16

The Bahá’í Calendar

In the not far distant future it will be necessary that all peoples in the world agree on a common calendar. It seems, therefore, fitting that the new age of unity should have a new calendar free from the objections and associations which make each of the older calendars unacceptable to large sections of the world’s population, and it is difficult to see how any other arrangement could exceed in simplicity and convenience that proposed by the Báb.

John Ebenezer Esslemont: *Bahá’u’lláh and the New Era: An Introduction to the Bahá’í Faith* (1923)[1]

16.1 Structure

The Bahá’í (or Badī‘) calendar begins its years on the day of the vernal equinox. If the actual time of the equinox in Tehran occurs after sunset, then the year begins a day later [3]. This astronomical version of the Bahá’í calendar [4] is described in Section 16.3. Until recently, practice in the West had been to begin years on March 21 of the Gregorian calendar, regardless. This arithmetical version is described in Section 16.2. The calendar, based on cycles of 19, was established by the Bāb (1819–1850), the martyred forerunner of Bahā’u’llāh, founder of the Bahá’í faith.

As in the Hebrew and Islamic calendars, days are from sunset to sunset. Unlike those calendars, years are solar; they are composed of 19 months of 19 days each with an additional period of 4 or 5 days after the eighteenth month. Until recently, leap years in the Western version of the calendar followed the same pattern as in the Gregorian calendar. As on the Persian calendar, the week begins on Saturday; weekdays have the following names (in Arabic):

Saturday	Jalāl	جلال	(Glory)
Sunday	Jamāl	جمال	(Beauty)
Monday	Kamāl	كمال	(Perfection)
Tuesday	Fiḍāl	فضال	(Grace)
Wednesday	‘Idāl	عدال	(Justice)
Thursday	Istijlāl	استجلال	(Majesty)
Friday	Istiqlāl	استقلال	(Independence)

[1] نویسندگان لزوماً با نقطه نظرهای در عبارت موافقت ندارند.

The months are called

(1) Bahā'	بهاء	(Splendor)	19 days
(2) Jalāl	جلال	(Glory)	19 days
(3) Jamāl	جمال	(Beauty)	19 days
(4) 'Aẓamat	عظمت	(Grandeur)	19 days
(5) Nūr	نور	(Light)	19 days
(6) Raḥmat	رحمت	(Mercy)	19 days
(7) Kalimāt	كلمات	(Words)	19 days
(8) Kamāl	كمال	(Perfection)	19 days
(9) Asmā'	اسماء	(Names)	19 days
(10) 'Izzat	عزّت	(Might)	19 days
(11) Mashīyyat	مشيئت	(Will)	19 days
(12) 'Ilm	علم	(Knowledge)	19 days
(13) Qudrat	قدرت	(Power)	19 days
(14) Qawl	قَول	(Speech)	19 days
(15) Masā'il	مسائل	(Questions)	19 days
(16) Sharaf	شرف	(Honor)	19 days
(17) Sulṭān	سلطان	(Sovereignty)	19 days
(18) Mulk	مُلك	(Dominion)	19 days
Ayyām-i-Hā	ايّام ها	(Days of God)	4 {5} days
(19) 'Alā'	علاء	(Loftiness)	19 days

The leap-year variation is given in braces. The 19 days of each month have the same names as the months, except that there is no intercalary Ayyām-i-Hā.

Years are also named in a 19-year cycle, called *Vāḥid*, meaning "unity" and having a numerological value of 19 in Arabic letters:

(1) Alif	الف	(letter A)
(2) Bā'	باء	(letter B)
(3) Ab	أب	(Father)
(4) Dāl	دال	(letter D)
(5) Bāb	باب	(Gate)
(6) Vāv	واو	(letter V)
(7) Abad	ابد	(Eternity)
(8) Jād	جاد	(Generosity)
(9) Bahā'	بهاء	(Splendor)
(10) Ḥubb	حُبّ	(Love)
(11) Bahhāj	بهّاج	(Delightful)
(12) Javāb	جواب	(Answer)
(13) Aḥad	احد	(Single)
(14) Vahhāb	وهّاب	(Bountiful)

(15) Vidād	وداد	(Affection)
(16) Badī’	بدیع	(Beginning)
(17) Bahī	بهی	(Luminous)
(18) Abhā	ابهی	(Most Luminous)
(19) Vāḥid	واحد	(Unity)

There is also a 361-year major cycle, called *Kull-i-Shay* (the name has the numerological value $361 = 19^2$ in Arabic). Thus, for example, Monday, April 21, 1930 would be called “Kamāl (Monday), the day of Qudrat (the thirteenth), of the month of Jalāl, of the year Bahhāj (the eleventh), of the fifth Vāḥid, of the first Kull-i-Shay, of the Bahá’í Era.”

