 49 24)
5             (mt 298) (hr 3)))

```

(14.3)

```

1 (defconstant jerusalem
2   ;; TYPE location
3   ;; Location of Jerusalem.
4   (location (deg 31.78L0) (deg 35.24L0) (mt 740) (hr 2)))

1 (defconstant acre
2   ;; TYPE location
3   ;; Location of Acre.
4   (location (deg 32.94L0) (deg 35.09L0) (mt 22) (hr 2)))

```

(14.4)

(14.5)

```

1 (defun direction (location focus)
2   ;; TYPE (location location) -> angle
3   ;; Angle (clockwise from North) to face focus when
4   ;; standing in location. Subject to errors near focus and
5   ;; its antipode.
6   (let* ((phi (latitude location))
7          (phi-prime (latitude focus))
8          (psi (longitude location))
9          (psi-prime (longitude focus))
10         (y (sin-degrees (- psi-prime psi)))
11         (x
12          (- (* (cos-degrees phi)
13               (tan-degrees phi-prime))
14             (* (sin-degrees phi)
15                (cos-degrees
16                 (- psi psi-prime))))))
17     (cond ((or (= x y 0) (= phi-prime (deg 90)))
18            (deg 0))
19           ((= phi-prime (deg -90))
20            (deg 180))
21           (t (arctan-degrees y x)))))

```

(14.6)

The following functions compute times:

```

1 (defun zone-from-longitude (phi)
2   ;; TYPE circle -> duration
3   ;; Difference between UT and local mean time at longitude
4   ;; phi as a fraction of a day.
5   (/ phi (deg 360)))

1 (defun universal-from-local (tee_ell location)
2   ;; TYPE (moment location) -> moment
3   ;; Universal time from local tee_ell at location.
4   (- tee_ell (zone-from-longitude (longitude location))))

```

(14.8)

(14.9)

```

1 (defun local-from-universal (tee_rom-u location)
2   ;; TYPE (moment location) -> moment
3   ;; Local time from universal tee_rom-u at location.
4   (+ tee_rom-u (zone-from-longitude (longitude location))))

```

(14.10)

```

1 (defun standard-from-universal (tee_rom-u location)
2   ;; TYPE (moment location) -> moment
3   ;; Standard time from tee_rom-u in universal time at
4   ;; location.
5   (+ tee_rom-u (zone location)))

```

(14.11)

```

1 (defun universal-from-standard (tee_rom-s location)
2   ;; TYPE (moment location) -> moment
3   ;; Universal time from tee_rom-s in standard time at
4   ;; location.
5   (- tee_rom-s (zone location)))

```

(14.12)

```

1 (defun standard-from-local (tee_ell location)
2   ;; TYPE (moment location) -> moment
3   ;; Standard time from local tee_ell at location.
4   (standard-from-universal
5    (universal-from-local tee_ell location)
6    location))

```

(14.13)

```

1 (defun local-from-standard (tee_rom-s location)
2   ;; TYPE (moment location) -> moment
3   ;; Local time from standard tee_rom-s at location.
4   (local-from-universal
5    (universal-from-standard tee_rom-s location)
6    location))

```

(14.14)

```

1 (defun ephemeris-correction (tee)
2   ;; TYPE moment -> fraction-of-day
3   ;; Dynamical Time minus Universal Time (in days) for
4   ;; moment tee. Adapted from "Astronomical Algorithms"
5   ;; by Jean Meeus, Willmann-Bell (1991) for years
6   ;; 1600-1986 and from polynomials on the NASA
7   ;; Eclipse web site for other years.
8   (let* ((year (gregorian-year-from-fixed (floor tee)))
9          (c (/ (gregorian-date-difference
10                  (gregorian-date 1900 january 1)
11                  (gregorian-date year july 1))
12                36525))
13          (c2051 (* 1/86400
14                    (+ -20 (* 32 (expt (/ (- year 1820) 100) 2))
15                      (* 0.5628L0 (- 2150 year))))))
16          (y2000 (- year 2000))
17          (c2006 (* 1/86400
18                    (poly y2000
19                      (list 62.92L0 0.32217L0 0.005589L0))))
20          (c1987 (* 1/86400
21                    (poly y2000
22                      (list 63.86L0 0.3345L0 -0.060374L0
23                          0.0017275L0
24                          0.000651814L0 0.00002373599L0))))
25          (c1900 (poly c
26                    (list -0.00002L0 0.000297L0 0.025184L0
27                        -0.181133L0 0.553040L0 -0.861938L0
28                        0.677066L0 -0.212591L0)))
29          (c1800 (poly c
30                    (list -0.000009L0 0.003844L0 0.083563L0
31                        0.865736L0
32                        4.867575L0 15.845535L0 31.332267L0
33                        38.291999L0 28.316289L0 11.636204L0
34                        2.043794L0)))
35          (y1700 (- year 1700))
36          (c1700 (* 1/86400

```

(14.15)

```

37      (poly y1700
38        (list 8.118780842L0 -0.005092142L0
39          0.003336121L0 -0.0000266484L0))))
40      (y1600 (- year 1600))
41      (c1600 (* 1/86400
42        (poly y1600
43          (list 120 -0.9808L0 -0.01532L0
44            0.000140272128L0))))
45      (y1000 (/ (- year 1000) 100L0))
46      (c500 (* 1/86400
47        (poly y1000
48          (list 1574.2L0 -556.01L0 71.23472L0 0.319781L0
49            -0.8503463L0 -0.005050998L0
50            0.0083572073L0))))
51      (y0 (/ year 100L0))
52      (c0 (* 1/86400
53        (poly y0
54          (list 10583.6L0 -1014.41L0 33.78311L0
55            -5.952053L0 -0.1798452L0 0.022174192L0
56            0.0090316521L0))))
57      (y1820 (/ (- year 1820) 100L0))
58      (other (* 1/86400
59        (poly y1820 (list -20 0 32)))))
60      (cond ((<= 2051 year 2150) c2051)
61        ((<= 2006 year 2050) c2006)
62        ((<= 1987 year 2005) c1987)
63        ((<= 1900 year 1986) c1900)
64        ((<= 1800 year 1899) c1800)
65        ((<= 1700 year 1799) c1700)
66        ((<= 1600 year 1699) c1600)
67        ((<= 500 year 1599) c500)
68        ((< -500 year 500) c0)
69        (t other))))

1      (defun dynamical-from-universal (tee_rom-u)      (14.16)
2        ;; TYPE moment -> moment

3      ;; Dynamical time at Universal moment tee_rom-u.
4      (+ tee_rom-u (ephemeris-correction tee_rom-u)))

1      (defun universal-from-dynamical (tee)      (14.17)
2        ;; TYPE moment -> moment
3        ;; Universal moment from Dynamical time tee.
4        (- tee (ephemeris-correction tee)))

1      (defun julian-centuries (tee)      (14.18)
2        ;; TYPE moment -> century
3        ;; Julian centuries since 2000 at moment tee.
4        (/ (- (dynamical-from-universal tee) j2000)
5          36525))

1      (defconstant j2000      (14.19)
2        ;; TYPE moment
3        ;; Noon at start of Gregorian year 2000.
4        (+ (hr 12L0) (gregorian-new-year 2000)))

1      (defun equation-of-time (tee)      (14.20)
2        ;; TYPE moment -> fraction-of-day
3        ;; Equation of time (as fraction of day) for moment tee.
4        ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
5        ;; Willmann-Bell, 2nd edn., 1998, p. 185.
6        (let* ((c (julian-centuries tee))
7          (lambda
8            (poly c
9              (deg (list 280.46645L0 36000.76983L0
10                0.0003032L0))))
11          (anomaly
12            (poly c
13              (deg (list 357.52910L0 35999.05030L0
14                -0.0001559L0 -0.00000048L0))))))

```

```

15      (eccentricity
16      (poly c
17      (list 0.016708617L0 -0.000042037L0
18      -0.0000001236L0)))
19      (varepsilon (obliquity tee))
20      (y (expt (tan-degrees (/ varepsilon 2)) 2))
21      (equation
22      (* (/ 1 2 pi)
23      (+ (* y (sin-degrees (* 2 lambda)))
24      (* -2 eccentricity (sin-degrees anomaly))
25      (* 4 eccentricity y (sin-degrees anomaly)
26      (cos-degrees (* 2 lambda)))
27      (* -0.5L0 y y (sin-degrees (* 4 lambda)))
28      (* -1.25L0 eccentricity eccentricity
29      (sin-degrees (* 2 anomaly))))))
30      (* (sign equation) (min (abs equation) (hr 12L0)))))

```

```

1  (defun apparent-from-local (tee_ell location)          (14.21)
2    ;; TYPE (moment location) -> moment
3    ;; Sundial time from local time tee_ell at location.
4    (+ tee_ell (equation-of-time
5    (universal-from-local tee_ell location))))

```

```

1  (defun local-from-apparent (tee location)              (14.22)
2    ;; TYPE (moment location) -> moment
3    ;; Local time from sundial time tee at location.
4    (- tee (equation-of-time (universal-from-local tee location))))

```

```

1  (defun apparent-from-universal (tee_rom-u location)   (14.23)
2    ;; TYPE (moment location) -> moment
3    ;; True (apparent) time at universal time tee at location.
4    (apparent-from-local
5    (local-from-universal tee_rom-u location)
6    location))

```

```

1  (defun universal-from-apparent (tee location)          (14.24)
2    ;; TYPE (moment location) -> moment
3    ;; Universal time from sundial time tee at location.
4    (universal-from-local
5    (local-from-apparent tee location)
6    location))

```

```

1  (defun midnight (date location)                       (14.25)
2    ;; TYPE (fixed-date location) -> moment
3    ;; Universal time of true (apparent)
4    ;; midnight of fixed date at location.
5    (universal-from-apparent date location))

```

```

1  (defun midday (date location)                         (14.26)
2    ;; TYPE (fixed-date location) -> moment
3    ;; Universal time on fixed date of midday at location.
4    (universal-from-apparent (+ date (hr 12)) location))

```

```

1  (defun sidereal-from-moment (tee)                    (14.27)
2    ;; TYPE moment -> angle
3    ;; Mean sidereal time of day from moment tee expressed
4    ;; as hour angle. Adapted from "Astronomical Algorithms"
5    ;; by Jean Meeus, Willmann-Bell, Inc., 2nd edn., 1998, p. 88.
6    (let* ((c (/ (- tee j2000) 36525)))
7      (mod (poly c
8      (deg (list 280.46061837L0
9      (* 36525 360.98564736629L0
10      0.000387933L0 -1/38710000)))
11      360)))

```

Additional solar and lunar astronomical functions are:

```

1 (defun obliquity (tee)
2   ;; TYPE moment -> angle
3   ;; Obliquity of ecliptic at moment tee.
4   (let* ((c (julian-centuries tee))
5          (+ (angle 23 26 21.448L0)
6             (poly c (list 0L0
7                       (angle 0 0 -46.8150L0)
8                       (angle 0 0 -0.00059L0)
9                       (angle 0 0 0.001813L0))))))

```

(14.28)

```

1 (defun declination (tee beta lambda)
2   ;; TYPE (moment half-circle circle) -> angle
3   ;; Declination at moment UT tee of object at
4   ;; latitude beta and longitude lambda.
5   (let* ((varepsilon (obliquity tee))
6          (arcsin-degrees (+ (* (sin-degrees beta)
7                                (cos-degrees varepsilon))
8                              (* (cos-degrees beta)
9                                (sin-degrees varepsilon))
10                             (sin-degrees lambda))))))

```

(14.29)

```

1 (defun right-ascension (tee beta lambda)
2   ;; TYPE (moment half-circle circle) -> angle
3   ;; Right ascension at moment UT tee of object at
4   ;; latitude beta and longitude lambda.
5   (let* ((varepsilon (obliquity tee))
6          (arctan-degrees ; Cannot be bogus
7            (- (* (sin-degrees lambda)
8                  (cos-degrees varepsilon))
9               (* (tan-degrees beta)
10                  (sin-degrees varepsilon)))
11          (cos-degrees lambda))))

```

(14.30)

```

1 (defconstant mean-tropical-year (14.31)
2   ;; TYPE duration
3   365.242189L0)

```

```

1 (defconstant mean-sidereal-year (14.32)
2   ;; TYPE duration
3   365.25636L0)

```

```

1 (defun solar-longitude (tee) (14.33)
2   ;; TYPE moment -> season
3   ;; Longitude of sun at moment tee.
4   ;; Adapted from "Planetary Programs and Tables from -4000
5   ;; to +2800" by Pierre Bretagnon and Jean-Louis Simon,
6   ;; Willmann-Bell, 1986.
7   (let* ((c ; moment in Julian centuries
8          (julian-centuries tee))
9          (coefficients
10            (list 403406 195207 119433 112392 3891 2819 1721
11                  660 350 334 314 268 242 234 158 132 129 114
12                  99 93 86 78 72 68 64 46 38 37 32 29 28 27 27
13                  25 24 21 21 20 18 17 14 13 13 13 12 10 10 10
14                  10))
15          (multipliers
16            (list 0.9287892L0 35999.1376958L0 35999.4089666L0
17                  35998.7287385L0 71998.20261L0 71998.4403L0
18                  36000.35726L0 71997.4812L0 32964.4678L0
19                  -19.4410L0 445267.1117L0 45036.8840L0 3.1008L0
20                  22518.4434L0 -19.9739L0 65928.9345L0
21                  9038.0293L0 3034.7684L0 33718.148L0 3034.448L0
22                  -2280.773L0 29929.992L0 31556.493L0 149.588L0
23                  9037.750L0 107997.405L0 -4444.176L0 151.771L0
24                  67555.316L0 31556.080L0 -4561.540L0
25                  107996.706L0 1221.655L0 62894.167L0
26                  31437.369L0 14578.298L0 -31931.757L0)

```

```

27      34777.243L0 1221.999L0 62894.511L0
28      -4442.039L0 107997.909L0 119.066L0 16859.071L0
29      -4.578L0 26895.292L0 -39.127L0 12297.536L0
30      90073.778L0))
31      (addends
32      (list 270.54861L0 340.19128L0 63.91854L0 331.26220L0
33            317.843L0 86.631L0 240.052L0 310.26L0 247.23L0
34            260.87L0 297.82L0 343.14L0 166.79L0 81.53L0
35            3.50L0 132.75L0 182.95L0 162.03L0 29.8L0
36            266.4L0 249.2L0 157.6L0 257.8L0 185.1L0 69.9L0
37            8.0L0 197.1L0 250.4L0 65.3L0 162.7L0 341.5L0
38            291.6L0 98.5L0 146.7L0 110.0L0 5.2L0 342.6L0
39            230.9L0 256.1L0 45.3L0 242.9L0 115.2L0 151.8L0
40            285.3L0 53.3L0 126.6L0 205.7L0 85.9L0
41            146.1L0))
42      (lambda
43      (+ (deg 282.7771834L0)
44      (* (deg 36000.76953744L0) c)
45      (* (deg 0.000005729577951308232L0)
46      (sigma ((x coefficients)
47              (y addends)
48              (z multipliers))
49      (* x (sin-degrees (+ y (* z c))))))))))
50      (mod (+ lambda (aberration tee) (nutatation tee))
51      360)))

```

```

1      (defun nutatation (tee)
2      ;; TYPE moment -> circle
3      ;; Longitudinal nutatation at moment tee.
4      (let* ((c ; moment in Julian centuries
5              (julian-centuries tee))
6      (cap-A (poly c (deg (list 124.90L0 -1934.134L0
7                              0.002063L0))))
8      (cap-B (poly c (deg (list 201.11L0 72001.5377L0
9                              0.00057L0))))))

```

(14.34)

```

10      (+ (* (deg -0.004778L0) (sin-degrees cap-A))
11      (* (deg -0.0003667L0) (sin-degrees cap-B))))))

```

```

1      (defun aberration (tee)
2      ;; TYPE moment -> circle
3      ;; Aberration at moment tee.
4      (let* ((c ; moment in Julian centuries
5              (julian-centuries tee)))
6      (- (* (deg 0.0000974L0)
7      (cos-degrees
8      (+ (deg 177.63L0) (* (deg 35999.01848L0) c))))
9      (deg 0.005575L0))))

```

(14.35)

```

1      (defun solar-longitude-after (lambda tee)
2      ;; TYPE (season moment) -> moment
3      ;; Moment UT of the first time at or after tee
4      ;; when the solar longitude will be lambda degrees.
5      (let* ((rate ; Mean days for 1 degree change.
6              (/ mean-tropical-year (deg 360)))
7      (tau ; Estimate (within 5 days).
8      (+ tee
9      (* rate
10      (mod (- lambda (solar-longitude tee)) 360))))
11      (a (max tee (- tau 5))) ; At or after tee.
12      (b (+ tau 5)))
13      (invert-angular solar-longitude lambda
14      (interval-closed a b))))

```

(14.36)

```

1      (defun season-in-gregorian (season g-year)
2      ;; TYPE (season gregorian-year) -> moment
3      ;; Moment UT of season in Gregorian year g-year.
4      (let* ((jan1 (gregorian-new-year g-year))
5      (solar-longitude-after season jan1)))

```

(14.37)

```

1  (defun precession (tee)
2    ;; TYPE moment -> angle
3    ;; Precession at moment tee using 0,0 as J2000 coordinates.
4    ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
5    ;; Willmann-Bell, 2nd edn., 1998, pp. 136-137.
6    (let* ((c (julian-centuries tee))
7           (eta (mod
8                 (poly c (list 0 (secs 47.0029L0)
9                               (secs -0.03302L0)
10                              (secs 0.000060L0)))
11            360))
12          (cap-P (mod (poly c (list (deg 174.876384L0)
13                                   (secs -869.8089L0)
14                                   (secs 0.03536L0)))
15                     360))
16          (p (mod (poly c (list 0 (secs 5029.0966L0)
17                                (secs 1.11113L0)
18                                (secs 0.000006L0)))
19                 360))
20          (cap-A (* (cos-degrees eta) (sin-degrees cap-P)))
21          (cap-B (cos-degrees cap-P))
22          (arg (arctan-degrees cap-A cap-B)))
23    (mod (- (+ p cap-P) arg) 360)))

```

```

1  (defun sidereal-solar-longitude (tee)
2    ;; TYPE moment -> angle
3    ;; Sidereal solar longitude at moment tee
4    (mod (+ (solar-longitude tee)
5           (- (precession tee)
              sidereal-start))
6         360))

```

```

1  (defun solar-altitude (tee location)
2    ;; TYPE (moment location) -> half-circle

```

(14.39)

```

3    ;; Geocentric altitude of sun at tee at location,
4    ;; as a positive/negative angle in degrees, ignoring
5    ;; parallax and refraction.
6    (let* ((phi ; Local latitude.
7           (latitude location))
8           (psi ; Local longitude.
9           (longitude location))
10          (lambda ; Solar longitude.
11          (solar-longitude tee))
12          (alpha ; Solar right ascension.
13          (right-ascension tee 0 lambda))
14          (delta ; Solar declination.
15          (declination tee 0 lambda))
16          (theta0 ; Sidereal time.
17          (sidereal-from-moment tee))
18          (cap-H ; Local hour angle.
19          (mod (- theta0 (- psi) alpha) 360))
20          (altitude
21          (arcsin-degrees (+ (* (sin-degrees phi)
22                                (sin-degrees delta))
23                              (* (cos-degrees phi)
24                                (cos-degrees delta)
25                                (cos-degrees cap-H)))))
26    (mod3 altitude -180 180)))

```

(14.40)

```

1  (defun estimate-prior-solar-longitude (lambda tee)
2    ;; TYPE (season moment) -> moment
3    ;; Approximate moment at or before tee
4    ;; when solar longitude just exceeded lambda degrees.
5    (let* ((rate ; Mean change of one degree.
6           (/ mean-tropical-year (deg 360)))
7           (tau ; First approximation.
8           (- tee
9             (* rate (mod (- (solar-longitude tee)
10                             lambda)

```

(14.42)

(14.41)

```

10          lambda)

```

```

11             360))))
12         (cap-Delta ; Difference in longitude.
13         (mod3 (- (solar-longitude tau) lambda)
14         -180 180)))
15         (min tee (- tau (* rate cap-Delta))))))

1  (defconstant mean-synodic-month          (14.44)
2    ;; TYPE duration
3    29.530588861L0)