Accordingly, we represent a Bahá’í date by a list

$$\boxed{\textit{major} \mid \textit{cycle} \mid \textit{year} \mid \textit{month} \mid \textit{day}}$$

The first component, *major*, is an integer (positive for real Bahá’í dates); the components *cycle*, *year*, and *day*, take on integer values in the range 1 .. 19; because the intercalary period interrupts the sequence of month numbers, *month* is either an integer between 1 and 19 or else the special constant value

$$\textbf{ayyam-i-ha} \stackrel{\text{def}}{=} 0 \tag{16.1}$$

The epoch of the calendar, day 1 of year 1 B.E.,[2] is March 21, 1844 (Gregorian):

$$\textbf{bahai-epoch} \stackrel{\text{def}}{=} \textbf{fixed-from-gregorian}\left(\boxed{1844 \mid \textbf{march} \mid 21}\right) \tag{16.2}$$

which is R.D. 673222.

16.2 The Arithmetical Calendar

Mr. Frank E. Osborne read a complete Bahai calendar on which he has been working for the past four or five years. Abdul-Baha gave it his verbal sanction. It was referred to the executive board.

Star of the West, vol. 8 (1917)

The Bahá’í calendar used in the West until 2015 was based on the Gregorian calendar, and thus its functions are relatively straightforward:

$$\textbf{fixed-from-bahai}\left(\boxed{\textit{major} \mid \textit{cycle} \mid \textit{year} \mid \textit{month} \mid \textit{day}}\right) \stackrel{\text{def}}{=} \tag{16.3}$$

$$\textbf{fixed-from-gregorian}\left(\boxed{\textit{g-year} \mid \textbf{march} \mid 20}\right) + \left\{\begin{array}{ll} 342 & \textbf{if } \textit{month} = \textbf{ayyam-i-ha} \\ \left\{\begin{array}{ll} 347 & \textbf{if gregorian-leap-year?}\,(\textit{g-year}+1) \\ 346 & \textbf{otherwise} \end{array}\right\} & \\ & \textbf{if } \textit{month} = 19 \\ 19 \times (\textit{month}-1) & \textbf{otherwise} \end{array}\right\} + \textit{day}$$

[2] Bahá’í Era

where

$$\begin{aligned} \textit{g-year} = {} & 361 \times (\textit{major} - 1) + 19 \times (\textit{cycle} - 1) + \textit{year} - 1 \\ & + \textbf{gregorian-year-from-fixed}\,(\textbf{bahai-epoch}) \end{aligned}$$

We first find the corresponding Gregorian year by counting how many years (361 for each major cycle and 19 for each minor cycle) have elapsed since the epoch in 1844. Starting with the R.D. date of the last day (March 20) of the prior Bahá'í year, we add the number of days in the given month plus 19 days for each month, except that the intercalary period has only 4 or 5 days (for a total of 346 or 347 days), depending on whether February of the Gregorian calendar had a leap day or not.

The inverse function is

$$\textbf{bahai-from-fixed}\,(\textit{date}) \stackrel{\text{def}}{=} \tag{16.4}$$

$$\boxed{\textit{major} \mid \textit{cycle} \mid \textit{year} \mid \textit{month} \mid \textit{day}}$$

where

$$\begin{aligned}
\textit{g-year} &= \textbf{gregorian-year-from-fixed}\,(\textit{date}) \\
\textit{start} &= \textbf{gregorian-year-from-fixed}\,(\textbf{bahai-epoch}) \\
\textit{years} &= \textit{g-year} - \textit{start} \\
& \quad - \begin{cases} 1 & \textbf{if } \textit{date} \leqslant \textbf{fixed-from-gregorian} \\ & \qquad \left(\boxed{\textit{g-year} \mid \textbf{march} \mid 20}\right) \\ 0 & \textbf{otherwise} \end{cases} \\
\textit{major} &= \left\lfloor \frac{\textit{years}}{361} \right\rfloor + 1 \\
\textit{cycle} &= \left\lfloor \frac{1}{19} \times (\textit{years} \bmod 361) \right\rfloor + 1 \\
\textit{year} &= (\textit{years} \bmod 19) + 1 \\
\textit{days} &= \textit{date} - \textbf{fixed-from-bahai} \\
& \qquad \left(\boxed{\textit{major} \mid \textit{cycle} \mid \textit{year} \mid 1 \mid 1}\right) \\
\textit{month} &= \begin{cases} 19 & \textbf{if } \textit{date} \geqslant \textbf{fixed-from-bahai} \\ & \qquad \left(\boxed{\textit{major} \mid \textit{cycle} \mid \textit{year} \mid 19 \mid 1}\right) \\ \textbf{ayyam-i-ha} & \\ & \textbf{if } \textit{date} \geqslant \textbf{fixed-from-bahai} \\ & \qquad \left(\boxed{\textit{major} \mid \textit{cycle} \mid \textit{year} \mid \textbf{ayyam-i-ha} \mid 1}\right) \\ \left\lfloor \dfrac{\textit{days}}{19} \right\rfloor + 1 & \\ & \textbf{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned} day &= date + 1 \\ &\quad - \textbf{fixed-from-bahai}\left(\ \boxed{major \mid cycle \mid year \mid month \mid 1}\ \right) \end{aligned}$$

Here we compute the number of years that have elapsed since the start of the Bahá'í calendar by looking at the Gregorian year number, considering whether the date is before or after Bahá'í New Year, and then using the result to get the number of elapsed major and minor cycles and years within the cycle. Division of the remaining days by 19, the length of a month, gives the month number, but again special consideration must be given for the intercalary period and for the last month of the Bahá'í year.

16.3 The Astronomical Calendar

The chief element of the day after to-morrow in the political calendar will be All Europe as One. There can be no doubt on this point. Unhappily, however, no European nation seems yet to have realized the fact.

German contributor to *Revue de Genéve*, quoted in *The Literary Digest*, vol. 75 (1922)

The Bahá'í year was intended [3] to begin at the sunset preceding the vernal equinox, which is frequently a day before or after March 21. The location at which sunset occurs for this purpose had been undetermined for some time, as explained in the following explanatory letter [2] written in 1974:

> Until the Universal House of Justice decides upon the spot on which the calculations for establishing the date of Naw-Rúz each year are to be based it is not possible to state exactly the correspondence between Bahá'í dates and Gregorian dates for any year. Therefore for the present the believers in the West commemorate Bahá'í events on their traditional Gregorian anniversaries. Once the necessary legislation to determine Naw-Rúz has been made, the correspondence between Bahá'í and Gregorian dates will vary from year to year depending upon whether the Spring Equinox falls on the 20th, 21st or 22nd of March. In fact in Persia the friends have been, over the years, following the Spring Equinox as observed in Tehran, to determine Naw-Rúz, and the National Spiritual Assembly has to issue every year a Bahá'í calendar for the guidance of the friends. The Universal House of Justice feels that this is not a matter of urgency and, in the meantime, is having research conducted into such questions.

Thus, the version of the Bahá'í calendar employed in the Near East (which included, besides Iran, also Israel, Persian Gulf countries, and the Arabian Peninsula) used Tehran for determining the time of sunset on the day of the equinox, which in turn fixes the first day of the year. In 2014, the decision was taken to use Tehran as the determining location the world over [4]:

> "The Festival of Naw-Rúz falleth on the day that the sun entereth the sign of Aries," Bahá'u'lláh explains in His Most Holy Book, "even should this

> occur no more than one minute before sunset." However, details have, until now, been left undefined. We have decided that Ṭihrán, the birthplace of the Abhá Beauty, will be the spot on the earth that will serve as the standard for determining, by means of astronomical computations from reliable sources, the moment of the vernal equinox in the northern hemisphere and thereby the day of Naw-Rúz for the Bahá'í world.