1  (defun nth-new-moon (n)                  (14.45)
2    ;; TYPE integer -> moment
3    ;; Moment of n-th new moon after (or before) the new moon
4    ;; of January 11, 1. Adapted from "Astronomical Algorithms"
5    ;; by Jean Meeus, Willmann-Bell, corrected 2nd edn., 2005.
6    (let* ((n0 24724) ; Months from RD 0 until j2000.
7           (k (- n n0)) ; Months since j2000.
8           (c (/ k 1236.85L0)) ; Julian centuries.
9           (approx (+ j2000
10                     (poly c (list 5.09766L0
11                                 (* mean-synodic-month
12                                 1236.85L0)
13                                 0.00015437L0
14                                 -0.000000150L0
15                                 0.00000000073L0))))))
16    (cap-E (poly c (list 1 -0.002516L0 -0.0000074L0))))
17    (solar-anomaly
18    (poly c (deg (list 2.5534L0
19                      (* 1236.85L0 29.10535670L0)
20                      -0.0000014L0 -0.00000011L0))))))
21    (lunar-anomaly
22    (poly c (deg (list 201.5643L0 (* 385.81693528L0
23                                1236.85L0)
24                                0.0107582L0 0.00001238L0
25                                -0.000000058L0))))))
26    (moon-argument ; Moon's argument of latitude.
27    (poly c (deg (list 160.7108L0 (* 390.67050284L0
28                                1236.85L0)
29                                -0.0016118L0 -0.00000227L0
30                                0.000000011L0))))))
31    (cap-omega ; Longitude of ascending node.
32    (poly c (deg (list 124.7746L0 (* -1.56375588L0 1236.85L0)
33                                0.0020672L0 0.00000215L0))))))
34    (E-factor (list 0 1 0 0 1 1 2 0 0 1 0 1 1 1 0 0 0 0
35                  0 0 0 0 0 0))
36    (solar-coeff (list 0 1 0 0 -1 1 2 0 0 1 0 1 1 -1 2
37                     0 3 1 0 1 -1 -1 1 0))
38    (lunar-coeff (list 1 0 2 0 1 1 0 1 1 2 3 0 0 2 1 2
39                     0 1 2 1 1 1 3 4))
40    (moon-coeff (list 0 0 0 2 0 0 0 -2 2 0 0 2 -2 0 0
41                    -2 0 -2 2 2 2 -2 0 0))
42    (sine-coeff
43    (list -0.40720L0 0.17241L0 0.01608L0 0.01039L0
44          0.00739L0 -0.00514L0 0.00208L0
45          -0.00111L0 -0.00057L0 0.00056L0
46          -0.00042L0 0.00042L0 0.00038L0
47          -0.00024L0 -0.00007L0 0.00004L0
48          0.00004L0 0.00003L0 0.00003L0
49          -0.00003L0 0.00003L0 -0.00002L0
50          -0.00002L0 0.00002L0))
51    (correction
52    (+ (* -0.00017L0 (sin-degrees cap-omega))
53    (sigma ((v sine-coeff)
54            (w E-factor)
55            (x solar-coeff)
56            (y lunar-coeff)
57            (z moon-coeff))
58    (* v (expt cap-E w)
59    (sin-degrees
60    (+ (* x solar-anomaly)

```

```

61      (* y lunar-anomaly)
62      (* z moon-argument))))))
63      (add-const
64      (list 251.88L0 251.83L0 349.42L0 84.66L0
65            141.74L0 207.14L0 154.84L0 34.52L0 207.19L0
66            291.34L0 161.72L0 239.56L0 331.55L0))
67      (add-coeff
68      (list 0.016321L0 26.651886L0
69            36.412478L0 18.206239L0 53.303771L0
70            2.453732L0 7.306860L0 27.261239L0 0.121824L0
71            1.844379L0 24.198154L0 25.513099L0
72            3.592518L0))
73      (add-factor
74      (list 0.000165L0 0.000164L0 0.000126L0
75            0.000110L0 0.000062L0 0.000060L0 0.000056L0
76            0.000047L0 0.000042L0 0.000040L0 0.000037L0
77            0.000035L0 0.000023L0))
78      (extra
79      (* 0.000325L0
80      (sin-degrees
81      (poly c
82      (deg (list 299.77L0 132.8475848L0
83              -0.009173L0))))))
84      (additional
85      (sigma ((i add-const)
86              (j add-coeff)
87              (l add-factor)
88              (* 1 (sin-degrees (+ i (* j k)))))))
89      (universal-from-dynamical
90      (+ approx correction extra additional))))

```

```

1  (defun new-moon-before (tee)
2    ;; TYPE moment -> moment
3    ;; Moment UT of last new moon before tee.
4    (let* ((t0 (nth-new-moon 0))

```

(14.46)

```

5      (phi (lunar-phase tee))
6      (n (round (- (/ (- tee t0) mean-synodic-month)
7                  (/ phi (deg 360))))))
8      (nth-new-moon (final k (1- n) (< (nth-new-moon k) tee))))))

1  (defun new-moon-at-or-after (tee)
2    ;; TYPE moment -> moment
3    ;; Moment UT of first new moon at or after tee.
4    (let* ((t0 (nth-new-moon 0))
5            (phi (lunar-phase tee))
6            (n (round (- (/ (- tee t0) mean-synodic-month)
7                          (/ phi (deg 360))))))
8            (nth-new-moon (next k n (>= (nth-new-moon k) tee))))))

```

(14.47)

```

1  (defun lunar-longitude (tee)
2    ;; TYPE moment -> angle
3    ;; Longitude of moon (in degrees) at moment tee.
4    ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
5    ;; Willmann-Bell, 2nd edn., 1998, pp. 338-342.
6    (let* ((c (julian-centuries tee))
7            (cap-L-prime (mean-lunar-longitude c))
8            (cap-D (lunar-elongation c))
9            (cap-M (solar-anomaly c))
10           (cap-M-prime (lunar-anomaly c))
11           (cap-F (moon-node c))
12           (cap-E (poly c (list 1 -0.002516L0 -0.0000074L0)))
13           (args-lunar-elongation
14           (list 0 2 2 0 0 0 2 2 2 2 0 1 0 2 0 0 4 0 4 2 2 1
15                 1 2 2 4 2 0 2 2 1 2 0 0 2 2 2 2 4 0 3 2 4 0 2
16                 2 2 4 0 4 1 2 0 1 3 4 2 0 1 2)))
17           (args-solar-anomaly
18           (list 0 0 0 0 1 0 0 -1 0 -1 1 0 1 0 0 0 0 0 0 1 1
19                 0 1 -1 0 0 0 1 0 -1 0 -2 1 2 -2 0 0 -1 0 0 1
20                 -1 2 2 1 -1 0 0 -1 0 1 0 1 0 0 -1 2 1 0)))

```

(14.48)

```

21 (args-lunar-anomaly
22 (list 1 -1 0 2 0 0 -2 -1 1 0 -1 0 1 0 1 1 -1 3 -2
23       -1 0 -1 0 1 2 0 -3 -2 -1 -2 1 0 2 0 -1 1 0
24       -1 2 -1 1 -2 -1 -1 -2 0 1 4 0 -2 0 2 1 -2 -3
25       2 1 -1 3))
26 (args-moon-node
27 (list 0 0 0 0 0 2 0 0 0 0 0 0 0 -2 2 -2 0 0 0 0 0
28       0 0 0 0 0 0 2 0 0 0 0 0 0 -2 2 0 2 0 0 0 0
29       0 0 -2 0 0 0 0 -2 -2 0 0 0 0 0 0 0))
30 (sine-coeff
31 (list 6288774 1274027 658314 213618 -185116 -114332
32       58793 57066 53322 45758 -40923 -34720 -30383
33       15327 -12528 10980 10675 10034 8548 -7888
34       -6766 -5163 4987 4036 3994 3861 3665 -2689
35       -2602 2390 -2348 2236 -2120 -2069 2048 -1773
36       -1595 1215 -1110 -892 -810 759 -713 -700 691
37       596 549 537 520 -487 -399 -381 351 -340 330
38       327 -323 299 294))
39 (correction
40 (* (deg 1/1000000)
41    (sigma ((v sine-coeff)
42            (w args-lunar-elongation)
43            (x args-solar-anomaly)
44            (y args-lunar-anomaly)
45            (z args-moon-node))
46          (* v (expt cap-E (abs x))
47              (sin-degrees
48                (+ (* w cap-D)
49                   (* x cap-M)
50                   (* y cap-M-prime)
51                   (* z cap-F)))))))
52 (venus (* (deg 3958/1000000)
53          (sin-degrees
54            (+ (deg 119.75L0) (* c (deg 131.849L0))))))
55 (jupiter (* (deg 318/1000000)
56            (sin-degrees

```

```

57          (+ (deg 53.09L0)
58             (* c (deg 479264.29L0))))))
59 (flat-earth
60 (* (deg 1962/1000000)
61    (sin-degrees (- cap-L-prime cap-F))))
62 (mod (+ cap-L-prime correction venus jupiter flat-earth
63       (nutaton tee))
64      360)))

```

```
1 (defun mean-lunar-longitude (c) (14.49)
```

```

2 ;; TYPE century -> angle
3 ;; Mean longitude of moon (in degrees) at moment
4 ;; given in Julian centuries c.
5 ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
6 ;; Willmann-Bell, 2nd edn., 1998, pp. 337-340.
7 (mod
8   (poly c
9     (deg (list 218.3164477L0 481267.88123421L0
10              -0.0015786L0 1/538841 -1/65194000))))
11 360))

```

```
1 (defun lunar-elongation (c) (14.50)
```

```

2 ;; TYPE century -> angle
3 ;; Elongation of moon (in degrees) at moment
4 ;; given in Julian centuries c.
5 ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
6 ;; Willmann-Bell, 2nd edn., 1998, p. 338.
7 (mod
8   (poly c
9     (deg (list 297.8501921L0 445267.1114034L0
10              -0.0018819L0 1/545868 -1/113065000))))
11 360))

```

```
1 (defun solar-anomaly (c) (14.51)
```

```

2 ;; TYPE century -> angle
3 ;; Mean anomaly of sun (in degrees) at moment

```

```

4  ;; given in Julian centuries c.
5  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
6  ;; Willmann-Bell, 2nd edn., 1998, p. 338.
7  (mod
8    (poly c
9      (deg (list 357.5291092L0 35999.0502909L0
10               -0.0001536L0 1/24490000)))
11    360))

```

```

1  (defun lunar-anomaly (c)
2    ;; TYPE century -> angle
3    ;; Mean anomaly of moon (in degrees) at moment
4    ;; given in Julian centuries c.
5    ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
6    ;; Willmann-Bell, 2nd edn., 1998, p. 338.
7    (mod
8      (poly c
9        (deg (list 134.9633964L0 477198.8675055L0
10                 0.0087414L0 1/69699 -1/14712000)))
11      360))

```

```

1  (defun moon-node (c)
2    ;; TYPE century -> angle
3    ;; Moon's argument of latitude (in degrees) at moment
4    ;; given in Julian centuries c.
5    ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
6    ;; Willmann-Bell, 2nd edn., 1998, p. 338.
7    (mod
8      (poly c
9        (deg (list 93.2720950L0 483202.0175233L0
10                 -0.0036539L0 -1/3526000 1/863310000)))
11      360))

```

(14.52)

(14.53)

```

1  (defun lunar-node (date)
2    ;; TYPE fixed-date -> angle
3    ;; Angular distance of the lunar node from the equinoctial
4    ;; point on fixed date.
5    (mod3 (+ (moon-node (julian-centuries date)))
6           -90 90))

```

(14.54)

```

1  (defun sidereal-lunar-longitude (tee)
2    ;; TYPE moment -> angle
3    ;; Sidereal lunar longitude at moment tee.
4    (mod (+ (lunar-longitude tee)
5            (- (precession tee)
6              sidereal-start)
7            360))

```

(14.55)

```

1  (defun lunar-phase (tee)
2    ;; TYPE moment -> phase
3    ;; Lunar phase, as an angle in degrees, at moment tee.
4    ;; An angle of 0 means a new moon, 90 degrees means the
5    ;; first quarter, 180 means a full moon, and 270 degrees
6    ;; means the last quarter.
7    (let* ((phi (mod (- (lunar-longitude tee)
8                        (solar-longitude tee))
9                      360))
10           (t0 (nth-new-moon 0))
11           (n (round (/ (- tee t0) mean-synodic-month)))
12           (phi-prime (* (deg 360)
13                         (mod (/ (- tee (nth-new-moon n))
14                               mean-synodic-month)
15                               1))))
16      (if (> (abs (- phi phi-prime)) (deg 180)) ; close call
17          phi-prime
18          phi)))

```

(14.56)

```

1 (defun lunar-phase-at-or-before (phi tee) (14.57) 3 ;; Excess of lunar longitude over solar longitude at full
2 ;; TYPE (phase moment) -> moment 4 ;; moon.
3 ;; Moment UT of the last time at or before tee 5 (deg 180))
4 ;; when the lunar-phase was phi degrees.
5 (let* ((tau ; Estimate.
6 (- tee
7 (* mean-synodic-month (/ 1 (deg 360))
8 (mod (- (lunar-phase tee) phi) 360))))
9 (a (- tau 2))
10 (b (min tee (+ tau 2)))) ; At or before tee.
11 (invert-angular lunar-phase phi
12 (interval-closed a b)))

1 (defun lunar-phase-at-or-after (phi tee) (14.58) 3 ;; Excess of lunar longitude over solar longitude at last
2 ;; TYPE (phase moment) -> moment 4 ;; quarter moon.
3 ;; Moment UT of the next time at or after tee 5 (deg 270))
4 ;; when the lunar-phase is phi degrees.
5 (let* ((tau ; Estimate.
6 (+ tee
7 (* mean-synodic-month (/ 1 (deg 360))
8 (mod (- phi (lunar-phase tee)) 360))))
9 (a (max tee (- tau 2))) ; At or after tee.
10 (b (+ tau 2)))
11 (invert-angular lunar-phase phi
12 (interval-closed a b)))

1 (defconstant new (14.59) 10
2 ;; TYPE phase
3 ;; Excess of lunar longitude over solar longitude at new
4 ;; moon.
5 (deg 0))

1 (defconstant full (14.61) 17
2 ;; TYPE phase
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
259
```

```

19      0 1 0 1 1 1 0 0 0 0 0 0 0 0 -1 0 0 0 0 1 1
20      0 -1 -2 0 1 1 1 1 1 0 -1 1 0 -1 0 0 0 -1 -2))
21  (args-lunar-anomaly
22  (list 0 1 1 0 -1 -1 0 2 1 2 0 -2 1 0 -1 0 -1 -1 -1
23        0 0 -1 0 1 1 0 0 3 0 -1 1 -2 0 2 1 -2 3 2 -3
24        -1 0 0 1 0 1 1 0 0 -2 -1 1 -2 2 -2 -1 1 1 -1
25        0 0))
26  (args-moon-node
27  (list 1 1 -1 -1 1 1 -1 1 1 -1 -1 -1 -1 1 1 -1 -1
28        -1 1 3 1 1 1 -1 -1 1 -1 1 -3 1 -3 -1 -1 1
29        -1 1 -1 1 1 1 1 -1 3 -1 -1 1 -1 -1 1 -1 -1
30        -1 -1 -1 -1 -1 1))
31  (sine-coeff
32  (list 5128122 280602 277693 173237 55413 46271 32573
33        17198 9266 8822 8216 4324 4200 -3359 2463 2211
34        2065 -1870 1828 -1794 -1749 -1565 -1491 -1475
35        -1410 -1344 -1335 1107 1021 833 777 671 607
36        596 491 -451 439 422 421 -366 -351 331 315
37        302 -283 -229 223 223 -220 -220 -185 181
38        -177 176 166 -164 132 -119 115 107))
39  (beta
40  (* (deg 1/1000000)
41     (sigma ((v sine-coeff)
42             (w args-lunar-elongation)
43             (x args-solar-anomaly)
44             (y args-lunar-anomaly)
45             (z args-moon-node))
46     (* v (expt cap-E (abs x))
47        (sin-degrees
48         (+ (* w cap-D)
49            (* x cap-M)
50            (* y cap-M-prime)
51            (* z cap-F))))))
52  (venus (* (deg 175/1000000)
53           (+ (sin-degrees
54              (+ (deg 119.75L0) (* c (deg 131.849L0))

```

```

55           cap-F))
56           (sin-degrees
57            (+ (deg 119.75L0) (* c (deg 131.849L0))
58              (- cap-F))))))
59  (flat-earth
60  (+ (* (deg -2235/1000000)
61        (sin-degrees cap-L-prime))
62     (* (deg 127/1000000) (sin-degrees
63                           (- cap-L-prime cap-M-prime)))
64     (* (deg -115/1000000) (sin-degrees
65                           (+ cap-L-prime cap-M-prime))))))
66  (extra (* (deg 382/1000000)
67            (sin-degrees
68             (+ (deg 313.45L0)
69                (* c (deg 481266.484L0))))))
70  (+ beta venus flat-earth extra))

```

```

1  (defun lunar-altitude (tee location)                                     (14.64)
2    ;; TYPE (moment location) -> half-circle
3    ;; Geocentric altitude of moon at tee at location,
4    ;; as a small positive/negative angle in degrees, ignoring
5    ;; parallax and refraction. Adapted from "Astronomical
6    ;; Algorithms" by Jean Meeus, Willmann-Bell, 2nd edn.,
7    ;; 1998.
8    (let* ((phi ; Local latitude.
9            (latitude location))
10           (psi ; Local longitude.
11              (longitude location))
12           (lambda ; Lunar longitude.
13              (lunar-longitude tee))
14           (beta ; Lunar latitude.
15              (lunar-latitude tee))
16           (alpha ; Lunar right ascension.
17              (right-ascension tee beta lambda))
18           (delta ; Lunar declination.

```

```

19      (declination tee beta lambda))
20      (theta0 ; Sidereal time.
21      (sidereal-from-moment tee))
22      (cap-H ; Local hour angle.
23      (mod (- theta0 (- psi) alpha) 360))
24      (altitude
25      (arcsin-degrees (+ (* (sin-degrees phi)
26                             (sin-degrees delta))
27                          (* (cos-degrees phi)
28                             (cos-degrees delta)
29                             (cos-degrees cap-H))))))
30      (mod3 altitude -180 180)))

1  (defun lunar-distance (tee)
2    ;; TYPE moment -> distance
3    ;; Distance to moon (in meters) at moment tee.
4    ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
5    ;; Willmann-Bell, 2nd edn., 1998, pp. 338-342.
6    (let* ((c (julian-centuries tee))
7           (cap-D (lunar-elongation c))
8           (cap-M (solar-anomaly c))
9           (cap-M-prime (lunar-anomaly c))
10          (cap-F (moon-node c))
11          (cap-E (poly c (list 1 -0.002516L0 -0.0000074L0)))
12          (args-lunar-elongation
13          (list 0 2 2 0 0 0 2 2 2 2 0 1 0 2 0 0 4 0 4 2 2 1
14                1 2 2 4 2 0 2 2 1 2 0 0 2 2 2 4 0 3 2 4 0 2
15                2 2 4 0 4 1 2 0 1 3 4 2 0 1 2 2)))
16          (args-solar-anomaly
17          (list 0 0 0 0 1 0 0 -1 0 -1 1 0 1 0 0 0 0 0 0 1 1
18                0 1 -1 0 0 0 1 0 -1 0 -2 1 2 -2 0 0 -1 0 0 1
19                -1 2 2 1 -1 0 0 -1 0 1 0 1 0 0 -1 2 1 0 0))
20          (args-lunar-anomaly
21          (list 1 -1 0 2 0 0 -2 -1 1 0 -1 0 1 0 1 1 -1 3 -2
22                -1 0 -1 0 1 2 0 -3 -2 -1 -2 1 0 2 0 -1 1 0

```

(14.65)

```

23      -1 2 -1 1 -2 -1 -1 -2 0 1 4 0 -2 0 2 1 -2 -3
24      2 1 -1 3 -1))
25      (args-moon-node
26      (list 0 0 0 0 0 2 0 0 0 0 0 0 0 0 -2 2 -2 0 0 0 0 0
27            0 0 0 0 0 0 0 2 0 0 0 0 0 0 -2 2 0 2 0 0 0 0
28            0 0 -2 0 0 0 0 -2 -2 0 0 0 0 0 0 0 -2))
29      (cosine-coeff
30      (list -20905355 -3699111 -2955968 -569925 48888 -3149
31            246158 -152138 -170733 -204586 -129620 108743
32            104755 10321 0 79661 -34782 -23210 -21636 24208
33            30824 -8379 -16675 -12831 -10445 -11650 14403
34            -7003 0 10056 6322 -9884 5751 0 -4950 4130 0
35            -3958 0 3258 2616 -1897 -2117 2354 0 0 -1423
36            -1117 -1571 -1739 0 -4421 0 0 0 0 1165 0 0
37            8752))
38      (correction
39      (sigma ((v cosine-coeff)
40              (w args-lunar-elongation)
41              (x args-solar-anomaly)
42              (y args-lunar-anomaly)
43              (z args-moon-node))
44              (* v (expt cap-E (abs x))
45                 (cos-degrees
46                 (+ (* w cap-D)
47                    (* x cap-M)
48                    (* y cap-M-prime)
49                    (* z cap-F))))))
50      (+ (mt 385000560) correction)))

```

(14.66)

```

1  (defun lunar-parallax (tee location)
2    ;; TYPE (moment location) -> angle
3    ;; Parallax of moon at tee at location.
4    ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
5    ;; Willmann-Bell, 2nd edn., 1998.
6    (let* ((geo (lunar-altitude tee location))

```

```

7      (cap-Delta (lunar-distance tee))
8      (alt (/ (mt 6378140) cap-Delta))
9      (arg (* alt (cos-degrees geo))))
10     (arcsin-degrees arg)))

```

```

1  (defun topocentric-lunar-altitude (tee location) (14.67)
2    ;; TYPE (moment location) -> half-circle
3    ;; Topocentric altitude of moon at tee at location,
4    ;; as a small positive/negative angle in degrees,
5    ;; ignoring refraction.
6    (- (lunar-altitude tee location)
7       (lunar-parallax tee location)))

```

Times of day are computed by the following functions:

```

1  (defun approx-moment-of-depression (tee location alpha early?) (14.68)
2    ;; TYPE (moment location half-circle boolean) -> moment
3    ;; Moment in local time near tee when depression angle
4    ;; of sun is alpha (negative if above horizon) at
5    ;; location; early? is true when morning event is sought
6    ;; and false for evening. Returns bogus if depression
7    ;; angle is not reached.
8    (let* ((try (sine-offset tee location alpha))
9           (date (fixed-from-moment tee))
10          (alt (if (>= alpha 0)
11                  (if early? date (1+ date))
12                  (+ date (hr 12)))))
13      (value (if (> (abs try) 1)
14                (sine-offset alt location alpha)
15                try)))
16    (if (<= (abs value) 1) ; Event occurs
17        (let* ((offset (mod3 (/ (arcsin-degrees value) (deg 360))
18                                (hr -12) (hr 12))))
19            (local-from-apparent
20              (+ date

```