This change took effect with the year that began on March 21, 2015.

For fixing the time of sunset in Tehran, these coordinates are used:[3]

$$\textbf{bahai-location} \stackrel{\text{def}}{=} \tag{16.5}$$

35.696111°	51.423056°	0 m	$3\frac{1}{2}^{\text{h}}$

The determination of the U.T. moment of sunset on any specified day is straightforward:

$$\textbf{bahai-sunset}\,(\textit{date}) \stackrel{\text{def}}{=} \tag{16.6}$$

$$\textbf{universal-from-standard}\\ (\textbf{sunset}\,(\textit{date}, \textbf{bahai-location}), \textbf{bahai-location})$$

The first day of the year on the new, astronomical, Bahá'í calendar is the day on which the vernal equinox occurs before sunset. To implement the astronomical form of the calendar, we imitate the method used for the astronomical Persian calendar in Section 15.2. The date of the new year is computed using formula (14.43), analogously to what was done for the Persian calendar (page 259), by beginning shortly before the equinox and searching for the sunset when the longitude of the sun first switches from large (close to 360°) to small (less than 2°):

$$\textbf{astro-bahai-new-year-on-or-before}\,(\textit{date}) \stackrel{\text{def}}{=} \tag{16.7}$$

$$\underset{\textit{day} \geqslant \lfloor \textit{approx} \rfloor - 1}{\textbf{MIN}} \left\{ \textbf{solar-longitude}\,(\textbf{bahai-sunset}\,(\textit{day})) \leqslant \textbf{spring} + 2° \right\}$$

where

$$\textit{approx} = \textbf{estimate-prior-solar-longitude}\,(\textbf{spring}, \textbf{bahai-sunset}\,(\textit{date}))$$

Because of the unequal distribution of leap years on the Gregorian calendar, the equinox will be as early as 5:27 p.m. in Tehran on March 19 in 2096, which is before sunset, and it was as late as 10:41 p.m. on March 21 in 1903, long after sunset. By the new rule, the year would begin on March 19 in the former case and March 22 in the latter.

[3] The elevation of Tehran (1180 m) is not taken into account in the sunset calculation, because the mountains to its west are at about the same height, so apparent sunset occurs at approximately the same time as astronomical sunset at zero elevation [1].

To convert a Bahá'í date on the new calendar into a fixed date, we take the R.D. date of the Bahá'í New Year and add 19 days for each full month plus the number of elapsed days in the current month. The intercalary days and last month of the year must be treated as exceptions: days in Ayyām-i-Hā are preceded by 18 full months (that is, 342 days); because the length of that period differs in ordinary and leap years, for dates in the last month, we count backwards from the following New Year. In the following function, we multiply the number of years since the epoch by the mean tropical year length, plus or minus half a year, and then use **astro-bahai-new-year-on-or-before** to get the R.D. date of the subsequent or prior Bahá'í New Year:

$$\textbf{fixed-from-astro-bahai}\left(\begin{array}{|c|c|c|c|c|}\hline \textit{major} & \textit{cycle} & \textit{year} & \textit{month} & \textit{day} \\ \hline\end{array}\right) \stackrel{\text{def}}{=} \tag{16.8}$$

$$\begin{cases} \textbf{astro-bahai-new-year-on-or-before} \\ \quad \left(\textbf{bahai-epoch} + \left\lfloor \textbf{mean-tropical-year} \times \left(\textit{years} + \frac{1}{2}\right)\right\rfloor\right) \\ - 20 + \textit{day} \\ \quad \textbf{if } \textit{month} = 19 \\ \textbf{astro-bahai-new-year-on-or-before} \\ \quad \left(\textbf{bahai-epoch} + \left\lfloor \textbf{mean-tropical-year} \times \left(\textit{years} - \frac{1}{2}\right)\right\rfloor\right) \\ + 341 + \textit{day} \\ \quad \textbf{if } \textit{month} = \textbf{ayyam-i-ha} \\ \textbf{astro-bahai-new-year-on-or-before} \\ \quad \left(\textbf{bahai-epoch} + \left\lfloor \textbf{mean-tropical-year} \times \left(\textit{years} - \frac{1}{2}\right)\right\rfloor\right) \\ + (\textit{month} - 1) \times 19 + \textit{day} - 1 \\ \quad \textbf{otherwise} \end{cases}$$

where

$$\textit{years} = 361 \times (\textit{major} - 1) + 19 \times (\textit{cycle} - 1) + \textit{year}$$

The inverse function is

$$\textbf{astro-bahai-from-fixed}\,(\textit{date}) \stackrel{\text{def}}{=} \tag{16.9}$$

$$\begin{array}{|c|c|c|c|c|}\hline \textit{major} & \textit{cycle} & \textit{year} & \textit{month} & \textit{day} \\ \hline\end{array}$$

where

$$\textit{new-year} = \textbf{astro-bahai-new-year-on-or-before}\,(\textit{date})$$

$$\textit{years} \quad = \text{round}\left(\frac{\textit{new-year} - \textbf{bahai-epoch}}{\textbf{mean-tropical-year}}\right)$$

$$\begin{aligned}
major &= \left\lfloor \frac{years}{361} \right\rfloor + 1\\
cycle &= \left\lfloor \frac{1}{19} \times (years \bmod 361) \right\rfloor + 1\\
year &= (years \bmod 19) + 1\\
days &= date - new\text{-}year\\
month &= \begin{cases} 19 & \textbf{if } date \\ & \quad \geqslant \textbf{fixed-from-astro-bahai} \\ & \qquad \left(\boxed{major \mid cycle \mid year \mid 19 \mid 1} \right) \\ \textbf{ayyam-i-ha} & \\ & \textbf{if } date \\ & \quad \geqslant \textbf{fixed-from-astro-bahai} \\ & \qquad \left(\boxed{major \mid cycle \mid year \mid \textbf{ayyam-i-ha} \mid 1} \right) \\ \left\lfloor \dfrac{days}{19} \right\rfloor + 1 & \\ & \textbf{otherwise} \end{cases}\\
day &= date + 1\\
&\quad - \textbf{fixed-from-astro-bahai}\\
&\qquad \left(\boxed{major \mid cycle \mid year \mid month \mid 1} \right)
\end{aligned}$$

Here we compute the number of years that have elapsed since the start of the Bahá'í calendar by dividing the numbers of days since the epoch by the mean tropical year length and then using the result to get the number of elapsed major and minor cycles and years within the cycle. Division of the remaining days by 19 (the length of a Bahá'í month) gives the month number but, again, consideration must be given to the intercalary days and for the last month of the Bahá'í year.[4]

[4] The published tables of the ad hoc calendar committee at the Bahá'í World Centre for the years 172–221 B.E. (2015–2064 C.E.) were prepared "using data provided by Her Majesty's Nautical Almanac Office in the United Kingdom" and are available at `wilmetteinstitute.org/wp-content/uploads/2014/11/Bahai-Dates-172-to-221-B-E-_UK-December-2014.pdf`. There is almost complete correspondence between the dates calculated with our functions and those in the table. The only divergence is for 2026, for which the table has New Year occurring on March 21, and our calculations place it on the previous day. This is, however, a very close call, since both sunset and the equinox will occur on March 20 between 6:15 and 6:16 p.m. local standard time in Tehran. On account of the very close proximity of the two events, the decision was made to set Bahá'í New Year to be March 21 [1].

16.4 Holidays and Observances

In all, there are 58 excusable days of observance for various religions on the state's academic calendar – which requires schools to open 180 days a year. It should be noted that in many schools, Christmas and Hanukkah are losing ground to Birth of the Bab Day (Baha'i), and the Rama Navami (Hindu) and Eid El Fitr (Islamic) holy days.

Lisa Suhay: "Want the Day Off? Get Some Religion,"
The New York Times (September 19, 1999)

When the Bahá'í calendar used in the West was synchronized with the Gregorian, holidays were a trivial matter. Bahá'í New Year was always celebrated on March 21, the assumed date of the spring equinox. It is called the Feast of Naw-Rūz, like the Persian New Year, which also celebrates the vernal equinox (see Chapter 15). The computation is trivial:

$$\textbf{bahai-new-year}\,(\textit{g-year}) \stackrel{\text{def}}{=} \tag{16.10}$$
$$\textbf{fixed-from-gregorian}\left(\boxed{\textit{g-year} \mid \textbf{march} \mid 21}\right)$$

The only holiday that was not aligned with the Gregorian calendar was Ayyām-i-Hā 4, which fell on March 1 in ordinary years, but on February 29 in leap years.