```

21      (if early?
22        (- (hr 6) offset)
23        (+ (hr 18) offset)))
24      location))
25      bogus)))

```

```

1  (defun sine-offset (tee location alpha) (14.69)
2    ;; TYPE (moment location half-circle) -> real
3    ;; Sine of angle between position of sun at
4    ;; local time tee and
5    ;; when its depression is alpha at location.
6    ;; Out of range when it does not occur.
7    (let* ((phi (latitude location))
8           (tee-prime (universal-from-local tee location))
9           (delta ; Declination of sun.
10              (declination tee-prime (deg 0L0)
11                (solar-longitude tee-prime))))
12      (+ (* (tan-degrees phi)
13            (tan-degrees delta))
14         (/ (sin-degrees alpha)
15            (* (cos-degrees delta)
16               (cos-degrees phi))))))

```

```

1  (defun moment-of-depression (approx location alpha early?) (14.70)
2    ;; TYPE (moment location half-circle boolean) -> moment
3    ;; Moment in local time near approx when depression
4    ;; angle of sun is alpha (negative if above horizon) at
5    ;; location; early? is true when morning event is
6    ;; sought, and false for evening.
7    ;; Returns bogus if depression angle is not reached.
8    (let* ((tee (approx-moment-of-depression
9                approx location alpha early?)))
10      (if (equal tee bogus)
11          bogus

```

```

12      (if (< (abs (- approx tee))
13          (sec 30))
14          tee
15          (moment-of-depression tee location alpha early?))))

```

```

1  (defconstant morning
2    ;; TYPE boolean
3    ;; Signifies morning.
4    true)

```

```

1  (defun dawn (date location alpha)
2    ;; TYPE (fixed-date location half-circle) -> moment
3    ;; Standard time in morning on fixed date at
4    ;; location when depression angle of sun is alpha.
5    ;; Returns bogus if there is no dawn on date.
6    (let* ((result (moment-of-depression
7                  (+ date (hr 6)) location alpha morning)))
8      (if (equal result bogus)
9          bogus
10         (standard-from-local result location))))

```

```

1  (defconstant evening
2    ;; TYPE boolean
3    ;; Signifies evening.
4    false)

```

```

1  (defun dusk (date location alpha)
2    ;; TYPE (fixed-date location half-circle) -> moment
3    ;; Standard time in evening on fixed date at
4    ;; location when depression angle of sun is alpha.
5    ;; Returns bogus if there is no dusk on date.
6    (let* ((result (moment-of-depression

```

```

7          (+ date (hr 18)) location alpha evening)))
8    (if (equal result bogus)
9        bogus
10       (standard-from-local result location)))

```

```

(14.71) 1  (defun refraction (tee location) (14.75)

```

```

2    ;; TYPE (moment location) -> half-circle
3    ;; Refraction angle at moment tee at location.
4    ;; The moment is not used.
5    (let* ((h (max (mt 0) (elevation location)))
6           (cap-R (mt 6.372d6)) ; Radius of Earth.
7           (dip ; Depression of visible horizon.
8               (arccos-degrees (/ cap-R (+ cap-R h))))
9           (+ (mins 34) dip
10             (* (secs 19) (sqrt h))))

```

```

(14.72) 1  (defun sunrise (date location) (14.76)
2    ;; TYPE (fixed-date location) -> moment
3    ;; Standard time of sunrise on fixed date at
4    ;; location.
5    (let* ((alpha (+ (refraction (+ date (hr 6)) location)
6                      (mins 16))))
7      (dawn date location alpha)))

```

```

(14.73) 1  (defun sunset (date location) (14.77)

```

```

2    ;; TYPE (fixed-date location) -> moment
3    ;; Standard time of sunset on fixed date at
4    ;; location.
5    (let* ((alpha (+ (refraction (+ date (hr 18)) location)
6                      (mins 16))))
7      (dusk date location alpha)))

```

```

(14.74) 1  (defun sunset (date location) (14.77)
2    ;; TYPE (fixed-date location) -> moment
3    ;; Standard time of sunset on fixed date at
4    ;; location.
5    (let* ((alpha (+ (refraction (+ date (hr 18)) location)
6                      (mins 16))))
7      (dusk date location alpha)))

```

```

1 (defun jewish-sabbath-ends (date location) (14.80) 13 (if waning
2 ;; TYPE (fixed-date location) -> moment 14 (if (> offset 0)
3 ;; Standard time of end of Jewish sabbath on fixed date 15 (- tee -1 offset)
4 ;; at location (as per Berthold Cohn). 16 (- tee offset))
5 (dusk date location (angle 7 5 0))) 17 (+ tee 1/2 offset)))
18 (rise (binary-search
19 1 (- approx (hr 6))
20 u (+ approx (hr 6))
21 x (> (observed-lunar-altitude x location)
22 (deg 0))
23 (< (- u 1) (mn 1))))))
24 (if (< rise (1+ tee))
25 (max (standard-from-universal rise location)
26 date) ; May be just before to midnight.
27 ;; Else no moonrise this day.
28 bogus)))

1 (defun observed-lunar-altitude (tee location) (14.82) 27
2 ;; TYPE (moment location) -> half-circle 28
3 ;; Observed altitude of upper limb of moon at tee at location,
4 ;; as a small positive/negative angle in degrees, including
5 ;; refraction and elevation.
6 (+ (topocentric-lunar-altitude tee location)
7 (refraction tee location)
8 (mins 16)))

1 (defun moonrise (date location) (14.83) 7
2 ;; TYPE (fixed-date location) -> moment
3 ;; Standard time of moonrise on fixed date at location.
4 ;; Returns bogus if there is no moonrise on date.
5 (let* ((tee ; Midnight.
6 (universal-from-standard date location))
7 (waning (> (lunar-phase tee) (deg 180)))
8 (alt ; Altitude at midnight.
9 (observed-lunar-altitude tee location))
10 (lat (latitude location))
11 (offset (/ alt (* 4 (- (deg 90) (abs lat)))))
12 (approx ; Approximate setting time.
13 (if waxing
14 (if (> offset 0)
15 (+ tee offset)
16 (+ tee 1 offset))
17 (- tee offset -1/2)))
18 (set (binary-search
19 (defun moonset (date location) (14.84)
20 ;; TYPE (fixed-date location) -> moment
21 ;; Standard time of moonset on fixed date at location.
22 ;; Returns bogus if there is no moonset on date.
23 (let* ((tee ; Midnight.
24 (universal-from-standard date location))
25 (waning (< (lunar-phase tee) (deg 180)))
26 (alt ; Altitude at midnight.
27 (observed-lunar-altitude tee location))
28 (lat (latitude location))
29 (offset (/ alt (* 4 (- (deg 90) (abs lat)))))
30 (approx ; Approximate setting time.
31 (if waxing
32 (if (> offset 0)
33 (+ tee offset)
34 (+ tee 1 offset))
35 (- tee offset -1/2)))
36 (set (binary-search

```

```

19      l (- approx (hr 6))
20      u (+ approx (hr 6))
21      x (< (observed-lunar-altitude x location) (deg 0))
22      (< (- u l) (mn 1))))
23  (if (< set (1+ tee))
24      (max (standard-from-universal set location)
25          date) ; May be just before to midnight.
26      ;; Else no moonset this day.
27      bogus))

```

```

1  (defconstant padua                                (14.85)
2      ;; TYPE location
3      ;; Location of Padua, Italy.
4      (location (angle 45 24 28) (angle 11 53 9) (mt 18) (hr 1)))

```

```

1  (defun local-zero-hour (tee)                       (14.86)
2      ;; TYPE moment -> moment
3      ;; Local time of dusk in Padua, Italy on date of moment tee.
4      (let* ((date (fixed-from-moment tee))
5             (local-from-standard
6              (+ (dusk date padua (angle 0 16 0)) ; Sunset.
7                 (mn 30)) ; Dusk.
8              padua)))

```

```

1  (defun local-from-italian (tee)                   (14.87)
2      ;; TYPE moment -> moment
3      ;; Local time corresponding to Italian time tee.
4      (let* ((date (fixed-from-moment tee))
5             (z (local-zero-hour (1- tee))))
6          (- tee (- date z))))

```

```

1  (defun italian-from-local (tee_ell)                (14.88)
2      ;; TYPE moment -> moment
3      ;; Italian time corresponding to local time tee_ell.
4      (let* ((date (fixed-from-moment tee_ell))
5             (z0 (local-zero-hour (1- tee_ell)))
6             (z (local-zero-hour tee_ell)))
7          (if (> tee_ell z) ; if after zero hour
8              (+ tee_ell (- date -1 z)) ; then next day
9              (+ tee_ell (- date z0)))))

```

```

1  (defun daytime-temporal-hour (date location)       (14.89)
2      ;; TYPE (fixed-date location) -> real
3      ;; Length of daytime temporal hour on fixed date at location.
4      ;; Returns bogus if there no sunrise or sunset on date.
5      (if (or (equal (sunrise date location) bogus)
6              (equal (sunset date location) bogus))
7          bogus
8          (/ (- (sunset date location)
9                (sunrise date location))
10             12)))

```

```

1  (defun nighttime-temporal-hour (date location)     (14.90)
2      ;; TYPE (fixed-date location) -> real
3      ;; Length of nighttime temporal hour on fixed date at location.
4      ;; Returns bogus if there no sunrise or sunset on date.
5      (if (or (equal (sunrise (1+ date) location) bogus)
6              (equal (sunset date location) bogus))
7          bogus
8          (/ (- (sunrise (1+ date) location)
9                (sunset date location))
10             12)))

```

```

1  (defun standard-from-sundial (tee location)        (14.91)
2      ;; TYPE (moment location) -> moment
3      ;; Standard time of temporal moment tee at location.
4      ;; Returns bogus if temporal hour is undefined that day.

```

```

5      (let* ((date (fixed-from-moment tee))
6             (hour (* 24 (time-from-moment tee)))
7             (h (cond ((<= 6 hour 18); daytime today
8                      (daytime-temporal-hour date location))
9                      ((< hour 6) ; early this morning
10                     (nighttime-temporal-hour (1- date) location))
11                     (t ; this evening
12                      (nighttime-temporal-hour date location))))))
13      (cond ((equal h bogus) bogus)
14            ((<= 6 hour 18); daytime today
15             (+ (sunrise date location) (* (- hour 6) h)))
16            ((< hour 6) ; early this morning
17             (+ (sunset (1- date) location) (* (+ hour 6) h)))
18            (t ; this evening
19             (+ (sunset date location) (* (- hour 18) h))))))

```

```

1      (defun jewish-morning-end (date location)                (14.92)
2        ;; TYPE (fixed-date location) -> moment
3        ;; Standard time on fixed date at location of end of
4        ;; morning according to Jewish ritual.
5        (standard-from-sundial (+ date (hr 10)) location))

```

```

1      (defun asr (date location)                                (14.93)
2        ;; TYPE (fixed-date location) -> moment
3        ;; Standard time of asr on fixed date at location.
4        ;; According to Hanafi rule.
5        ;; Returns bogus is no asr occurs.
6        (let* ((noon ; Time when sun nearest zenith.
7               (midday date location))
8               (phi (latitude location))
9               (delta ; Solar declination at noon.
10              (declination noon (deg 0) (solar-longitude noon)))
11              (altitude ; Solar altitude at noon.
12              (arcsin-degrees

```

```

13              (+ (* (cos-degrees delta) (cos-degrees phi))
14                 (* (sin-degrees delta) (sin-degrees phi))))))
15        (h ; Sun's altitude when shadow increases by
16         (mod3 (arctan-degrees ; ... double its length.
17                (tan-degrees altitude)
18                (1+ (* 2 (tan-degrees altitude))))
19         -90 90)))
20        (if (<= altitude (deg 0)) ; No shadow.
21            bogus
22            (dusk date location (- h))))))

```

```

1      (defun alt-asr (date location)                            (14.94)
2        ;; TYPE (fixed-date location) -> moment
3        ;; Standard time of asr on fixed date at location.
4        ;; According to Shafi'i rule.
5        ;; Returns bogus is no asr occurs.
6        (let* ((noon ; Time when sun nearest zenith.
7               (midday date location))
8               (phi (latitude location))
9               (delta ; Solar declination at noon.
10              (declination noon (deg 0) (solar-longitude noon)))
11              (altitude ; Solar altitude at noon.
12              (arcsin-degrees
13              (+ (* (cos-degrees delta) (cos-degrees phi))
14                 (* (sin-degrees delta) (sin-degrees phi))))))
15        (h ; Sun's altitude when shadow increases by
16         (mod3 (arctan-degrees ; ... its length.
17                (tan-degrees altitude)
18                (1+ (tan-degrees altitude))))
19         -90 90)))
20        (if (<= altitude (deg 0)) ; No shadow.
21            bogus
22            (dusk date location (- h))))))

```

The functions for lunar visibility are:

```

1  (defun arc-of-light (tee)
2    ;; TYPE moment -> half-circle
3    ;; Angular separation of sun and moon
4    ;; at moment tee.
5    (arccos-degrees
6      (* (cos-degrees (lunar-latitude tee))
7         (cos-degrees (lunar-phase tee)))))
14.95)

1  (defun simple-best-view (date location)
2    ;; TYPE (fixed-date location) -> moment
3    ;; Best viewing time (UT) in the evening.
4    ;; Simple version.
5    (let* ((dark ; Best viewing time prior evening.
6           (dusk date location (deg 4.5L0)))
7          (best (if (equal dark bogus)
8                    (1+ date) ; An arbitrary time.
9                    dark)))
10      (universal-from-standard best location)))

1  (defun shaukat-criterion (date location)
2    ;; TYPE (fixed-date location) -> boolean
3    ;; S. K. Shaukat's criterion for likely
4    ;; visibility of crescent moon on eve of date at location.
5    ;; Not intended for high altitudes or polar regions.
6    (let* ((tee (simple-best-view (1- date) location))
7          (phase (lunar-phase tee))
8          (h (lunar-altitude tee location))
9          (cap-ARCL (arc-of-light tee)))
10      (and (< new phase first-quarter)
11           (<= (deg 10.6L0) cap-ARCL (deg 90))
12           (> h (deg 4.1L0)))))

1  (defun arc-of-vision (tee location)
2    ;; TYPE (moment location) -> half-circle
3    ;; Angular difference in altitudes of sun and moon
4    ;; at moment tee at location.
5    (- (lunar-altitude tee location)
6       (solar-altitude tee location)))
14.98)

1  (defun bruin-best-view (date location)
2    ;; TYPE (fixed-date location) -> moment
3    ;; Best viewing time (UT) in the evening.
4    ;; Yallop version, per Bruin (1977).
5    (let* ((sun (sunset date location))
6          (moon (moonset date location))
7          (best ; Best viewing time prior evening.
8               (if (or (equal sun bogus) (equal moon bogus))
                   (1+ date) ; An arbitrary time.
                   (+ (* 5/9 sun) (* 4/9 moon)))))
10      (universal-from-standard best location)))
14.96)

1  (defun yallop-criterion (date location)
2    ;; TYPE (fixed-date location) -> boolean
3    ;; B. D. Yallop's criterion for possible
4    ;; visibility of crescent moon on eve of date at location.
5    ;; Not intended for high altitudes or polar regions.
6    (let* ((tee ; Best viewing time prior evening.
7           (bruin-best-view (1- date) location))
8          (phase (lunar-phase tee))
9          (cap-D (lunar-semi-diameter tee location))
10         (cap-ARCL (arc-of-light tee))
11         (cap-W (* cap-D (- 1 (cos-degrees cap-ARCL))))
12         (cap-ARCV (arc-of-vision tee location))
13         (e -0.14L0) ; Crescent visible under perfect conditions.
14         (q1 (poly cap-W
15                  (list 11.8371L0 -6.3226L0 0.7319L0 -0.1018L0))))
16      (and (< new phase first-quarter)
17           (> cap-ARCV (+ q1 e)))))
14.100)

```

```

1 (defun lunar-semi-diameter (tee location) (14.101) 12 (- moon 30) ; Must go back a month.
2 ;; TYPE (moment location) -> half-circle 13 moon)))
3 ;; Topocentric lunar semi-diameter at moment tee and location. 14 (next d tau (visible-crescent d location)))
4 (let* ((h (lunar-altitude tee location))
5 (p (lunar-parallax tee location)))
6 (* 0.27245L0 p (1+ (* (sin-degrees h) (sin-degrees p))))))

1 (defun lunar-diameter (tee) (14.102) 5 (defun phasis-on-or-after (date location) (14.105)
2 ;; TYPE moment -> angle 6 (fixed-from-moment
3 ;; Geocentric apparent lunar diameter of the moon (in 7 (lunar-phase-at-or-before new date)))
4 ;; degrees) at moment tee. Adapted from "Astronomical 8 (age (- date moon))
5 ;; Algorithms" by Jean Meeus, Willmann-Bell, 2nd edn., 9 (tau ; Check if not visible yet on eve of date.
6 ;; 1998. 10 (if (or (<= 4 age)
7 (/ (deg 1792367000/9) (lunar-distance tee))) 11 (visible-crescent (1- date) location))
12 (+ moon 29) ; Next new moon
13 date)))
1 (defun visible-crescent (date location) (14.103) 14 (next d tau (visible-crescent d location)))
2 ;; TYPE (fixed-date location) -> boolean
3 ;; Criterion for possible visibility of crescent moon
4 ;; on eve of date at location.
5 ;; Shaukat's criterion may be replaced with another.
6 (shaukat-criterion date location))

1 (defun phasis-on-or-before (date location) (14.104)
2 ;; TYPE (fixed-date location) -> fixed-date
3 ;; Closest fixed date on or before date when crescent
4 ;; moon first became visible at location.
5 (let* ((moon ; Prior new moon.
6 (fixed-from-moment
7 (lunar-phase-at-or-before new date)))
8 (age (- date moon))
9 (tau ; Check if not visible yet on eve of date.
10 (if (and (<= age 3)
11 (not (visible-crescent date location)))

1 (defun persian-date (year month day)
2 ;; TYPE (persian-year persian-month persian-day)
3 ;; TYPE -> persian-date
4 (list year month day))

1 (defconstant persian-epoch (15.1)
2 ;; TYPE fixed-date
3 ;; Fixed date of start of the Persian calendar.
4 (fixed-from-julian (julian-date (ce 622) march 19)))

1 (defconstant tehran (15.2)
2 ;; TYPE location
3 ;; Location of Tehran, Iran.
4 (location (deg 35.68L0) (deg 51.42L0)
5 (mt 1100) (hr (+ 3 1/2))))

```

D.15 The Persian Calendar

```

1  (defun midday-in-tehran (date)
2    ;; TYPE fixed-date -> moment
3    ;; Universal time of true noon on fixed date in Tehran.
4    (midday date tehran))

1  (defun persian-new-year-on-or-before (date)
2    ;; TYPE fixed-date -> fixed-date
3    ;; Fixed date of Astronomical Persian New Year on or
4    ;; before fixed date.
5    (let* ((approx ; Approximate time of equinox.
6            (estimate-prior-solar-longitude
7              spring (midday-in-tehran date))))
8      (next day (- (floor approx) 1)
9        (<= (solar-longitude (midday-in-tehran day))
10          (+ spring (deg 2))))))

1  (defun fixed-from-persian (p-date)
2    ;; TYPE persian-date -> fixed-date
3    ;; Fixed date of Astronomical Persian date p-date.
4    (let* ((month (standard-month p-date))
5           (day (standard-day p-date))
6           (year (standard-year p-date))
7           (new-year
8             (persian-new-year-on-or-before
9               (+ persian-epoch 180; Fall after epoch.
10                 (floor
11                   (* mean-tropical-year
12                     (if (< 0 year)
13                       (1- year)
14                       year)))))))); No year zero.
15      (+ (1- new-year) ; Days in prior years.
16        (if (<= month 7) ; Days in prior months this year.
17          (* 31 (1- month))
18          (+ (* 30 (1- month)) 6))
19        day))) ; Days so far this month.

```

```

(15.3) 1  (defun persian-from-fixed (date)
2    ;; TYPE fixed-date -> persian-date
3    ;; Astronomical Persian date (year month day)
4    ;; corresponding to fixed date.
5    (let* ((new-year
6            (persian-new-year-on-or-before date))
7           (y (1+ (round (/ (- new-year persian-epoch)
8                             mean-tropical-year))))
9           (year (if (< 0 y)
10                    y
11                    (1- y)))); No year zero
12      (day-of-year (1+ (- date
13                          (fixed-from-persian
14                            (persian-date year 1 1)))))
15      (month (if (<= day-of-year 186)
16                 (ceiling (/ day-of-year 31))
17                 (ceiling (/ (- day-of-year 6) 30))))
18      (day ; Calculate the day by subtraction
19        (- date (1- (fixed-from-persian
20                     (persian-date year month 1)))))
21      (persian-date year month day)))

(15.5) 1  (defun arithmetic-persian-leap-year? (p-year)
2    ;; TYPE persian-year -> boolean
3    ;; True if p-year is a leap year on the Persian calendar.
4    (let* ((y ; Years since start of 2820-year cycles
5            (if (< 0 p-year)
6                (- p-year 474)
7                (- p-year 473)))); No year zero
8      (year ; Equivalent year in the range 474..3263
9        (+ (mod y 2820) 474)))
10      (< (mod (* (+ year 38)
11                 31)
12          128)
13        31)))

```

(15.7)

```

1 (defun fixed-from-arithmetic-persian (p-date) (15.8) 10 (d1 ; Prior days not in n2820--that is, days
2 ;; TYPE persian-date -> fixed-date 11 ; since start of last 2820-year cycle
3 ;; Fixed date equivalent to Persian date p-date. 12 (mod d0 1029983))
4 (let* ((day (standard-day p-date)) 13 (y2820 ; Years since start of last 2820-year cycle
5 (month (standard-month p-date)) 14 (if (= d1 1029982)
6 (p-year (standard-year p-date)) 15 ; Last day of 2820-year cycle
7 (y ; Years since start of 2820-year cycle 16 2820
8 (if (< 0 p-year) 17 ; Otherwise use cycle of years formula
9 (- p-year 474) 18 (quotient (+ (* 128 d1) 46878)
10 (- p-year 473))) ; No year zero 19 46751)))
11 (year ; Equivalent year in the range 474..3263 20 (year ; Years since Persian epoch
12 (+ (mod y 2820) 474))) 21 (+ 474 ; Years before start of 2820-year cycles
13 (+ (1- persian-epoch); Days before epoch 22 (* 2820 n2820) ; Years in prior 2820-year cycles
14 (* 1029983 ; Days in 2820-year cycles 23 y2820)); Years since start of last 2820-year
15 ; before Persian year 474 24 ; cycle
16 (quotient y 2820)) 25 (if (< 0 year)
17 (* 365 (1- year)) ; Nonleap days in prior years this 26 year
18 ; 2820-year cycle 27 (1- year)))) ; No year zero
19 (quotient ; Leap days in prior years this
20 ; 2820-year cycle
21 (- (* 31 year) 5) 128)
22 (if (<= month 7) ; Days in prior months this year
23 (* 31 (1- month))
24 (+ (* 30 (1- month)) 6))
25 day))) ; Days so far this month

1 (defun arithmetic-persian-year-from-fixed (date) (15.9)
2 ;; TYPE fixed-date -> persian-year
3 ;; Persian year corresponding to the fixed date.
4 (let* ((d0 ; Prior days since start of 2820-year cycle
5 ; beginning in Persian year 474
6 (- date (fixed-from-arithmetic-persian
7 (persian-date 475 1 1))))
8 (n2820 ; Completed prior 2820-year cycles
9 (quotient d0 1029983))

1 (defun arithmetic-persian-from-fixed (date) (15.10)
2 ;; TYPE fixed-date -> persian-date
3 ;; Persian date corresponding to fixed date.
4 (let* ((year (arithmetic-persian-year-from-fixed date))
5 (day-of-year (1+ (- date
6 (fixed-from-arithmetic-persian
7 (persian-date year 1 1)))))
8 (month (if (<= day-of-year 186)
9 (ceiling (/ day-of-year 31))
10 (ceiling (/ (- day-of-year 6) 30))))
11 (day ; Calculate the day by subtraction
12 (- date (1- (fixed-from-arithmetic-persian
13 (persian-date year month 1)))))
14 (persian-date year month day)))

1 (defun nowruz (g-year) (15.11)
2 ;; TYPE gregorian-year -> fixed-date

```

```

3      ;; Fixed date of Persian New Year (Nowruz) in Gregorian
4      ;; year g-year.
5      (let* ((persian-year
6              (1+ (- g-year
7                    (gregorian-year-from-fixed
8                      persian-epoch))))
9              (y (if (<= persian-year 0)
10                   ;; No Persian year 0
11                   (1- persian-year)
12                   persian-year)))
13      (fixed-from-persian (persian-date y 1 1)))

```

D.16 The Bahá'í Calendar

```

1 (defun bahai-date (major cycle year month day)
2   ;; TYPE (bahai-major bahai-cycle bahai-year
3   ;; TYPE bahai-month bahai-day) -> bahai-date
4   (list major cycle year month day))

```

```

1 (defun bahai-major (date)
2   ;; TYPE bahai-date -> bahai-major
3   (first date))

```

```

1 (defun bahai-cycle (date)
2   ;; TYPE bahai-date -> bahai-cycle
3   (second date))

```

```

1 (defun bahai-year (date)
2   ;; TYPE bahai-date -> bahai-year
3   (third date))

```

```

1 (defun bahai-month (date)
2   ;; TYPE bahai-date -> bahai-month
3   (fourth date))

```

```

1 (defun bahai-day (date)
2   ;; TYPE bahai-date -> bahai-day
3   (fifth date))

```

```

1 (defconstant ayyam-i-ha                                     (16.1)
2   ;; TYPE bahai-month
3   ;; Signifies intercalary period of 4 or 5 days.
4   0)

```

```

1 (defconstant bahai-epoch                                     (16.2)
2   ;; TYPE fixed-date
3   ;; Fixed date of start of Baha'i calendar.
4   (fixed-from-gregorian (gregorian-date 1844 march 21)))

```

```

1 (defun fixed-from-bahai (b-date)                             (16.3)
2   ;; TYPE bahai-date -> fixed-date
3   ;; Fixed date equivalent to the Baha'i date b-date.
4   (let* ((major (bahai-major b-date))
5           (cycle (bahai-cycle b-date))
6           (year (bahai-year b-date))
7           (month (bahai-month b-date))
8           (day (bahai-day b-date))
9           (g-year; Corresponding Gregorian year.
10            (+ (* 361 (1- major))
11               (* 19 (1- cycle)) year -1
12               (gregorian-year-from-fixed bahai-epoch))))
13     (+ (fixed-from-gregorian ; Prior years.
14        (gregorian-date g-year march 20))

```

```

15      (cond ((= month ayyam-i-ha) ; Intercalary period.
16            342) ; 18 months have elapsed.
17            ((= month 19); Last month of year.
18            (if (gregorian-leap-year? (1+ g-year))
19              347 ; Long ayyam-i-ha.
20              346)); Ordinary ayyam-i-ha.
21            (t (* 19 (1- month))))); Elapsed months.
22            day))) ; Days of current month.

```

```

1  (defun bahai-from-fixed (date)                                (16.4)
2    ;; TYPE fixed-date -> bahai-date
3    ;; Baha'i (major cycle year month day) corresponding to fixed
4    ;; date.
5    (let* ((g-year (gregorian-year-from-fixed date))
6           (start ; 1844
7             (gregorian-year-from-fixed bahai-epoch))
8           (years ; Since start of Baha'i calendar.
9             (- g-year start
10              (if (<= date
11                  (fixed-from-gregorian
12                    (gregorian-date g-year march 20)))
13                  1 0)))
14           (major (1+ (quotient years 361)))
15           (cycle (1+ (quotient (mod years 361) 19)))
16           (year (1+ (mod years 19)))
17           (days; Since start of year
18            (- date (fixed-from-bahai
19                    (bahai-date major cycle year 1 1))))
20           (month
21            (cond ((>= date
22                   (fixed-from-bahai
23                     (bahai-date major cycle year 19 1)))
24                  19) ; Last month of year.
25                  (>= date ; Intercalary days.
26                    (fixed-from-bahai

```

```

27            (bahai-date major cycle year
28              ayyam-i-ha 1)))
29            ayyam-i-ha) ; Intercalary period.
30            (t (1+ (quotient days 19))))))
31            (day (- date -1
32                  (fixed-from-bahai
33                    (bahai-date major cycle year month 1))))))
34            (bahai-date major cycle year month day)))

```

```

1  (defconstant bahai-location                                (16.5)
2    ;; TYPE location
3    ;; Location of Tehran for astronomical Baha'i calendar.
4    (location (deg 35.696111L0) (deg 51.423056L0)
5              (mt 0) (hr (+ 3 1/2))))

```

```

1  (defun bahai-sunset (date)                                (16.6)
2    ;; TYPE fixed-date -> moment
3    ;; Universal time of sunset on fixed date
4    ;; in Bahai-Location.
5    (universal-from-standard
6      (sunset date bahai-location)
7      bahai-location))

```

```

1  (defun astro-bahai-new-year-on-or-before (date)           (16.7)
2    ;; TYPE fixed-date -> fixed-date
3    ;; Fixed date of astronomical Bahai New Year on or before fixed
4    ;; date.
5    (let* ((approx ; Approximate time of equinox.
6            (estimate-prior-solar-longitude
7              spring (bahai-sunset date))))
8            (next day (1- (floor approx)))
9            (<= (solar-longitude (bahai-sunset day))
10              (+ spring (deg 2)))))

```

```

1  (defun fixed-from-astro-bahai (b-date)
2    ;; TYPE bahai-date -> fixed-date
3    ;; Fixed date of Baha'i date b-date.
4    (let* ((major (bahai-major b-date))
5           (cycle (bahai-cycle b-date))
6           (year (bahai-year b-date))
7           (month (bahai-month b-date))
8           (day (bahai-day b-date))
9           (years; Years from epoch
10            (+ (* 361 (1- major))
11               (* 19 (1- cycle))
12               year)))
13      (cond ((= month 19); last month of year
14             (+ (astro-bahai-new-year-on-or-before
15                 (+ bahai-epoch
16                   (floor (* mean-tropical-year
17                           (+ years 1/2))))))
18                 -20 day))
19             ((= month ayyam-i-ha)
20              ;; intercalary month, between 18th & 19th
21              (+ (astro-bahai-new-year-on-or-before
22                  (+ bahai-epoch
23                    (floor (* mean-tropical-year
24                            (- years 1/2))))))
25                  341 day))
26             (t (+ (astro-bahai-new-year-on-or-before
27                   (+ bahai-epoch
28                     (floor (* mean-tropical-year
29                             (- years 1/2))))))
30                   (* (1- month) 19)
31                   day -1))))

```

```

1  (defun astro-bahai-from-fixed (date)
2    ;; TYPE fixed-date -> bahai-date
3    ;; Astronomical Baha'i date corresponding to fixed date.

```

```

(16.8) 4  (let* ((new-year (astro-bahai-new-year-on-or-before date))
5         (years (round (/ (- new-year bahai-epoch)
6                             mean-tropical-year)))
7         (major (1+ (quotient years 361)))
8         (cycle (1+ (quotient (mod years 361) 19)))
9         (year (1+ (mod years 19)))
10        (days; Since start of year
11              (- date new-year))
12        (month
13         (cond
14          ((>= date (fixed-from-astro-bahai
15                     (bahai-date major cycle year 19 1)))
16           ; last month of year
17            19)
18          ((>= date
19            (fixed-from-astro-bahai
20              (bahai-date major cycle year ayyam-i-ha 1)))
21           ; intercalary month
22            ayyam-i-ha)
23          (t (1+ (quotient days 19))))))
24        (day (- date -1
25                (fixed-from-astro-bahai
26                  (bahai-date major cycle year month 1))))))
27        (bahai-date major cycle year month day)))

```

```

1  (defun bahai-new-year (g-year) (16.10)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Baha'i New Year in Gregorian year g-year.
4    (fixed-from-gregorian
5     (gregorian-date g-year march 21)))

```

```

(16.9) 1  (defun naw-ruz (g-year) (16.11)
2    ;; TYPE gregorian-year -> fixed-date
3    ;; Fixed date of Baha'i New Year (Naw-Ruz) in Gregorian

```

```
4 ;; year g-year.
5 (astro-bahai-new-year-on-or-before
6   (gregorian-new-year (1+ g-year)))
```

```
1 (defun feast-of-ridvan (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of Feast of Ridvan in Gregorian year g-year.
4   (+ (naw-ruz g-year) 31))
```

```
1 (defun birth-of-the-bab (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of the Birthday of the Bab
4   ;; in Gregorian year g-year.
5   (let* ((ny ; Beginning of Baha'i year.
6           (naw-ruz g-year))
7          (set1 (bahai-sunset ny))
8          (m1 (new-moon-at-or-after set1))
9          (m8 (new-moon-at-or-after (+ m1 190)))
10         (day (fixed-from-moment m8))
11         (set8 (bahai-sunset day)))
12     (if (< m8 set8)
13         (1+ day)
14         (+ day 2))))
```

D.17 The French Revolutionary Calendar

```
1 (defun french-date (year month day)
2   ;; TYPE (french-year french-month french-day) -> french-date
3   (list year month day))
```

```
1 (defconstant paris
2   ;; TYPE location
```

```
3   ;; Location of Paris Observatory. Longitude corresponds
4   ;; to difference of 9m 21s between Paris time zone and
5   ;; Universal Time.
6   (location (angle 48 50 11) (angle 2 20 15) (mt 27) (hr 1)))
```

```
1 (defun midnight-in-paris (date)
2   ;; TYPE fixed-date -> moment
3   ;; Universal time of true midnight at end of fixed date
4   ;; in Paris.
5   (midnight (+ date 1) paris))
```

```
1 (defun french-new-year-on-or-before (date)
2   ;; TYPE fixed-date -> fixed-date
3   ;; Fixed date of French Revolutionary New Year on or
4   ;; before fixed date.
5   (let* ((approx ; Approximate time of solstice.
6             (estimate-prior-solar-longitude
7              autumn (midnight-in-paris date))))
8       (next day (- (floor approx) 1)
9                 (<= autumn (solar-longitude
10                          (midnight-in-paris day))))))
```

```
1 (defconstant french-epoch
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the French Revolutionary
4   ;; calendar.
5   (fixed-from-gregorian (gregorian-date 1792 september 22)))
```

```
1 (defun fixed-from-french (f-date)
2   ;; TYPE french-date -> fixed-date
3   ;; Fixed date of French Revolutionary date.
4   (let* ((month (standard-month f-date))
5          (day (standard-day f-date))
6          (year (standard-year f-date))
7          (new-year
```

```

8      (french-new-year-on-or-before
9      (floor (+ french-epoch 180; Spring after epoch.
10             (* mean-tropical-year
11               (1- year))))))
12      (+ new-year -1      ; Days in prior years
13        (* 30 (1- month)); Days in prior months
14        day)))           ; Days this month

```

```

1  (defun french-from-fixed (date)
2    ;; TYPE fixed-date -> french-date
3    ;; French Revolutionary date of fixed date.
4    (let* ((new-year
5            (french-new-year-on-or-before date))
6           (year (1+ (round (/ (- new-year french-epoch)
7                               mean-tropical-year))))
7           (month (1+ (quotient (- date new-year) 30)))
8           (day (1+ (mod (- date new-year) 30))))
9           (french-date year month day)))
10

```

(17.6)

```

1  (defun french-leap-year? (f-year)
2    ;; TYPE french-year -> boolean
3    ;; True if f-year is a leap year on the
4    ;; French Revolutionary calendar.
5    (> (- (fixed-from-french
6          (french-date (1+ f-year) 1 1))
7          (fixed-from-french
8            (french-date f-year 1 1)))
9       365))

```

(17.7)

```

1  (defun arithmetic-french-leap-year? (f-year)
2    ;; TYPE french-year -> boolean
3    ;; True if f-year is a leap year on the
4    ;; Arithmetic French Revolutionary calendar.

```

(17.8)

```

5      (and (= (mod f-year 4) 0)
6            (not (member (mod f-year 400) (list 100 200 300)))
7            (not (= (mod f-year 4000) 0))))

```

```

1  (defun fixed-from-arithmetic-french (f-date)
2    ;; TYPE french-date -> fixed-date
3    ;; Fixed date of Arithmetic French Revolutionary
4    ;; date f-date.
5    (let* ((month (standard-month f-date))
6           (day (standard-day f-date))
7           (year (standard-year f-date)))
8      (+ french-epoch -1; Days before start of calendar.
9        (* 365 (1- year)); Ordinary days in prior years.
10        ; Leap days in prior years.
11        (quotient (1- year) 4)
12        (- (quotient (1- year) 100))
13        (quotient (1- year) 400)
14        (- (quotient (1- year) 4000))
15        (* 30 (1- month)); Days in prior months this year.
16        day)); Days this month.

```

(17.9)

```

1  (defun arithmetic-french-from-fixed (date)
2    ;; TYPE fixed-date -> french-date
3    ;; Arithmetic French Revolutionary date (year month day)
4    ;; of fixed date.
5    (let* ((approx ; Approximate year (may be off by 1).
6            (1+ (quotient (- date french-epoch -2)
7                            1460969/4000)))
7           (year (if (< date
8                      (fixed-from-arithmetic-french
9                        (french-date approx 1 1)))
10                   (1- approx)
11                   approx))
12           (month ; Calculate the month by division.
13                 (1+ (quotient
14                      (- date (fixed-from-arithmetic-french

```

(17.10)

```

16         (french-date year 1 1)))
17         30)))
18         (day ; Calculate the day by subtraction.
19         (1+ (- date
20             (fixed-from-arithmetic-french
21              (french-date year month 1))))))
22         (french-date year month day)))

```

D.18 Astronomical Lunar Calendars

```

1 (defun babylonian-date (year month leap day)
2   ;; TYPE (babylonian-year babylonian-month
3   ;; TYPE babylonian-leap babylonian-day)
4   ;; TYPE -> babylonian-date
5   (list year month leap day))

```

```

1 (defun babylonian-year (date)
2   ;; TYPE babylonian-date -> babylonian-year
3   (first date))

```

```

1 (defun babylonian-month (date)
2   ;; TYPE babylonian-date -> babylonian-month
3   (second date))

```

```

1 (defun babylonian-leap (date)
2   ;; TYPE babylonian-date -> babylonian-leap
3   (third date))

```

```

1 (defun babylonian-day (date)
2   ;; TYPE babylonian-date -> babylonian-day
3   (fourth date))

```

```

1 (defun moonlag (date location) (18.1)
2   ;; TYPE (fixed-date location) -> duration
3   ;; Time between sunset and moonset on date at location.
4   ;; Returns bogus if there is no sunset on date.
5   (let* ((sun (sunset date location))
6          (moon (moonset date location)))
7     (cond ((equal sun bogus) bogus)
8           ((equal moon bogus) (hr 24)) ; Arbitrary.
9           (t (- moon sun)))))

```

```

1 (defconstant babylon (18.2)
2   ;; TYPE location
3   ;; Location of Babylon.
4   (location (deg 32.4794L0) (deg 44.4328L0)
5             (mt 26) (hr (+ 3 1/2))))

```

```

1 (defun babylonian-criterion (date) (18.3)
2   ;; TYPE (fixed-date location) -> boolean
3   ;; Moonlag criterion for visibility of crescent moon on
4   ;; eve of date in Babylon.
5   (let* ((set (sunset (1- date) babylon))
6          (tee (universal-from-standard set babylon))
7          (phase (lunar-phase tee)))
8     (and (< new phase first-quarter)
9          (<= (new-moon-before tee) (- tee (hr 24)))
10          (> (moonlag (1- date) babylon) (mn 48)))))

```

```

1 (defun babylonian-new-month-on-or-before (date) (18.4)
2   ;; TYPE fixed-date -> fixed-date
3   ;; Fixed date of start of Babylonian month on or before
4   ;; Babylonian date. Using lag of moonset criterion.
5   (let* ((moon ; Prior new moon.
6          (fixed-from-moment
7           (lunar-phase-at-or-before new date))))

```

```

8      (age (- date moon))
9      (tau ; Check if not visible yet on eve of date.
10     (if (and (<= age 3)
11           (not (babylonian-criterion date)))
12         (- moon 30) ; Must go back a month.
13         moon)))
14     (next d tau (babylonian-criterion d))))

```

```

1 (defconstant babylonian-epoch (18.5)
2   ;; TYPE fixed-date
3   ;; Fixed date of start of the Babylonian calendar
4   ;; (Seleucid era). April 3, 311 BCE (Julian).
5   (fixed-from-julian (julian-date (bce 311) april 3)))

```

```

1 (defun babylonian-leap-year? (b-year) (18.6)
2   ;; TYPE babylonian-year -> boolean
3   ;; True if b-year is a leap year on Babylonian calendar.
4   (< (mod (+ (* 7 b-year) 13) 19) 7))

```

```

1 (defun fixed-from-babylonian (b-date) (18.7)
2   ;; TYPE babylonian-date -> fixed-date
3   ;; Fixed date equivalent to Babylonian date.
4   (let* ((month (babylonian-month b-date))
5          (leap (babylonian-leap b-date))
6          (day (babylonian-day b-date))
7          (year (babylonian-year b-date))
8          (month1 ; Elapsed months this year.
9                (if (or leap
10                      (and (= (mod year 19) 18)
11                          (> month 6)))
12                    month (1- month)))
13          (months ; Elapsed months since epoch.
14                (+ (quotient (+ (* (1- year) 235) 13) 19)

```

```

15                month1))
16                (midmonth ; Middle of given month.
17                (+ babylonian-epoch
18                  (round (* mean-synodic-month months)) 15)))
19                (+ (babylonian-new-month-on-or-before midmonth)
20                  day -1)))

```

```

1 (defun babylonian-from-fixed (date) (18.8)
2   ;; TYPE fixed-date -> babylonian-date
3   ;; Babylonian date corresponding to fixed date.
4   (let* ((crescent ; Most recent new month.
5          (babylonian-new-month-on-or-before date))
6          (months ; Elapsed months since epoch.
7                (round (/ (- crescent babylonian-epoch)
8                          mean-synodic-month)))
9          (year (1+ (quotient (+ (* 19 months) 5) 235)))
10          (approx ; Approximate date of new year.
11                (+ babylonian-epoch
12                  (round (* (quotient (+ (* (1- year) 235) 13) 19)
13                          mean-synodic-month))))
14          (new-year (babylonian-new-month-on-or-before
15                    (+ approx 15)))
16          (month1 (1+ (round (/ (- crescent new-year) 29.5L0))))
17          (special (= (mod year 19) 18))
18          (leap (if special (= month1 7) (= month1 13)))
19          (month (if (or leap (and special (> month1 6)))
20                    (1- month1)
21                    month1))
22          (day (- date crescent -1)))
23          (babylonian-date year month leap day)))

```

```

1 (defun astronomical-easter (g-year) (18.9)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Date of (proposed) astronomical Easter in Gregorian
4   ;; year g-year.
5   (let* ((equinox ; Spring equinox.

```

```

6      (season-in-gregorian spring g-year))
7      (paschal-moon ; Date of next full moon.
8      (floor (apparent-from-universal
9      (lunar-phase-at-or-after full equinox)
10     jerusalem))))
11     ;; Return the Sunday following the Paschal moon.
12     (kday-after sunday paschal-moon)))

```

```

1  (defconstant islamic-location          (18.10)
2    ;; TYPE location
3    ;; Sample location for Observational Islamic calendar
4    ;; (Cairo, Egypt).
5    (location (deg 30.1L0) (deg 31.3L0) (mt 200) (hr 2)))

```

```

1  (defun fixed-from-observational-islamic (i-date)          (18.11)
2    ;; TYPE islamic-date -> fixed-date
3    ;; Fixed date equivalent to Observational Islamic date
4    ;; i-date.
5    (let* ((month (standard-month i-date))
6           (day (standard-day i-date))
7           (year (standard-year i-date))
8           (midmonth ; Middle of given month.
9           (+ islamic-epoch
10             (floor (* (+ (* (1- year) 12)
11                           month -1/2)
12                       mean-synodic-month))))))
13      (+ (phasis-on-or-before ; First day of month.
14          midmonth islamic-location)
15         day -1)))

```

```

1  (defun observational-islamic-from-fixed (date)          (18.12)
2    ;; TYPE fixed-date -> islamic-date
3    ;; Observational Islamic date (year month day)

```

```

4    ;; corresponding to fixed date.
5    (let* ((crescent ; Most recent new moon.
6           (phasis-on-or-before date islamic-location))
7           (elapsed-months
8           (round (/ (- crescent islamic-epoch)
9                       mean-synodic-month)))
10           (year (1+ (quotient elapsed-months 12)))
11           (month (1+ (mod elapsed-months 12)))
12           (day (1+ (- date crescent))))
13           (islamic-date year month day)))

```

```

1  (defun month-length (date location)                      (18.13)
2    ;; TYPE (fixed-date location) -> 1..31
3    ;; Length of lunar month based on observability at location,
4    ;; which includes date.
5    (let* ((moon (phasis-on-or-after (1+ date) location))
6           (prev (phasis-on-or-before date location)))
7      (- moon prev)))

```

```

1  (defun early-month? (date location)                     (18.14)
2    ;; TYPE (fixed-date location) -> boolean
3    ;; Fixed date in location is in a month that was forced to
4    ;; start early.
5    (let* ((start (phasis-on-or-before date location))
6           (prev (- start 15)))
7      (or (>= (- date start) 30)
8          (> (month-length prev location) 30)
9          (and (= (month-length prev location) 30)
10              (early-month? prev location)))))

```

```

1  (defun alt-fixed-from-observational-islamic (i-date)    (18.15)
2    ;; TYPE islamic-date -> fixed-date
3    ;; Fixed date equivalent to Observational Islamic i-date.
4    ;; Months are never longer than 30 days.
5    (let* ((month (standard-month i-date))

```

```

6      (day (standard-day i-date))
7      (year (standard-year i-date))
8      (midmonth ; Middle of given month.
9      (+ islamic-epoch
10      (floor (* (+ (* (1- year) 12)
11      month -1/2)
12      mean-synodic-month))))
13      (moon (phasis-on-or-before ; First day of month.
14      midmonth islamic-location))
15      (date (+ moon day -1)))
16      (if (early-month? midmonth islamic-location) (1- date) date)))

```

```

1  (defun alt-observational-islamic-from-fixed (date)          (18.16)
2  ;; TYPE fixed-date -> islamic-date
3  ;; Observational Islamic date (year month day)
4  ;; corresponding to fixed date.
5  ;; Months are never longer than 30 days.
6  (let* ((early (early-month? date islamic-location))
7  (long (and early
8  (> (month-length date islamic-location) 29)))
9  (date-prime
10  (if long (1+ date) date))
11  (moon ; Most recent new moon.
12  (phasis-on-or-before date-prime islamic-location))
13  (elapsed-months
14  (round / (- moon islamic-epoch)
15  mean-synodic-month)))
16  (year (1+ (quotient elapsed-months 12)))
17  (month (1+ (mod elapsed-months 12)))
18  (day (- date-prime moon
19  (if (and early (not long)) -2 -1))))
20  (islamic-date year month day)))

```

```

1  (defun saudi-criterion (date)          (18.17)
2  ;; TYPE fixed-date -> boolean

```

```

3  ;; Saudi visibility criterion on eve of fixed date in Mecca.
4  (let* ((set (sunset (1- date) mecca))
5  (tee (universal-from-standard set mecca))
6  (phase (lunar-phase tee)))
7  (and (< new phase first-quarter)
8  (> (moonlag (1- date) mecca) 0))))

```

```

1  (defun saudi-new-month-on-or-before (date)          (18.18)
2  ;; TYPE fixed-date -> fixed-date
3  ;; Closest fixed date on or before date when Saudi
4  ;; visibility criterion held.
5  (let* ((moon ; Prior new moon.
6  (fixed-from-moment
7  (lunar-phase-at-or-before new date)))
8  (age (- date moon))
9  (tau ; Check if not visible yet on eve of date.
10  (if (and (<= age 3)
11  (not (saudi-criterion date)))
12  (- moon 30) ; Must go back a month.
13  moon)))
14  (next d tau (saudi-criterion d))))

```

```

1  (defun fixed-from-saudi-islamic (s-date)          (18.19)
2  ;; TYPE islamic-date -> fixed-date
3  ;; Fixed date equivalent to Saudi Islamic date s-date.
4  (let* ((month (standard-month s-date))
5  (day (standard-day s-date))
6  (year (standard-year s-date))
7  (midmonth ; Middle of given month.
8  (+ islamic-epoch
9  (floor (* (+ (* (1- year) 12)
10  month -1/2)
11  mean-synodic-month))))
12  (+ (saudi-new-month-on-or-before ; First day of month.
13  midmonth)
14  day -1)))

```

```

1 (defun saudi-islamic-from-fixed (date)
2   ;; TYPE fixed-date -> islamic-date
3   ;; Saudi Islamic date (year month day) corresponding to
4   ;; fixed date.
5   (let* ((crescent ; Most recent new moon.
6           (saudi-new-month-on-or-before date))
7           (elapsed-months
8            (round (/ (- crescent islamic-epoch)
9                       mean-synodic-month)))
10          (year (1+ (quotient elapsed-months 12)))
11          (month (1+ (mod elapsed-months 12)))
12          (day (1+ (- date crescent))))
13     (islamic-date year month day)))

```

```

1 (defconstant hebrew-location
2   ;; TYPE location
3   ;; Sample location for Observational Hebrew calendar
4   ;; (Haifa, Israel).
5   (location (deg 32.82L0) (deg 35) (mt 0) (hr 2)))

```

```

1 (defun observational-hebrew-first-of-nisan (g-year)
2   ;; TYPE gregorian-year -> fixed-date
3   ;; Fixed date of Observational (classical)
4   ;; Nisan 1 occurring in Gregorian year g-year.
5   (let* ((equinox ; Spring equinox.
6           (season-in-gregorian spring g-year))
7           (set ; Moment (UT) of sunset on day of equinox.
8            (universal-from-standard
9             (sunset (floor equinox) hebrew-location)
10             hebrew-location)))
11      (phasis-on-or-after
12       (- (floor equinox) ; Day of equinox
13          (if ; Spring starts before sunset.
14              (< equinox set) 14 13))
15       hebrew-location)))

```

```

(18.20) 1 (defun observational-hebrew-from-fixed (date)
2         ;; TYPE fixed-date -> hebrew-date
3         ;; Observational Hebrew date (year month day)
4         ;; corresponding to fixed date.
5         (let* ((crescent ; Most recent new moon.
6                 (phasis-on-or-before date hebrew-location))
7                 (g-year (gregorian-year-from-fixed date))
8                 (ny (observational-hebrew-first-of-nisan g-year))
9                 (new-year (if (< date ny)
10                              (observational-hebrew-first-of-nisan
11                               (1- g-year))
12                              ny))
13                 (month (1+ (round (/ (- crescent new-year) 29.5L0))))
14                 (year (+ (standard-year (hebrew-from-fixed new-year))
15                          (if (>= month tishri) 1 0)))
15                 (day (- date crescent -1)))
16            (hebrew-date year month day)))

```

```

(18.21) 1 (defun fixed-from-observational-hebrew (h-date)
2         ;; TYPE hebrew-date -> fixed-date
3         ;; Fixed date equivalent to Observational Hebrew date.
4         (let* ((month (standard-month h-date))
5                 (day (standard-day h-date))
6                 (year (standard-year h-date))
7                 (year1 (if (>= month tishri) (1- year) year))
8                 (start (fixed-from-hebrew
9                          (hebrew-date year1 nisan 1)))
10                (g-year (gregorian-year-from-fixed
11                          (+ start 60)))
12                (new-year (observational-hebrew-first-of-nisan g-year))
13                (midmonth ; Middle of given month.
14                          (+ new-year (round (* 29.5L0 (1- month))) 15)))
15            (+ (phasis-on-or-before ; First day of month.
16               midmonth hebrew-location)
17               day -1)))

```

```

1 (defun classical-passover-eve (g-year) (18.25) 5 (let* ((month (standard-month h-date))
2 ;; TYPE gregorian-year -> fixed-date 6 (day (standard-day h-date))
3 ;; Fixed date of Classical (observational) Passover Eve 7 (year (standard-year h-date))
4 ;; (Nisan 14) occurring in Gregorian year g-year. 8 (year1 (if (>= month tishri) (1- year) year))
5 (+ (observational-hebrew-first-of-nisan g-year) 13)) 9 (start (fixed-from-hebrew
10 (hebrew-date year1 nisan 1)))
11 (g-year (gregorian-year-from-fixed
12 (+ start 60)))
13 (new-year (observational-hebrew-first-of-nisan g-year))
14 (midmonth ; Middle of given month.
15 (+ new-year (round (* 29.5L0 (1- month))) 15))
16 (moon (phasis-on-or-before ; First day of month.
17 midmonth hebrew-location))
18 (date (+ moon day -1)))
19 (if (early-month? midmonth hebrew-location) (1- date) date)))

1 (defun alt-observational-hebrew-from-fixed (date) (18.26)
2 ;; TYPE fixed-date -> hebrew-date
3 ;; Observational Hebrew date (year month day)
4 ;; corresponding to fixed date.
5 ;; Months are never longer than 30 days.
6 (let* ((early (early-month? date hebrew-location))
7 (long (and early (> (month-length date hebrew-location) 29)))
8 (date-prime
9 (if long (1+ date) date))
10 (moon ; Most recent new moon.
11 (phasis-on-or-before date-prime hebrew-location))
12 (g-year (gregorian-year-from-fixed date-prime))
13 (ny (observational-hebrew-first-of-nisan g-year))
14 (new-year (if (< date-prime ny)
15 (observational-hebrew-first-of-nisan
16 (1- g-year))
17 ny))
18 (month (1+ (round (/ (- moon new-year) 29.5L0))))
19 (year (+ (standard-year (hebrew-from-fixed new-year))
20 (if (>= month tishri) 1 0)))
21 (day (- date-prime moon
22 (if (and early (not long)) -2 -1))))
23 (hebrew-date year month day)))

1 (defun alt-fixed-from-observational-hebrew (h-date) (18.27)
2 ;; TYPE hebrew-date -> fixed-date
3 ;; Fixed date equivalent to Observational Hebrew h-date.
4 ;; Months are never longer than 30 days.

5 (let* ((month (standard-month h-date))
6 (day (standard-day h-date))
7 (year (standard-year h-date))
8 (year1 (if (>= month tishri) (1- year) year))
9 (start (fixed-from-hebrew
10 (hebrew-date year1 nisan 1)))
11 (g-year (gregorian-year-from-fixed
12 (+ start 60)))
13 (new-year (observational-hebrew-first-of-nisan g-year))
14 (midmonth ; Middle of given month.
15 (+ new-year (round (* 29.5L0 (1- month))) 15))
16 (moon (phasis-on-or-before ; First day of month.
17 midmonth hebrew-location))
18 (date (+ moon day -1)))
19 (if (early-month? midmonth hebrew-location) (1- date) date)))

1 (defconstant samaritan-location (18.28)
2 ;; TYPE location
3 ;; Location of Mt. Gerizim.
4 (location (deg 32.1994) (deg 35.2728) (mt 881) (hr 2)))

1 (defun samaritan-noon (date) (18.29)
2 ;; TYPE fixed-date -> moment
3 ;; Universal time of true noon on date at Samaritan location.
4 (midday date samaritan-location))

1 (defun samaritan-new-moon-after (tee) (18.30)
2 ;; TYPE moment -> fixed-date
3 ;; Fixed date of first new moon after UT moment tee.
4 ;; Modern calculation.
5 (ceiling
6 (- (apparent-from-universal (new-moon-at-or-after tee)
7 samaritan-location)
8 (hr 12))))

```

```

1 (defun samaritan-new-moon-at-or-before (tee) (18.31) 3 ;; Fixed date of Samaritan date h-date.
2 ;; TYPE moment -> fixed-date 4 (let* ((month (standard-month s-date))
3 ;; Fixed-date of last new moon before UT moment tee. 5 (day (standard-day s-date))
4 ;; Modern calculation. 6 (year (standard-year s-date))
5 (ceiling 7 (ny (samaritan-new-year-on-or-before
6 (- (apparent-from-universal (new-moon-before tee) 8 (floor (+ samaritan-epoch 50
7 samaritan-location) 9 (* 365.25L0 (- year
8 (hr 12)))) 10 (ceiling (- month 5) 8))))))
11 (nm (samaritan-new-moon-at-or-before
12 (+ ny (* 29.5L0 (1- month)) 15))))
13 (+ nm day -1)))

1 (defconstant samaritan-epoch (18.32)
2 ;; TYPE fixed-date
3 ;; Fixed date of start of the Samaritan Entry Era.
4 (fixed-from-julian (julian-date (bce 1639) march 15)))

1 (defun samaritan-new-year-on-or-before (date) (18.33)
2 ;; TYPE fixed-date -> fixed-date
3 ;; Fixed date of Samaritan New Year on or before fixed
4 ;; date.
5 (let* ((g-year (gregorian-year-from-fixed date))
6 (dates ; All possible March 11's.
7 (append
8 (julian-in-gregorian march 11 (1- g-year))
9 (julian-in-gregorian march 11 g-year)
10 (list (1+ date)))) ; Extra to stop search.
11 (n
12 (final i 0
13 (<= (samaritan-new-moon-after
14 (samaritan-noon (nth i dates)))
15 date))))
16 (samaritan-new-moon-after (samaritan-noon (nth n dates)))))

1 (defun fixed-from-samaritan (s-date) (18.34)
2 ;; TYPE hebrew-date -> fixed-date
3 (first date))

1 (defun samaritan-from-fixed (date) (18.35)
2 ;; TYPE fixed-date -> hebrew-date
3 ;; Samaritan date corresponding to fixed date.
4 (let* ((moon ; First of month
5 (samaritan-new-moon-at-or-before
6 (samaritan-noon date)))
7 (new-year (samaritan-new-year-on-or-before moon))
8 (month (1+ (round (/ (- moon new-year) 29.5L0))))
9 (year (+ (round (/ (- new-year samaritan-epoch) 365.25L0))
10 (ceiling (- month 5) 8)))
11 (day (- date moon -1)))
12 (hebrew-date year month day)))

```

D.19 The Chinese Calendar

```

1 (defun chinese-date (cycle year month leap day)
2 ;; TYPE (chinese-cycle chinese-year chinese-month
3 ;; TYPE chinese-leap chinese-day) -> chinese-date
4 (list cycle year month leap day))

1 (defun chinese-cycle (date)
2 ;; TYPE chinese-date -> chinese-cycle
3 (first date))

```

```

1 (defun chinese-year (date)
2   ;; TYPE chinese-date -> chinese-year
3   (second date))

```

```

1 (defun chinese-month (date)
2   ;; TYPE chinese-date -> chinese-month
3   (third date))

```

```

1 (defun chinese-leap (date)
2   ;; TYPE chinese-date -> chinese-leap
3   (fourth date))

```

```

1 (defun chinese-day (date)
2   ;; TYPE chinese-date -> chinese-day
3   (fifth date))

```

```

1 (defun current-major-solar-term (date)
2   ;; TYPE fixed-date -> integer
3   ;; Last Chinese major solar term (zhongqi) before fixed
4   ;; date.
5   (let* ((s (solar-longitude
6             (universal-from-standard
7              date
8              (chinese-location date))))))
9     (amod (+ 2 (quotient s (deg 30))) 12)))

```

```

1 (defun chinese-location (tee)
2   ;; TYPE moment -> location
3   ;; Location of Beijing; time zone varies with tee.
4   (let* ((year (gregorian-year-from-fixed (floor tee))))
5     (if (< year 1929)

```

```

6       (location (angle 39 55 0) (angle 116 25 0)
7                 (mt 43.5) (hr 1397/180))
8       (location (angle 39 55 0) (angle 116 25 0)
9                 (mt 43.5) (hr 8))))))

```

```

1 (defun chinese-solar-longitude-on-or-after (lambda tee)
2   ;; TYPE (season moment) -> moment
3   ;; Moment (Beijing time) of the first time at or after
4   ;; tee (Beijing time) when the solar longitude
5   ;; will be lambda degrees.
6   (let* ((sun (solar-longitude-after
7                 lambda
8                 (universal-from-standard
9                  tee
10                  (chinese-location tee))))))
11     (standard-from-universal
12      sun
13      (chinese-location sun))))

```

(19.1)

```

1 (defun major-solar-term-on-or-after (date)
2   ;; TYPE fixed-date -> moment
3   ;; Moment (in Beijing) of the first Chinese major
4   ;; solar term (zhongqi) on or after fixed date. The
5   ;; major terms begin when the sun's longitude is a
6   ;; multiple of 30 degrees.
7   (let* ((s (solar-longitude (midnight-in-china date)))
8          (l (mod (* 30 (ceiling (/ s 30))) 360)))
9     (chinese-solar-longitude-on-or-after l date))

```

(19.2)

```

1 (defun current-minor-solar-term (date)
2   ;; TYPE fixed-date -> integer
3   ;; Last Chinese minor solar term (jieqi) before date.
4   (let* ((s (solar-longitude

```

```

5      (universal-from-standard
6      date
7      (chinese-location date))))
8      (amod (+ 3 (quotient (- s (deg 15)) (deg 30)))
9      12)))

```

```

1  (defun minor-solar-term-on-or-after (date)          (19.6)
2  ;; TYPE fixed-date -> moment
3  ;; Moment (in Beijing) of the first Chinese minor solar
4  ;; term (jieqi) on or after fixed date. The minor terms
5  ;; begin when the sun's longitude is an odd multiple of 15
6  ;; degrees.
7  (let* ((s (solar-longitude (midnight-in-china date)))
8         (l (mod
9              (+ (* 30
10                 (ceiling
11                  (/ (- s (deg 15)) 30)))
12                  (deg 15))
13              360)))
14    (chinese-solar-longitude-on-or-after 1 date)))

```

```

1  (defun midnight-in-china (date)                    (19.7)
2  ;; TYPE fixed-date -> moment
3  ;; Universal time of (clock) midnight at start of fixed
4  ;; date in China.
5  (universal-from-standard date (chinese-location date)))

```

```

1  (defun chinese-winter-solstice-on-or-before (date) (19.8)
2  ;; TYPE fixed-date -> fixed-date
3  ;; Fixed date, in the Chinese zone, of winter solstice
4  ;; on or before fixed date.
5  (let* ((approx ; Approximate time of solstice.
6          (estimate-prior-solar-longitude

```

```

7          winter (midnight-in-china (+ date 1))))
8          (next day (1- (floor approx)))
9          (< winter (solar-longitude
10             (midnight-in-china (1+ day))))))

```

```

1  (defun chinese-new-moon-on-or-after (date)          (19.9)
2  ;; TYPE fixed-date -> fixed-date
3  ;; Fixed date (Beijing) of first new moon on or after
4  ;; fixed date.
5  (let* ((tee (new-moon-at-or-after
6               (midnight-in-china date))))
7          (floor
8           (standard-from-universal
9            tee
10             (chinese-location tee)))))

```

```

1  (defun chinese-new-moon-before (date)              (19.10)
2  ;; TYPE fixed-date -> fixed-date
3  ;; Fixed date (Beijing) of first new moon before fixed
4  ;; date.
5  (let* ((tee (new-moon-before
6               (midnight-in-china date))))
7          (floor
8           (standard-from-universal
9            tee
10             (chinese-location tee)))))

```

```

1  (defun chinese-no-major-solar-term? (date)         (19.11)
2  ;; TYPE fixed-date -> boolean
3  ;; True if Chinese lunar month starting on date
4  ;; has no major solar term.
5  (= (current-major-solar-term date)
6     (current-major-solar-term
7      (chinese-new-moon-on-or-after (+ date 1))))

```

```

1 (defun chinese-prior-leap-month? (m-prime m)
2   ;; TYPE (fixed-date fixed-date) -> boolean
3   ;; True if there is a Chinese leap month on or after lunar
4   ;; month starting on fixed day m-prime and at or before
5   ;; lunar month starting at fixed date m.
6   (and (>= m m-prime)
7        (or (chinese-no-major-solar-term? m)
8            (chinese-prior-leap-month?
9              m-prime
10              (chinese-new-moon-before m))))))

```

```

1 (defun chinese-new-year-in-sui (date)
2   ;; TYPE fixed-date -> fixed-date
3   ;; Fixed date of Chinese New Year in sui (period from
4   ;; solstice to solstice) containing date.
5   (let* ((s1; prior solstice
6           (chinese-winter-solstice-on-or-before date))
7          (s2; following solstice
8           (chinese-winter-solstice-on-or-before
9             (+ s1 370)))
10          (m12 ; month after 11th month--either 12 or leap 11
11            (chinese-new-moon-on-or-after (1+ s1)))
12          (m13 ; month after m12--either 12 (or leap 12) or 1
13            (chinese-new-moon-on-or-after (1+ m12)))
14          (next-m11 ; next 11th month
15            (chinese-new-moon-before (1+ s2))))
16     (if ; Either m12 or m13 is a leap month if there are
17         ; 13 new moons (12 full lunar months) and
18         ; either m12 or m13 has no major solar term
19         (and (= (round (/ (- next-m11 m12)
20                             mean-synodic-month))
21                12)
22              (or (chinese-no-major-solar-term? m12)
23                  (chinese-no-major-solar-term? m13)))
24         (chinese-new-moon-on-or-after (1+ m13)))
25     m13)))

```

```

(19.12) 1 (defun chinese-new-year-on-or-before (date)
2         ;; TYPE fixed-date -> fixed-date
3         ;; Fixed date of Chinese New Year on or before fixed date.
4         (let* ((new-year (chinese-new-year-in-sui date)))
5           (if (>= date new-year)
6               new-year
7               ;; Got the New Year after--this happens if date is
8               ;; after the solstice but before the new year.
9               ;; So, go back half a year.
10              (chinese-new-year-in-sui (- date 180)))))

```

```

(19.13) 1 (defconstant chinese-epoch
2         ;; TYPE fixed-date
3         ;; Fixed date of start of the Chinese calendar.
4         (fixed-from-gregorian (gregorian-date -2636 february 15)))

```

```

(19.16) 1 (defun chinese-from-fixed (date)
2         ;; TYPE fixed-date -> chinese-date
3         ;; Chinese date (cycle year month leap day) of fixed date.
4         (let* ((s1; Prior solstice
5                 (chinese-winter-solstice-on-or-before date))
6                (s2; Following solstice
7                 (chinese-winter-solstice-on-or-before (+ s1 370)))
8                (m12 ; month after last 11th month
9                 (chinese-new-moon-on-or-after (1+ s1)))
10               (next-m11; next 11th month
11                 (chinese-new-moon-before (1+ s2)))
12               (m ; start of month containing date
13                 (chinese-new-moon-before (1+ date)))
14               (leap-year; if there are 13 new moons (12 full
15                 ; lunar months)
16                 (= (round (/ (- next-m11 m12)
17                             mean-synodic-month))
18                    12)))

```

```

18      12))
19      (month ; month number
20      (amod
21      (-
22      ;; ordinal position of month in year
23      (round (/ (- m m12) mean-synodic-month))
24      ;; minus 1 during or after a leap month
25      (if (and leap-year
26            (chinese-prior-leap-month? m12 m))
27          1
28          0))
29      12))
30      (leap-month ; it's a leap month if...
31      (and
32      leap-year; ...there are 13 months
33      (chinese-no-major-solar-term?
34      m) ; no major solar term
35      (not (chinese-prior-leap-month? ; and no prior leap
36            month
37            m12 (chinese-new-moon-before m))))
38      (elapsed-years ; Approximate since the epoch
39      (floor (+ 1.5L0 ; 18 months (because of truncation)
40              (- (/ month 12)); after at start of year
41              (/ (- date chinese-epoch)
42                mean-tropical-year))))
43      (cycle (1+ (quotient (1- elapsed-years) 60)))
44      (year (amod elapsed-years 60))
45      (day (1+ (- date m))))
46      (chinese-date cycle year month leap-month day)))

```

```

1  (defun fixed-from-chinese (c-date)
2    ;; TYPE chinese-date -> fixed-date
3    ;; Fixed date of Chinese date c-date.
4    (let* ((cycle (chinese-cycle c-date))
5           (year (chinese-year c-date))

```

(19.17)

```

6      (month (chinese-month c-date))
7      (leap (chinese-leap c-date))
8      (day (chinese-day c-date))
9      (mid-year ; Middle of the Chinese year
10     (floor
11     (+ chinese-epoch
12       (* (+ (* (1- cycle) 60); years in prior cycles
13          (1- year) ; prior years this cycle
14          1/2) ; half a year
15       mean-tropical-year))))
16     (new-year (chinese-new-year-on-or-before mid-year))
17     (p ; new moon before date--a month too early if
18     ; there was prior leap month that year
19     (chinese-new-moon-on-or-after
20     (+ new-year (* (1- month) 29))))
21     (d (chinese-from-fixed p))
22     (prior-new-moon
23     (if ; If the months match...
24         (and (= month (chinese-month d))
25              (equal leap (chinese-leap d)))
26         p; ...that's the right month
27     ;; otherwise, there was a prior leap month that
28     ;; year, so we want the next month
29     (chinese-new-moon-on-or-after (1+ p))))
30     (+ prior-new-moon day -1)))

```

```

1  (defun chinese-name (stem branch)
2    ;; TYPE (chinese-stem chinese-branch) -> chinese-name
3    ;; Combination is impossible if stem and branch
4    ;; are not the equal mod 2.
5    (list stem branch))

```

```

1  (defun chinese-stem (name)
2    ;; TYPE chinese-name -> chinese-stem
3    (first name))

```

```

1 (defun chinese-branch (name)
2   ;; TYPE chinese-name -> chinese-branch
3   (second name))

1 (defun chinese-sexagesimal-name (n)
2   ;; TYPE integer -> chinese-name
3   ;; The  $n$ -th name of the Chinese sexagesimal cycle.
4   (chinese-name (amod n 10)
5                 (amod n 12)))

1 (defun chinese-name-difference (c-name1 c-name2)
2   ;; TYPE (chinese-name chinese-name) -> nonnegative-integer
3   ;; Number of names from Chinese name  $c\text{-name1}$  to the
4   ;; next occurrence of Chinese name  $c\text{-name2}$ .
5   (let* ((stem1 (chinese-stem c-name1))
6          (stem2 (chinese-stem c-name2))
7          (branch1 (chinese-branch c-name1))
8          (branch2 (chinese-branch c-name2))
9          (stem-difference (- stem2 stem1))
10         (branch-difference (- branch2 branch1)))
11     (amod (+ stem-difference
12             (* 25 (- branch-difference
13                     stem-difference)))
14           60)))

1 (defun chinese-year-name (year)
2   ;; TYPE chinese-year -> chinese-name
3   ;; Sexagesimal name for Chinese year of any cycle.
4   (chinese-sexagesimal-name year))

1 (defconstant chinese-month-name-epoch
2   ;; TYPE integer
3   ;; Elapsed months at start of Chinese sexagesimal month
4   ;; cycle.
5   57)

```

(19.18)

(19.19)

(19.20)

(19.21)

```

1 (defun chinese-month-name (month year)
2   ;; TYPE (chinese-month chinese-year) -> chinese-name
3   ;; Sexagesimal name for month  $month$  of Chinese year
4   ;;  $year$ .
5   (let* ((elapsed-months (+ (* 12 (1- year))
6                             (1- month))))
7     (chinese-sexagesimal-name
8       (- elapsed-months chinese-month-name-epoch)))

```

(19.22)

```

1 (defconstant chinese-day-name-epoch
2   ;; TYPE integer
3   ;; RD date of a start of Chinese sexagesimal day cycle.
4   (rd 45))

```

(19.23)

```

1 (defun chinese-day-name (date)
2   ;; TYPE fixed-date -> chinese-name
3   ;; Chinese sexagesimal name for  $date$ .
4   (chinese-sexagesimal-name
5     (- date chinese-day-name-epoch)))

```

(19.24)

```

1 (defun chinese-day-name-on-or-before (name date)
2   ;; TYPE (chinese-name fixed-date) -> fixed-date
3   ;; Fixed date of latest date on or before fixed  $date$ 
4   ;; that has Chinese  $name$ .
5   (mod3 (chinese-name-difference
6         (chinese-day-name 0) name)
7         date (- date 60)))

```

(19.25)

```

1 (defun chinese-new-year (g-year) (19.26) 6 (let* ((today (chinese-from-fixed date)))
2 ;; TYPE gregorian-year -> fixed-date 7 (if (>= date (fixed-from-chinese birthdate))
3 ;; Fixed date of Chinese New Year in Gregorian year g-year. 8 (+ (* 60 (- (chinese-cycle today)
4 (chinese-new-year-on-or-before 9 (chinese-cycle birthdate)))
5 (fixed-from-gregorian 10 (- (chinese-year today)
6 (gregorian-date g-year july 1)))) 11 (chinese-year birthdate))
12 1)
13 bogus)))

1 (defun dragon-festival (g-year) (19.27)
2 ;; TYPE gregorian-year -> fixed-date
3 ;; Fixed date of the Dragon Festival occurring in
4 ;; Gregorian year g-year.
5 (let* ((elapsed-years
6 (1+ (- g-year
7 (gregorian-year-from-fixed
8 chinese-epoch))))
9 (cycle (1+ (quotient (1- elapsed-years) 60)))
10 (year (amod elapsed-years 60)))
11 (fixed-from-chinese (chinese-date cycle year 5 false 5))))

1 (defun qing-ming (g-year) (19.28) 1 (defconstant blind (19.32)
2 ;; TYPE gregorian-year -> fixed-date 2 ;; TYPE augury
3 ;; Fixed date of Qingming occurring in Gregorian year 3 ;; Lichun occurs once at the end.
4 ;; g-year. 4 1)
5 (floor
6 (minor-solar-term-on-or-after
7 (fixed-from-gregorian
8 (gregorian-date g-year march 30)))))

1 (defun chinese-age (birthdate date) (19.29)
2 ;; TYPE (chinese-date fixed-date) -> nonnegative-integer
3 ;; Age at fixed date, given Chinese birthdate,
4 ;; according to the Chinese custom. Returns bogus if
5 ;; date is before birthdate.

1 (defconstant double-bright (19.30)
2 ;; TYPE augury
3 ;; Lichun occurs twice (double-happiness).
4 3)

1 (defconstant bright (19.31)
2 ;; TYPE augury
3 ;; Lichun occurs once at the start.
4 2)

1 (defconstant widow (19.33)
2 ;; TYPE augury
3 ;; Lichun does not occur (double-blind year).
4 0)

1 (defun chinese-year-marriage-augury (cycle year) (19.34)
2 ;; TYPE (chinese-cycle chinese-year) -> augury
3 ;; The marriage augury type of Chinese year in cycle.
4 (let* ((new-year (fixed-from-chinese

```

```

5         (chinese-date cycle year 1 false 1)))
6     (c (if (= year 60); next year's cycle
7         (1+ cycle)
8         cycle))
9     (y (if (= year 60); next year's number
10        1
11        (1+ year)))
12     (next-new-year (fixed-from-chinese
13                    (chinese-date c y 1 false 1)))
14     (first-minor-term
15      (current-minor-solar-term new-year))
16     (next-first-minor-term
17      (current-minor-solar-term next-new-year)))
18     (cond
19      ((and
20       (= first-minor-term 1)          ; no lichun at start...
21       (= next-first-minor-term 12)) ; ...or at end
22       widow)
23      ((and
24       (= first-minor-term 1)          ; no lichun at start...
25       (/= next-first-minor-term 12)); ...only at end
26       blind)
27      ((and
28       (/= first-minor-term 1)          ; lichun at start...
29       (= next-first-minor-term 12)) ; ... not at end
30       bright)
31      (t double-bright))))          ; lichun at start and end

```

```

1 (defun japanese-location (tee)          (19.35)
2   ;; TYPE moment -> location
3   ;; Location for Japanese calendar; varies with tee.
4   (let* ((year (gregorian-year-from-fixed (floor tee))))
5     (if (< year 1888)
6         ;; Tokyo (139 deg 46 min east) local time
7         (location (deg 35.7L0) (angle 139 46 0)

```

```

8         (mt 24) (hr (+ 9 143/450)))
9         ; Longitude 135 time zone
10        (location (deg 35) (deg 135) (mt 0) (hr 9))))))

```

```

1 (defun korean-location (tee)          (19.36)
2   ;; TYPE moment -> location
3   ;; Location for Korean calendar; varies with tee.
4   ;; Seoul city hall at a varying time zone.
5   (let* ((z (cond
6             ((< tee
7              (fixed-from-gregorian
8               (gregorian-date 1908 april 1)))
9              ; local mean time for longitude 126 deg 58 min
10              3809/450)
11             ((< tee
12              (fixed-from-gregorian
13               (gregorian-date 1912 january 1)))
14              8.5)
15             ((< tee
16              (fixed-from-gregorian
17               (gregorian-date 1954 march 21)))
18              9)
19             ((< tee
20              (fixed-from-gregorian
21               (gregorian-date 1961 august 10)))
22              8.5)
23             (t 9))))
24         (location (angle 37 34 0) (angle 126 58 0)
25                   (mt 0) (hr z))))

```

```

1 (defun korean-year (cycle year)          (19.37)
2   ;; TYPE (chinese-cycle chinese-year) -> integer
3   ;; Equivalent Korean year to Chinese cycle and year
4   (+ (* 60 cycle) year -364))

```

```

1 (defun vietnamese-location (tee)
2   ;; TYPE moment -> location
3   ;; Location for Vietnamese calendar is Hanoi; varies with
4   ;; tee. Time zone has changed over the years.
5   (let* ((z (if (< tee
6                 (gregorian-new-year 1968))
7                 8
8                 7)))
9     (location (angle 21 2 0) (angle 105 51 0)
10              (mt 12) (hr z))))

```

(19.38)

D.20 The Modern Hindu Calendars

Common Lisp supplies arithmetic with arbitrary rational numbers, and we take advantage of this for implementing the Hindu calendars. With other languages, 64-bit arithmetic is required for many of the calculations.

```

1 (defconstant hindu-sidereal-year
2   ;; TYPE rational
3   ;; Mean length of Hindu sidereal year.
4   (+ 365 279457/1080000))

```

(20.1)

```

1 (defconstant hindu-sidereal-month
2   ;; TYPE rational
3   ;; Mean length of Hindu sidereal month.
4   (+ 27 4644439/14438334))

```

(20.2)

```

1 (defconstant hindu-synodic-month
2   ;; TYPE rational
3   ;; Mean time from new moon to new moon.
4   (+ 29 7087771/13358334))

```

(20.3)

```

1 (defun hindu-sine-table (entry)
2   ;; TYPE integer -> rational-amplitude
3   ;; This simulates the Hindu sine table.
4   ;; entry is an angle given as a multiplier of 225'.
5   (let* ((exact (* 3438 (sin-degrees
6                      (* entry (angle 0 225 0))))))
7     (error (* 0.215L0 (sign exact)
8                  (sign (- (abs exact) 1716)))))
9     (/ (round (+ exact error)) 3438)))

```

(20.4)

```

1 (defun hindu-sine (theta)
2   ;; TYPE rational-angle -> rational-amplitude
3   ;; Linear interpolation for theta in Hindu table.
4   (let* ((entry
5           (/ theta (angle 0 225 0))); Interpolate in table.
6           (fraction (mod entry 1)))
7     (+ (* fraction
8           (hindu-sine-table (ceiling entry)))
9       (* (- 1 fraction)
10          (hindu-sine-table (floor entry))))))

```

(20.5)

```

1 (defun hindu-arcsin (amp)
2   ;; TYPE rational-amplitude -> rational-angle
3   ;; Inverse of Hindu sine function of amp.
4   (if (< amp 0) (- (hindu-arcsin (- amp))))
5   (let* ((pos (next k 0 (<= amp (hindu-sine-table k))))
6           (below ; Lower value in table.
7                 (hindu-sine-table (1- pos))))
8     (* (angle 0 225 0)
9        (+ pos -1 ; Interpolate.
10           (/ (- amp below)
11              (- (hindu-sine-table pos) below))))))

```

(20.6)

```

1 (defun hindu-mean-position (tee period)
2   ;; TYPE (rational-moment rational) -> rational-angle
3   ;; Position in degrees at moment tee in uniform circular
4   ;; orbit of period days.
5   (* (deg 360) (mod (/ (- tee hindu-creation) period) 1)))

```

(20.7)

```

12 (hindu-sine (hindu-mean-position tee anomalistic)))
13 (contraction (* (abs offset) change size))
14 (equation ; Equation of center
15   (hindu-arcsin (* offset (- size contraction)))))
16 (mod (- lambda equation 360)))

```

```

1 (defconstant hindu-creation
2   ;; TYPE fixed-date
3   ;; Fixed date of Hindu creation.
4   (- hindu-epoch (* 1955880000 hindu-sidereal-year)))

```

(20.8)

```

1 (defun hindu-solar-longitude (tee) (20.12)
2   ;; TYPE rational-moment -> rational-angle
3   ;; Solar longitude at moment tee.
4   (hindu-true-position tee hindu-sidereal-year
5     14/360 hindu-anomalistic-year 1/42))

```

```

1 (defconstant hindu-anomalistic-year
2   ;; TYPE rational
3   ;; Time from aphelion to aphelion.
4   (/ 1577917828000 (- 4320000000 387)))

```

(20.9)

```

1 (defun hindu-zodiac (tee) (20.13)
2   ;; TYPE rational-moment -> hindu-solar-month
3   ;; Zodiacal sign of the sun, as integer in range 1..12,
4   ;; at moment tee.
5   (1+ (quotient (hindu-solar-longitude tee) (deg 30))))

```

```

1 (defconstant hindu-anomalistic-month
2   ;; TYPE rational
3   ;; Time from apogee to apogee, with bija correction.
4   (/ 1577917828 (- 57753336 488199)))

```

(20.10)

```

1 (defun hindu-lunar-longitude (tee) (20.14)
2   ;; TYPE rational-moment -> rational-angle
3   ;; Lunar longitude at moment tee.
4   (hindu-true-position tee hindu-sidereal-month
5     32/360 hindu-anomalistic-month 1/96))

```

```

1 (defun hindu-true-position (tee period size anomalistic change) (20.11)
2   ;; TYPE (rational-moment rational rational rational
3   ;; TYPE rational) -> rational-angle
4   ;; Longitudinal position at moment tee. period is
5   ;; period of mean motion in days. size is ratio of
6   ;; radii of epicycle and deferent. anomalistic is the
7   ;; period of retrograde revolution about epicycle.
8   ;; change is maximum decrease in epicycle size.
9   (let* ((lambda ; Position of epicycle center
10    (hindu-mean-position tee period))
11    (offset ; Sine of anomaly

```

```

1 (defun hindu-lunar-phase (tee) (20.15)
2   ;; TYPE rational-moment -> rational-angle
3   ;; Longitudinal distance between the sun and moon
4   ;; at moment tee.
5   (mod (- (hindu-lunar-longitude tee)
6    (hindu-solar-longitude tee))
7     360))

```

```

1 (defun hindu-lunar-day-from-moment (tee)
2   ;; TYPE rational-moment -> hindu-lunar-day
3   ;; Phase of moon (tithi) at moment tee, as an integer in
4   ;; the range 1..30.
5   (1+ (quotient (hindu-lunar-phase tee) (deg 12))))

```

(20.16)

```

1 (defun hindu-new-moon-before (tee)
2   ;; TYPE rational-moment -> rational-moment
3   ;; Approximate moment of last new moon preceding moment
4   ;; tee, close enough to determine zodiacal sign.
5   (let* ((varepsilon (expt 2 -1000)) ; Safety margin.
6          (tau ; Can be off by almost a day.
7            (- tee (* (/ 1 (deg 360)) (hindu-lunar-phase tee)
8                      hindu-synodic-month))))
9     (binary-search ; Search for phase start.
10      1 (1- tau)
11      u (min tee (1+ tau))
12      x (< (hindu-lunar-phase x) (deg 180))
13      (or (= (hindu-zodiac 1) (hindu-zodiac u))
14          (< (- u 1) varepsilon))))

```

(20.17)

```

1 (defun hindu-solar-date (year month day)
2   ;; TYPE (hindu-solar-year hindu-solar-month hindu-solar-day)
3   ;; TYPE -> hindu-solar-date
4   (list year month day))

```

```

1 (defun hindu-calendar-year (tee)
2   ;; TYPE rational-moment -> hindu-solar-year
3   ;; Determine solar year at given moment tee.
4   (round (- (/ (- tee hindu-epoch)
5                hindu-sidereal-year)
6            (/ (hindu-solar-longitude tee)
7              (deg 360))))

```

(20.18)

```

1 (defconstant hindu-solar-era
2   ;; TYPE standard-year
3   ;; Years from Kali Yuga until Saka era.
4   3179)

```

(20.19)

```

1 (defun hindu-solar-from-fixed (date)
2   ;; TYPE fixed-date -> hindu-solar-date
3   ;; Hindu (Orissa) solar date equivalent to fixed date.
4   (let* ((critical ; Sunrise on Hindu date.
5           (hindu-sunrise (1+ date)))
6          (month (hindu-zodiac critical))
7          (year (- (hindu-calendar-year critical)
8                  hindu-solar-era))
9          (approx ; 3 days before start of mean month.
10            (- date 3
11              (mod (floor (hindu-solar-longitude critical))
12                  (deg 30)))))
12     (start ; Search forward for beginning...
13     (next i approx ; ... of month.
14         (= (hindu-zodiac (hindu-sunrise (1+ i)))
15           month)))
16     (day (- date start -1)))
17     (hindu-solar-date year month day)))

```

(20.20)

```

1 (defun fixed-from-hindu-solar (s-date)
2   ;; TYPE hindu-solar-date -> fixed-date
3   ;; Fixed date corresponding to Hindu solar date s-date
4   ;; (Saka era; Orissa rule.)
5   (let* ((month (standard-month s-date))
6          (day (standard-day s-date))
7          (year (standard-year s-date))
8          (start ; Approximate start of month
9                ; by adding days...
10            (+ (floor (* (+ year hindu-solar-era

```

(20.21)

```

11          (/ (1- month) 12)) ; in months...
12          hindu-sidereal-year)) ; ... and years
13          hindu-epoch))) ; and days before RD 0.
14      ;; Search forward to correct month
15      (+ day -1
16         (next d (- start 3)
17                (= (hindu-zodiac (hindu-sunrise (1+ d)))
18                   month))))))

```

```

1 (defun hindu-lunar-date (year month leap-month day leap-day)
2   ;; TYPE (hindu-lunar-year hindu-lunar-month
3   ;; TYPE hindu-lunar-leap-month hindu-lunar-day
4   ;; TYPE hindu-lunar-leap-day) -> hindu-lunar-date
5   (list year month leap-month day leap-day))

```

```

1 (defun hindu-lunar-month (date)
2   ;; TYPE hindu-lunar-date -> hindu-lunar-month
3   (second date))

```

```

1 (defun hindu-lunar-leap-month (date)
2   ;; TYPE hindu-lunar-date -> hindu-lunar-leap-month
3   (third date))

```

```

1 (defun hindu-lunar-day (date)
2   ;; TYPE hindu-lunar-date -> hindu-lunar-day
3   (fourth date))

```

```

1 (defun hindu-lunar-leap-day (date)
2   ;; TYPE hindu-lunar-date -> hindu-lunar-leap-day
3   (fifth date))

```

```

1 (defun hindu-lunar-year (date)
2   ;; TYPE hindu-lunar-date -> hindu-lunar-year
3   (first date))

```

```

1 (defconstant hindu-lunar-era                                (20.22)
2   ;; TYPE standard-year
3   ;; Years from Kali Yuga until Vikrama era.
4   3044)

```

```

1 (defun hindu-lunar-from-fixed (date)                        (20.23)
2   ;; TYPE fixed-date -> hindu-lunar-date
3   ;; Hindu lunar date, new-moon scheme,
4   ;; equivalent to fixed date.
5   (let* ((critical (hindu-sunrise date)) ; Sunrise that day.
6          (day (hindu-lunar-day-from-moment
7                critical)); Day of month.
8          (leap-day ; If previous day the same.
9                   (= day (hindu-lunar-day-from-moment
10                          (hindu-sunrise (- date 1)))))
11         (last-new-moon
12          (hindu-new-moon-before critical))
13         (next-new-moon
14          (hindu-new-moon-before
15           (+ (floor last-new-moon) 35)))
16         (solar-month ; Solar month name.
17          (hindu-zodiac last-new-moon))
18         (leap-month ; If begins and ends in same sign.
19                    (= solar-month (hindu-zodiac next-new-moon)))
20         (month ; Month of lunar year.
21                (amod (1+ solar-month) 12))
22         (year ; Solar year at end of month.
23                (- (hindu-calendar-year
24                    (if (<= month 2) ; date might precede solar
25                        ; new year.

```

```

26      (+ date 180)
27      date))
28      hindu-lunar-era)))
29      (hindu-lunar-date year month leap-month day leap-day)))

```

```

1  (defun fixed-from-hindu-lunar (l-date)
2    ;; TYPE hindu-lunar-date -> fixed-date
3    ;; Fixed date corresponding to Hindu lunar date l-date.
4    (let* ((year (hindu-lunar-year l-date))
5           (month (hindu-lunar-month l-date))
6           (leap-month (hindu-lunar-leap-month l-date))
7           (day (hindu-lunar-day l-date))
8           (leap-day (hindu-lunar-leap-day l-date))
9           (approx
10            (+ hindu-epoch
11              (* hindu-sidereal-year
12                (+ year hindu-lunar-era
13                  (/ (1- month) 12))))))
14      (s (floor
15          (- approx
16             (* hindu-sidereal-year
17               (mod3 (- (/ (hindu-solar-longitude approx)
18                           (deg 360))
19                         (/ (1- month) 12))
20                           -1/2 1/2))))))
21      (k (hindu-lunar-day-from-moment (+ s (hr 6))))
22      (est
23       (- s (- day)
24           (cond
25            ((< 3 k 27) ; Not borderline case.
26             k)
27            ((let* ((mid ; Middle of preceding solar month.
28                    (hindu-lunar-from-fixed
29                     (- s 15))))
30              (or ; In month starting near s.

```

(20.24)

```

31      (/= (hindu-lunar-month mid) month)
32      (and (hindu-lunar-leap-month mid)
33            (not leap-month))))
34      (mod3 k -15 15))
35      (t ; In preceding month.
36       (mod3 k 15 45))))
37      (tau ; Refined estimate.
38       (- est (mod3 (- (hindu-lunar-day-from-moment
39                       (+ est (hr 6)))
40                         day)
41                 -15 15)))
42      (date (next d (1- tau)
43              (member (hindu-lunar-day-from-moment
44                       (hindu-sunrise d))
45                      (list day (amod (1+ day) 30))))))
46      (if leap-day (1+ date) date)))

```

```

1  (defconstant ujjain
2    ;; TYPE location
3    ;; Location of Ujjain.
4    (location (angle 23 9 0) (angle 75 46 6)
5              (mt 0) (hr (+ 5 461/9000))))

```

(20.25)

```

1  (defconstant hindu-location
2    ;; TYPE location
3    ;; Location (Ujjain) for determining Hindu calendar.
4    ujjain)

```

(20.26)

```

1  (defun hindu-ascensional-difference (date location)
2    ;; TYPE (fixed-date location) -> rational-angle
3    ;; Difference between right and oblique ascension
4    ;; of sun on date at location.
5    (let* ((sin_delta
6           (* 1397/3438 ; Sine of inclination.
7             (hindu-sine (hindu-tropical-longitude date))))

```

(20.27)

```

8      (phi (latitude location))
9      (diurnal-radius
10      (hindu-sine (+ (deg 90) (hindu-arcsin sin_delta))))
11      (tan_phi ; Tangent of latitude as rational number.
12      (/ (hindu-sine phi)
13      (hindu-sine (+ (deg 90) phi))))
14      (earth-sine (* sin_delta tan_phi)))
15      (hindu-arcsin (- (/ earth-sine diurnal-radius))))

```

```

1  (defun hindu-tropical-longitude (date)
2    ;; TYPE fixed-date -> rational-angle
3    ;; Hindu tropical longitude on fixed date.
4    ;; Assumes precession with maximum of 27 degrees
5    ;; and period of 7200 sidereal years
6    ;; (= 1577917828/600 days).
7    (let* ((days (- date hindu-epoch)) ; Whole days.
8    (precession
9    (- (deg 27)
10    (abs
11    (* (deg 108)
12    (mod3 (- (* 600/1577917828 days)
13    1/4)
14    -1/2 1/2)))))
15    (mod (- (hindu-solar-longitude date) precession)
16    360)))

```

(20.28)

```

1  (defun hindu-solar-sidereal-difference (date)
2    ;; TYPE fixed-date -> rational-angle
3    ;; Difference between solar and sidereal day on date.
4    (* (hindu-daily-motion date) (hindu-rising-sign date)))

```

(20.29)

```

1  (defun hindu-daily-motion (date)
2    ;; TYPE fixed-date -> rational-angle

```

(20.30)

```

3    ;; Sidereal daily motion of sun on date.
4    (let* ((mean-motion ; Mean daily motion in degrees.
5    (/ (deg 360) hindu-sidereal-year))
6    (anomaly
7    (hindu-mean-position date hindu-anomalistic-year))
8    (epicycle ; Current size of epicycle.
9    (- 14/360 (/ (abs (hindu-sine anomaly)) 1080)))
10    (entry (quotient anomaly (angle 0 225 0)))
11    (sine-table-step ; Marginal change in anomaly
12    (- (hindu-sine-table (1+ entry))
13    (hindu-sine-table entry)))
14    (factor
15    (* -3438/225 sine-table-step epicycle)))
16    (* mean-motion (1+ factor))))

```

```

1  (defun hindu-rising-sign (date)
2    ;; TYPE fixed-date -> rational-amplitude
3    ;; Tabulated speed of rising of current zodiacal sign on
4    ;; date.
5    (let* ((i ; Index.
6    (quotient (hindu-tropical-longitude date)
7    (deg 30))))
8    (nth (mod i 6)
9    (list 1670/1800 1795/1800 1935/1800 1935/1800
10    1795/1800 1670/1800)))

```

(20.31)

```

1  (defun hindu-equation-of-time (date)
2    ;; TYPE fixed-date -> rational-moment
3    ;; Time from true to mean midnight of date.
4    ;; (This is a gross approximation to the correct value.)
5    (let* ((offset (hindu-sine
6    (hindu-mean-position
7    date
8    hindu-anomalistic-year)))
9    (equation-sun ; Sun's equation of center
10    ;; Arcsin is not needed since small

```

(20.32)

```

11      (* offset (angle 57 18 0)
12      (- 14/360 (/ (abs offset) 1080))))))
13      (* (/ (hindu-daily-motion date) (deg 360))
14      (/ equation-sun (deg 360))
15      hindu-sidereal-year)))

```

```

1  (defun hindu-sunrise (date)                                (20.33)
2    ;; TYPE fixed-date -> rational-moment
3    ;; Sunrise at hindu-location on date.
4    (+ date (hr 6) ; Mean sunrise.
5      (/ (- (longitude ujjain) (longitude hindu-location))
6          (deg 360)) ; Difference from longitude.
7      (- (hindu-equation-of-time date)) ; Apparent midnight.
8      (* ; Convert sidereal angle to fraction of civil day.
9        (/ 1577917828/1582237828 (deg 360))
10      (+ (hindu-ascensional-difference date hindu-location)
11        (* 1/4 (hindu-solar-sidereal-difference date))))))

```

```

1  (defun hindu-sunset (date)                                (20.34)
2    ;; TYPE fixed-date -> rational-moment
3    ;; Sunset at hindu-location on date.
4    (+ date (hr 18) ; Mean sunset.
5      (/ (- (longitude ujjain) (longitude hindu-location))
6          (deg 360)) ; Difference from longitude.
7      (- (hindu-equation-of-time date)) ; Apparent midnight.
8      (* ; Convert sidereal angle to fraction of civil day.
9        (/ 1577917828/1582237828 (deg 360))
10      (+ (- (hindu-ascensional-difference date hindu-location))
11        (* 3/4 (hindu-solar-sidereal-difference date))))))

```

```

1  (defun hindu-standard-from-sundial (tee)                  (20.35)
2    ;; TYPE rational-moment -> rational-moment
3    ;; Hindu local time of temporal moment tee.

```

```

4  (let* ((date (fixed-from-moment tee))
5         (time (time-from-moment tee))
6         (q (floor (* 4 time))) ; quarter of day
7         (a (cond ((= q 0) ; early this morning
8                   (hindu-sunset (1- date)))
9                 ((= q 3) ; this evening
10                  (hindu-sunset date))
11         (t ; daytime today
12          (hindu-sunrise date))))
13         (b (cond ((= q 0) (hindu-sunrise date))
14                 ((= q 3) (hindu-sunrise (1+ date)))
15                 (t (hindu-sunset date))))))
16    (+ a (* 2 (- b a) (- time
17                  (cond ((= q 3) (hr 18))
18                        ((= q 0) (hr -6))
19                        (t (hr 6)))))))

```

```

1  (defun hindu-fullmoon-from-fixed (date)                    (20.36)
2    ;; TYPE fixed-date -> hindu-lunar-date
3    ;; Hindu lunar date, full-moon scheme,
4    ;; equivalent to fixed date.
5    (let* ((l-date (hindu-lunar-from-fixed date))
6           (year (hindu-lunar-year l-date))
7           (month (hindu-lunar-month l-date))
8           (leap-month (hindu-lunar-leap-month l-date))
9           (day (hindu-lunar-day l-date))
10          (leap-day (hindu-lunar-leap-day l-date))
11          (m (if (>= day 16)
12                 (hindu-lunar-month
13                  (hindu-lunar-from-fixed (+ date 20)))
14                 month)))
15      (hindu-lunar-date year m leap-month day leap-day)))

```

```

1  (defun fixed-from-hindu-fullmoon (l-date)                  (20.37)
2    ;; TYPE hindu-lunar-date -> fixed-date
3    ;; Fixed date equivalent to Hindu lunar l-date

```

```

4      ;; in full-moon scheme.
5      (let* ((year (hindu-lunar-year l-date))
6             (month (hindu-lunar-month l-date))
7             (leap-month (hindu-lunar-leap-month l-date))
8             (day (hindu-lunar-day l-date))
9             (leap-day (hindu-lunar-leap-day l-date))
10            (m (cond ((or leap-month (<= day 15))
11                     month)
12                    ((hindu-expunged? year (amod (1- month) 12))
13                     (amod (- month 2) 12))
14                     (t (amod (1- month) 12))))))
15      (fixed-from-hindu-lunar
16      (hindu-lunar-date year m leap-month day leap-day)))

```

```

1      (defun hindu-expunged? (l-year l-month)
2      ;; TYPE (hindu-lunar-year hindu-lunar-month) ->
3      ;; TYPE boolean
4      ;; True of Hindu lunar month l-month in l-year
5      ;; is expunged.
6      (/= l-month
7         (hindu-lunar-month
8          (hindu-lunar-from-fixed
9           (fixed-from-hindu-lunar
10            (list l-year l-month false 15 false))))))

```

(20.38)

```

1      (defun alt-hindu-sunrise (date)
2      ;; TYPE fixed-date -> rational-moment
3      ;; Astronomical sunrise at Hindu location on date,
4      ;; per Lahiri,
5      ;; rounded to nearest minute, as a rational number.
6      (let* ((rise (dawn date hindu-location (angle 0 47 0)))
7             (* 1/24 1/60 (round (* rise 24 60)))))

```

(20.39)

```

1      (defun ayanamsha (tee)
2      ;; TYPE moment -> angle
3      ;; Difference between tropical and sidereal solar longitude.
4      (- (solar-longitude tee)
5         (sidereal-solar-longitude tee)))

```

(20.40)

```

1      (defconstant sidereal-start
2      ;; TYPE angle
3      (precession (universal-from-local
4                   (mesha-samkranti (ce 285))
5                   hindu-location)))

```

(20.41)

```

1      (defun astro-hindu-sunset (date)
2      ;; TYPE fixed-date -> moment
3      ;; Geometrical sunset at Hindu location on date.
4      (dusk date hindu-location (deg 0)))

```

(20.42)

```

1      (defun sidereal-zodiac (tee)
2      ;; TYPE moment -> hindu-solar-month
3      ;; Sidereal zodiacal sign of the sun, as integer in range
4      ;; 1..12, at moment tee.
5      (1+ (quotient (sidereal-solar-longitude tee) (deg 30))))

```

(20.43)

```

1      (defun astro-hindu-calendar-year (tee)
2      ;; TYPE moment -> hindu-solar-year
3      ;; Astronomical Hindu solar year KY at given moment tee.
4      (round (- (/ (- tee hindu-epoch)
5                   mean-sidereal-year)
6              (/ (sidereal-solar-longitude tee)
7                 (deg 360)))))

```

(20.44)

```

1  (defun astro-hindu-solar-from-fixed (date) (20.45) 16 month))))
2  ;; TYPE fixed-date -> hindu-solar-date 17 (+ start day -1)))
3  ;; Astronomical Hindu (Tamil) solar date equivalent to
4  ;; fixed date.
5  (let* ((critical ; Sunrise on Hindu date. 1 (defun astro-lunar-day-from-moment (tee) (20.47)
6  (astro-hindu-sunset date)) 2 ;; TYPE moment -> hindu-lunar-day
7  (month (sidereal-zodiac critical)) 3 ;; Phase of moon (tithi) at moment tee, as an integer in
8  (year (- (astro-hindu-calendar-year critical) 4 ;; the range 1..30.
9  hindu-solar-era)) 5 (1+ (quotient (lunar-phase tee) (deg 12))))
10 (approx ; 3 days before start of mean month.
11 (- date 3
12 (mod (floor (sidereal-solar-longitude critical))
13 (deg 30))))
14 (start ; Search forward for beginning...
15 (next i approx ; ... of month.
16 (= (sidereal-zodiac (astro-hindu-sunset i))
17 month)))
18 (day (- date start -1)))
19 (hindu-solar-date year month day)))

1  (defun fixed-from-astro-hindu-solar (s-date) (20.46)
2  ;; TYPE hindu-solar-date -> fixed-date
3  ;; Fixed date corresponding to Astronomical
4  ;; Hindu solar date (Tamil rule; Saka era).
5  (let* ((month (standard-month s-date))
6  (day (standard-day s-date))
7  (year (standard-year s-date))
8  (approx ; 3 days before start of mean month.
9  (+ hindu-epoch -3
10 (floor (* (+ (+ year hindu-solar-era)
11 (/ (1- month) 12))
12 mean-sidereal-year))))
13 (start ; Search forward for beginning...
14 (next i approx ; ... of month.
15 (= (sidereal-zodiac (astro-hindu-sunset i))
16 month))))
17 (last-new-moon
18 (new-moon-before critical))
19 (next-new-moon
20 (new-moon-at-or-after critical))
21 (solar-month ; Solar month name.
22 (sidereal-zodiac last-new-moon))
23 (leap-month ; If begins and ends in same sign.
24 (= solar-month (sidereal-zodiac next-new-moon)))
25 (month ; Month of lunar year.
26 (amod (1+ solar-month) 12))
27 (year ; Solar year at end of month.
28 (- (astro-hindu-calendar-year
29 (if (<= month 2) ; date might precede solar
30 ; new year.
31 (+ date 180)

```

```

26         date))
27         hindu-lunar-era)))
28 (hindu-lunar-date year month leap-month day leap-day)))

```

```

1  (defun fixed-from-astro-hindu-lunar (l-date)          (20.49)
2    ;; TYPE hindu-lunar-date -> fixed-date
3    ;; Fixed date corresponding to Hindu lunar date l-date.
4    (let* ((year (hindu-lunar-year l-date))
5           (month (hindu-lunar-month l-date))
6           (leap-month (hindu-lunar-leap-month l-date))
7           (day (hindu-lunar-day l-date))
8           (leap-day (hindu-lunar-leap-day l-date))
9           (approx
10            (+ hindu-epoch
11              (* mean-sidereal-year
12                (+ year hindu-lunar-era
13                  (/ (1- month) 12))))))
14      (s (floor
15         (- approx
16            (* hindu-sidereal-year
17              (mod3 (- (/ (sidereal-solar-longitude approx)
18                          (deg 360))
19                        (/ (1- month) 12))
20                        -1/2 1/2))))))
21        (k (astro-lunar-day-from-moment (+ s (hr 6))))
22        (est
23         (- s (- day)
24            (cond
25              ((< 3 k 27) ; Not borderline case.
26               k)
27              ((let* ((mid ; Middle of preceding solar month.
28                      (astro-hindu-lunar-from-fixed
29                       (- s 15))))
30               (or ; In month starting near s.
31                  (/= (hindu-lunar-month mid) month)

```

```

32         (and (hindu-lunar-leap-month mid)
33              (not leap-month))))
34         (mod3 k -15 15))
35         (t ; In preceding month.
36          (mod3 k 15 45))))))
37         (tau ; Refined estimate.
38          (- est (mod3 (- (astro-lunar-day-from-moment
39                          (+ est (hr 6)))
40                          day)
41                -15 15)))
42         (date (next d (1- tau)
43                  (member (astro-lunar-day-from-moment
44                          (alt-hindu-sunrise d))
45                          (list day (amod (1+ day) 30))))))
46         (if leap-day (1+ date) date)))

```

```

1  (defun hindu-solar-longitude-at-or-after (lambda tee)      (20.50)
2    ;; TYPE (season moment) -> moment
3    ;; Moment of the first time at or after tee
4    ;; when Hindu solar longitude will be lambda degrees.
5    (let* ((tau ; Estimate (within 5 days).
6           (+ tee
7              (* hindu-sidereal-year (/ 1 (deg 360))
8                (mod (- lambda (hindu-solar-longitude tee))
9                      360))))
10          (a (max tee (- tau 5))) ; At or after tee.
11          (b (+ tau 5)))
12      (invert-angular hindu-solar-longitude lambda
13                      (interval-closed a b)))

```

```

1  (defun mesha-samkranti (g-year)          (20.51)
2    ;; TYPE gregorian-year -> rational-moment
3    ;; Fixed moment of Mesha samkranti (Vernal equinox)
4    ;; in Gregorian g-year.

```

```

5      (let* ((jan1 (gregorian-new-year g-year)))
6          (hindu-solar-longitude-at-or-after (deg 0) jan1)))

```

```

1  (defun hindu-lunar-day-at-or-after (k tee)                (20.52)
2      ;; TYPE (rational rational-moment) -> rational-moment
3      ;; Time lunar-day (tithi) number k begins at or after
4      ;; moment tee. k can be fractional (for karanas).
5      (let* ((phase ; Degrees corresponding to k.
6              (* (1- k) (deg 12)))
7              (tau ; Mean occurrence of lunar-day.
8                  (+ tee (* (/ 1 (deg 360))
9                             (mod (- phase (hindu-lunar-phase tee))
10                                 360)
11                                 hindu-synodic-month)))
12              (a (max tee (- tau 2)))
13              (b (+ tau 2)))
14          (invert-angular hindu-lunar-phase phase
15                          (interval-closed a b)))

```

```

1  (defun hindu-lunar-new-year (g-year)                    (20.53)
2      ;; TYPE gregorian-year -> fixed-date
3      ;; Fixed date of Hindu lunisolar new year in Gregorian
4      ;; g-year.
5      (let* ((jan1 (gregorian-new-year g-year))
6              (mina ; Fixed moment of solar longitude 330.
7                  (hindu-solar-longitude-at-or-after (deg 330) jan1))
8              (new-moon ; Next new moon.
9                  (hindu-lunar-day-at-or-after 1 mina))
10             (h-day (floor new-moon))
11             (critical ; Sunrise that day.
12                 (hindu-sunrise h-day)))
13          (+ h-day
14             ;; Next day if new moon after sunrise,
15             ;; unless lunar day ends before next sunrise.

```

```

16         (if (or (< new-moon critical)
17                 (= (hindu-lunar-day-from-moment
18                     (hindu-sunrise (1+ h-day))) 2))
19             0 1)))

```

```

1  (defun hindu-lunar-on-or-before? (l-date1 l-date2)      (20.54)
2      ;; TYPE (hindu-lunar-date hindu-lunar-date) -> boolean
3      ;; True if Hindu lunar date l-date1 is on or before
4      ;; Hindu lunar date l-date2.
5      (let* ((month1 (hindu-lunar-month l-date1))
6              (month2 (hindu-lunar-month l-date2))
7              (leap1 (hindu-lunar-leap-month l-date1))
8              (leap2 (hindu-lunar-leap-month l-date2))
9              (day1 (hindu-lunar-day l-date1))
10             (day2 (hindu-lunar-day l-date2))
11             (leap-day1 (hindu-lunar-leap-day l-date1))
12             (leap-day2 (hindu-lunar-leap-day l-date2))
13             (year1 (hindu-lunar-year l-date1))
14             (year2 (hindu-lunar-year l-date2)))
15          (or (< year1 year2)
16              (and (= year1 year2)
17                  (or (< month1 month2)
18                      (and (= month1 month2)
19                          (or (and leap1 (not leap2))
20                              (and (equal leap1 leap2)
21                                  (or (< day1 day2)
22                                      (and (= day1 day2)
23                                          (or (not leap-day1)
24                                              leap-day2))))))
25                      )))))

```

```

1  (defun hindu-date-occur (l-year l-month l-day)          (20.55)
2      ;; TYPE (hindu-lunar-year hindu-lunar-month
3      ;; TYPE hindu-lunar-day) -> fixed-date
4      ;; Fixed date of occurrence of Hindu lunar l-month,
5      ;; l-day in Hindu lunar year l-year, taking leap and

```

```

6  ;; expunged days into account. When the month is
7  ;; expunged, then the following month is used.
8  (let* ((lunar (hindu-lunar-date 1-year 1-month false
9             1-day false))
10         (try (fixed-from-hindu-lunar lunar))
11         (mid (hindu-lunar-from-fixed
12              (if (> 1-day 15) (- try 5) try))))
13    (expunged? (/= 1-month (hindu-lunar-month mid)))
14    (1-date ; day in next month
15     (hindu-lunar-date (hindu-lunar-year mid)
16                       (hindu-lunar-month mid)
17                       (hindu-lunar-leap-month mid)
18                       1-day false)))
19    (cond (expunged?
20          (1- (next d try
21                (not
22                 (hindu-lunar-on-or-before?
23                  (hindu-lunar-from-fixed d) 1-date))))))
24          ( /= 1-day (hindu-lunar-day
25                      (hindu-lunar-from-fixed try)))
26          (1- try))
27          (t try)))

```

```

1  (defun hindu-lunar-holiday (1-month 1-day g-year)          (20.56)
2    ;; TYPE (hindu-lunar-month hindu-lunar-day
3    ;; TYPE gregorian-year) -> list-of-fixed-dates
4    ;; List of fixed dates of occurrences of Hindu lunar
5    ;; month, day in Gregorian year g-year.
6    (let* ((1-year (hindu-lunar-year
7                   (hindu-lunar-from-fixed
8                    (gregorian-new-year g-year))))
9           (date0 (hindu-date-occur 1-year 1-month 1-day))
10           (date1 (hindu-date-occur (1+ 1-year) 1-month 1-day)))
11      (list-range (list date0 date1)
12                  (gregorian-year-range g-year)))

```

```

1  (defun diwali (g-year)                                     (20.57)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of fixed date(s) of Diwali in Gregorian year
4    ;; g-year.
5    (hindu-lunar-holiday 8 1 g-year))

```

```

1  (defun hindu-tithi-occur (1-month tithi tee 1-year)        (20.58)
2    ;; TYPE (hindu-lunar-month rational rational
3    ;; TYPE hindu-lunar-year) -> fixed-date
4    ;; Fixed date of occurrence of Hindu lunar tithi prior
5    ;; to sundial time tee, in Hindu lunar 1-month, 1-year.
6    (let* ((approx
7            (hindu-date-occur 1-year 1-month (floor tithi)))
8            (lunar
9             (hindu-lunar-day-at-or-after tithi (- approx 2)))
10           (try (fixed-from-moment lunar))
11           (tee_h (standard-from-sundial (+ try tee) ujjain)))
12      (if (or (<= lunar tee_h)
13              (> (hindu-lunar-phase
14                  (standard-from-sundial (+ try 1 tee) ujjain))
15                (* 12 tithi))))
16          try
17          (1+ try))))

```

```

1  (defun hindu-lunar-event (1-month tithi tee g-year)        (20.59)
2    ;; TYPE (hindu-lunar-month rational rational
3    ;; TYPE gregorian-year) -> list-of-fixed-dates
4    ;; List of fixed dates of occurrences of Hindu lunar tithi
5    ;; prior to sundial time tee, in Hindu lunar 1-month,
6    ;; in Gregorian year g-year.
7    (let* ((1-year (hindu-lunar-year
8                   (hindu-lunar-from-fixed
9                    (gregorian-new-year g-year))))
9           (date0 (hindu-tithi-occur 1-month tithi tee 1-year))

```

```

11      (date1 (hindu-tithi-occur
12              1-month tithi tee (1+ 1-year))))
13      (list-range (list date0 date1)
14                  (gregorian-year-range g-year)))

1  (defun shiva (g-year)                                (20.60)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of fixed date(s) of Night of Shiva in Gregorian
4    ;; year g-year.
5    (hindu-lunar-event 11 29 (hr 24) g-year))

1  (defun rama (g-year)                                (20.61)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of fixed date(s) of Rama's Birthday in Gregorian
4    ;; year g-year.
5    (hindu-lunar-event 1 9 (hr 12) g-year))

1  (defun hindu-lunar-station (date)                    (20.62)
2    ;; TYPE fixed-date -> nakshatra
3    ;; Hindu lunar station (nakshatra) at sunrise on date.
4    (let* ((critical (hindu-sunrise date)))
5          (1+ (quotient (hindu-lunar-longitude critical)
6                        (angle 0 800 0)))))

1  (defun karana (n)                                    (20.63)
2    ;; TYPE 1-60 -> 0-10
3    ;; Number (0-10) of the name of the n-th (1-60) Hindu
4    ;; karana.
5    (cond ((= n 1) 0)
6          ((> n 57) (- n 50))
7          (t (amod (1- n) 7))))

1  (defun yoga (date)                                  (20.64)
2    ;; TYPE fixed-date -> 1-27
3    ;; Hindu yoga on date.
4    (1+ (floor (mod (/ (+ (hindu-solar-longitude date)
5                          (hindu-lunar-longitude date))
6                        (angle 0 800 0))
7                27))))

1  (defun sacred-wednesdays (g-year)                  (20.65)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of Wednesdays in Gregorian year g-year
4    ;; that are day 8 of Hindu lunar months.
5    (sacred-wednesdays-in-range
6      (gregorian-year-range g-year)))

1  (defun sacred-wednesdays-in-range (range)           (20.66)
2    ;; TYPE range -> list-of-fixed-dates
3    ;; List of Wednesdays within range of dates
4    ;; that are day 8 of Hindu lunar months.
5    (let* ((a (begin range))
6          (b (end range))
7          (wed (kday-on-or-after wednesday a))
8          (h-date (hindu-lunar-from-fixed wed)))
9      (if (in-range? wed range)
10         (append
11           (if (= (hindu-lunar-day h-date) 8)
12               (list wed)
13               nil)
14           (sacred-wednesdays-in-range
15             (interval (1+ wed) b)))
16         nil)))

```

D.21 The Tibetan Calendar

```

1 (defun tibetan-date (year month leap-month day leap-day)
2   ;; TYPE (tibetan-year tibetan-month
3   ;; TYPE tibetan-leap-month tibetan-day
4   ;; TYPE tibetan-leap-day) -> tibetan-date
5   (list year month leap-month day leap-day))

```

```

1 (defun tibetan-year (date)
2   ;; TYPE tibetan-date -> tibetan-year
3   (first date))

```

```

1 (defun tibetan-month (date)
2   ;; TYPE tibetan-date -> tibetan-month
3   (second date))

```

```

1 (defun tibetan-leap-month (date)
2   ;; TYPE tibetan-date -> tibetan-leap-month
3   (third date))

```

```

1 (defun tibetan-day (date)
2   ;; TYPE tibetan-date -> tibetan-day
3   (fourth date))

```

```

1 (defun tibetan-leap-day (date)
2   ;; TYPE tibetan-date -> tibetan-leap-day
3   (fifth date))

```

```

1 (defconstant tibetan-epoch
2   ;; TYPE fixed-date
3   (fixed-from-gregorian (gregorian-date -127 december 7)))

```

(21.1)

```

1 (defun tibetan-sun-equation (alpha)
2   ;; TYPE rational-angle -> rational
3   ;; Interpolated tabular sine of solar anomaly alpha.
4   (cond ((> alpha 6) (- (tibetan-sun-equation (- alpha 6))))
5         ((> alpha 3) (tibetan-sun-equation (- 6 alpha)))
6         ((integerp alpha)
7          (nth alpha (list (mins 0) (mins 6) (mins 10) (mins 11))))
8         (t (+ (* (mod alpha 1)
9                  (tibetan-sun-equation (ceiling alpha)))
10              (* (mod (- alpha) 1)
11                 (tibetan-sun-equation (floor alpha)))))))

```

(21.2)

```

1 (defun tibetan-moon-equation (alpha)
2   ;; TYPE rational-angle -> rational
3   ;; Interpolated tabular sine of lunar anomaly alpha.
4   (cond ((> alpha 14) (- (tibetan-moon-equation (- alpha 14))))
5         ((> alpha 7) (tibetan-moon-equation (- 14 alpha)))
6         ((integerp alpha)
7          (nth alpha
8               (list (mins 0) (mins 5) (mins 10) (mins 15)
                     (mins 19) (mins 22) (mins 24) (mins 25))))
9         (t (+ (* (mod alpha 1)
10                  (tibetan-moon-equation (ceiling alpha)))
11              (* (mod (- alpha) 1)
12                 (tibetan-moon-equation (floor alpha)))))))

```

(21.3)

```

1 (defun fixed-from-tibetan (t-date)
2   ;; TYPE tibetan-date -> fixed-date
3   ;; Fixed date corresponding to Tibetan lunar date t-date.
4   (let* ((year (tibetan-year t-date))
5          (month (tibetan-month t-date))
6          (leap-month (tibetan-leap-month t-date))
7          (day (tibetan-day t-date))
8          (leap-day (tibetan-leap-day t-date))

```

(21.4)

```

9      (months ; Lunar month count.
10      (floor (+ (* 804/65 (1- year)) (* 67/65 month)
11      (if leap-month -1 0) 64/65)))
12      (days ; Lunar day count.
13      (+ (* 30 months) day))
14      (mean ; Mean civil days since epoch.
15      (+ (* days 11135/11312) -30
16      (if leap-day 0 -1) 1071/1616))
17      (solar-anomaly
18      (mod (+ (* days 13/4824) 2117/4824) 1))
19      (lunar-anomaly
20      (mod (+ (* days 3781/105840) 2837/15120) 1))
21      (sun (- (tibtan-sun-equation (* 12 solar-anomaly))))
22      (moon (tibtan-moon-equation (* 28 lunar-anomaly)))
23      (floor (+ tibetan-epoch mean sun moon)))

```

```

1  (defun tibetan-from-fixed (date)
2    ;; TYPE fixed-date -> tibetan-date
3    ;; Tibetan lunar date corresponding to fixed date.
4    (let* ((cap-Y (+ 365 4975/18382)) ; Average Tibetan year.
5           (years (ceiling (/ (- date tibetan-epoch) cap-Y)))
6           (year0 ; Search for year.
7             (final y years
8               (>= date
9                 (fixed-from-tibetan
10                  (tibetan-date y 1 false 1 false)))))
11          (month0 ; Search for month.
12            (final m 1
13              (>= date
14                (fixed-from-tibetan
15                 (tibetan-date year0 m false 1 false)))))
16          (est ; Estimated day.
17            (- date (fixed-from-tibetan
18                    (tibetan-date year0 month0 false 1 false))))
19          (day0 ; Search for day.

```

(21.5)

```

20      (final
21      d (- est 2)
22      (>= date
23        (fixed-from-tibetan
24          (tibetan-date year0 month0 false d false)))))
25      (leap-month (> day0 30))
26      (day (amod day0 30))
27      (month (amod (cond ((> day day0) (1- month0))
28                        (leap-month (1+ month0))
29                        (t month0))
30              12))
31      (year (cond ((and (> day day0) (= month0 1))
32                  (1- year0))
33              ((and leap-month (= month0 12))
34                (1+ year0))
35              (t year0)))
36      (leap-day
37      (= date
38        (fixed-from-tibetan
39          (tibetan-date year month leap-month day true))))
40      (tibetan-date year month leap-month day leap-day)))

```

```

1  (defun tibetan-leap-month? (t-year t-month)
2    ;; TYPE (tibetan-year tibetan-month) -> boolean
3    ;; True if t-month is leap in Tibetan year t-year.
4    (= t-month
5      (tibetan-month
6        (tibetan-from-fixed
7          (fixed-from-tibetan
8            (tibetan-date t-year t-month true 2 false)))))

```

(21.6)

```

1  (defun tibetan-leap-day? (t-year t-month t-day)
2    ;; TYPE (tibetan-year tibetan-month tibetan-day) -> boolean
3    ;; True if t-day is leap in Tibetan

```

(21.7)

```

4      ;; month t-month and year t-year.
5      (or
6        (= t-day
7          (tibetan-day
8            (tibetan-from-fixed
9              (fixed-from-tibetan
10                (tibetan-date t-year t-month false t-day true))))))
11      ;; Check also in leap month if there is one.
12      (= t-day
13        (tibetan-day
14          (tibetan-from-fixed
15            (fixed-from-tibetan
16              (tibetan-date t-year t-month
17                (tibetan-leap-month? t-year t-month)
18                t-day true)))))))

```

```

1  (defun losar (t-year)
2    ;; TYPE tibetan-year -> fixed-date
3    ;; Fixed date of Tibetan New Year (Losar)
4    ;; in Tibetan year t-year.
5    (let* ((t-leap (tibetan-leap-month? t-year 1)))

```

(21.8)

```

6      (fixed-from-tibetan
7        (tibetan-date t-year 1 t-leap 1 false))))

1  (defun tibetan-new-year (g-year)                                     (21.9)
2    ;; TYPE gregorian-year -> list-of-fixed-dates
3    ;; List of fixed dates of Tibetan New Year in
4    ;; Gregorian year g-year.
5    (let* ((dec31 (gregorian-year-end g-year))
6          (t-year (tibetan-year (tibetan-from-fixed dec31))))
7      (list-range
8        (list (losar (1- t-year))
9              (losar t-year))
10       (gregorian-year-range g-year))))

```

References

- [1] N. Dershowitz and E. M. Reingold, "Modulo Intervals: A Proposed Notation," *ACM SIGACT News*, vol. 43, no. 3, pp. 60–64, 2012.
- [2] G. L. Steele, Jr., *Common LISP: The Language*, 2nd edn., Digital Press, Bedford, MA, 1990.



In octo libros De emendatione temporum Index.

A

A ^B	76.c.165.d.179.b.378.b	<i>Annas pontifex</i>	253.b
<i>Aban</i>	145.d.378.d.379.c	<i>Annius Piterbiensis notatur</i>	215.d
<i>Abysini</i>	338.c	<i>Annus</i>	7.d.294.c
<i>Achos sine Ochis</i>	214.b	<i>Annus Saturni</i>	7.d
<i>Adiaca victoria</i>	237.d	<i>Annus Lune</i>	ibid.
<i>A. D. 1111. EID.</i>	117.b	<i>Anni duo precipua genera apud veteres</i>	8.b
<i>Adar</i>	76.c.165.d.378.d.379.b	<i>Anni principium naturale & popolare</i>	25.a.b
<i>Adar pabafcht</i>	145.c.378.d.379.d	<i>Annus non ab ea die institutus qua mundus con-</i>	
<i>Adeler</i>	250.c	<i>ditus est, sed ab ea qua Sol & Luna</i>	199.a
<i>Adorare de geniculis</i>	346.b	<i>Annus defectivus, abundans, communis & e-</i>	
<i>Adrianus Imper.</i>	244.d	<i>quabilis</i>	10.a.11.d.86.c.127.a.315.c
<i>Adu</i>	316.d	<i>Annus causus & plenus</i>	9.d.315.c
<i>Aegon</i>	171.a.379.c	<i>Annus embolus meus</i>	298.c
<i>Aequatio anni Græcorum</i>	23.c	<i>Annus solaris</i>	7.d
<i>Aera quid</i>	235.d.236.a.313.b	<i>Anni solaris genera</i>	11.d
<i>Aera Hispanica</i>	234.d	<i>Annus solaris ante Casarem cognitus</i>	59.a
<i>Aera tres technica</i>	367.b	<i>Annus solaris Iudaicus</i>	167.d
<i>Aera Coptica vel martyrum</i>	245.b	<i>Annus solaris Hipparchi</i>	11.b
<i>Aequinoctium</i>	190.c	<i>Annus Lunaris</i>	9.d.70.b
<i>Aequinoctialia puncta non iisdem stellis semper</i>		<i>periodicus & simplex</i>	9.d
<i>affixa</i>	181.b	<i>Annus Lunaris aquabilis in duas partes aequales</i>	
<i>Aequinoctiorum epochæ immobilis</i>	181.a	<i>diuiditur</i>	317.d
<i>Aequinoctium autumnale observatum a Muba-</i>		<i>Anno lunari quo tempore Iudei viti caperunt</i>	
<i>mede Albateni</i>	192.d		79.a
<i>ab Hipparcho</i>	193.c	<i>Annus celestis</i>	11.d.180.c
<i>Aequinoctium tempore Niceni confusus 21. Mar-</i>		<i>eius examinatio</i>	190.b
<i>tij</i>	193.c	<i>Anni celestis hemerologium</i>	382
<i>Æthiopum lingua proxime abest a Chaldaea &</i>		<i>Annus celestis Dionysianus</i>	170.c
<i>Assyria</i>	338.d	<i>Orientalium</i>	172.a
<i>Æthiopes quando incunt ieiunia</i>	342.d	<i>Annus celestis & ἡμετέριος ἰσημερινός</i>	180.a
<i>344 d</i>		<i>duo genera</i>	ibid.c
<i>Agon Capitolinus</i>	243.c	<i>Annus aquabilis minor Græcorum</i>	15.a
<i>Abasibrueros</i>	285.a	<i>Annus Atricus 362. dierum</i>	27.c
<i>Abeli</i>	378.d	<i>eius initium incurrit in tempus brumæ</i>	25.a
<i>Alalcomenios</i>	ibid.	<i>principium popolare ab Hecatombaone, natu-</i>	
<i>Ἀλεξτροπόων</i>	303.c	<i>rale à Gamelione</i>	ibid.
<i>Alexander Imperator Asiæ salutus</i>	226.a	<i>Annis totius Græciæ modus</i>	17.b
<i>eius subtile commentum</i>	41.a.42.a	<i>Anno Græco aquabilis duæ castigationes adhibi-</i>	
<i>quo anno Darius vixit</i>	209.b	<i>te</i>	8.d
<i>eius obitus</i>	42.b.209.c	<i>Annis Romanorum Iulianus</i>	155.a. & inde
<i>Aliensis pugna</i>	225.b	<i>Annus Iulianus est fundamentum instituit au-</i>	
<i>Amaleki copie deletæ</i>	204.a.383.d	<i>ctioris</i>	14.b
<i>Amisrius & Ester, idem</i>	284.d	<i>Annus Romanorum vetus</i>	116.c.117.d
<i>Amischiir</i>	164.d.379.c	<i>ab initio duodecim mensum fuit</i>	ibid.
<i>ἡ ἀρχὴ ἡμετέρας</i>	48.c	<i>Annus Lilianns</i>	429.c

00

First page of the index to Joseph Scaliger's *De Emendatione Temporum* (Frankfort edition, 1593). (Courtesy of the University of Illinois, Urbana, IL.)