The other major holidays are the Birth of the Bāb (which was celebrated on ʻIlm 5 = October 20), the Birth of Bahā'u'llāh (which was celebrated on Qudrat 9 = November 12), the Feast of Riḍvān (Jalāl 13 = April 21 with the old Western version), Riḍvān 9 (Jamāl 2 = April 29), Riḍvān 12 (Jāmal 5 = May 2), the Declaration of the Bāb (ʻAẓamat 8 = May 24), the Ascension of Bahā'u'llāh (ʻAẓamat 13 = May 29), and the Martyrdom of the Bāb (Raḥmat 17 = July 10). Two other obligatory observances are the Day of the Covenant (Qawl 4 = November 26) and the Ascension of ʻAbdu'l-Bahā (Qawl 6 = November 28). There are additional days of significance, including the first day of each month (known as the Nineteen Day Feast) and the whole last month (comprising fast days).

With the new calendar, which depends on the actual time at which the equinox occurs, Bahá'í Naw-Rūz, on Bahá 1, coincides with Persian Nowruz unless the equinox occurs between noon and sunset in Tehran. A straightforward way to determine the date of Bahá'í Naw-Rūz is as follows:

$$\textbf{naw-ruz}\,(\textit{g-year}) \stackrel{\text{def}}{=} \tag{16.11}$$
$$\textbf{astro-bahai-new-year-on-or-before}\,(\textbf{gregorian-new-year}\,(\textit{g-year}+1))$$

Determining the date of holidays, apart from Birth of the Bāb and the Birth of Bahā'u'llāh, on the new astronomical calendar (as previously for the Eastern version) is simply a matter of counting a fixed number of days from Naw-Rūz, or before Naw-Rūz in the case of the month of ʻAlā'. For example, we have

$$\textbf{feast-of-ridvan}\,(\textit{g-year}) \stackrel{\text{def}}{=} \textbf{naw-ruz}\,(\textit{g-year}) + 31 \tag{16.12}$$

The other major holidays on the Bahá'í calendar are also observed on Bahá'í dates (given above), except for four that had been linked in the East to the Islamic calendar (Chapter 7) instead: Declaration of the Bāb (Islamic date Jamādā I 5), Martyrdom of the Bāb (Shʻabān 28), Birth of the Bāb (Muḥarram 1), and the Birth of Bahā'u'llāh (Muḥarram 2). (At the Bahá'í World Centre in Israel, these four had been observed on their Islamic dates whereas the other holidays had been observed on their Gregorian dates.) Following the recent decision Bahá'í dates are to be used for the first two of these four holidays, Declaration of the Bāb (on 'Aẓamat 8) and Martyrdom of the Bāb (Raḥmat 17), while astronomical lunisolar dates are to be used everywhere for the other two: The rule is that the Birth of the Bāb and the Birth of Bahā'u'llāh are observed on the first and second day, respectively, of the eighth lunisolar month, counting new moons from sunset at the end of Naw-Rūz. Using **new-moon-at-or-after** (page 231) for this purpose, we have:

$$\textbf{birth-of-the-bab}\,(\mathit{g\text{-}year}) \stackrel{\text{def}}{=} \begin{cases} \mathit{day}+1 & \textbf{if } m_8 < \mathit{set}_8 \\ \mathit{day}+2 & \textbf{otherwise} \end{cases} \tag{16.13}$$

where

$$\begin{aligned} \mathit{ny} &= \textbf{naw-ruz}\,(\mathit{g\text{-}year}) \\ \mathit{set}_1 &= \textbf{bahai-sunset}\,(\mathit{ny}) \\ m_1 &= \textbf{new-moon-at-or-after}\,(\mathit{set}_1) \\ m_8 &= \textbf{new-moon-at-or-after}\,(m_1+190) \\ \mathit{day} &= \textbf{fixed-from-moment}\,(m_8) \\ \mathit{set}_8 &= \textbf{bahai-sunset}\,(\mathit{day}) \end{aligned}$$

and m_8 is the moment of the eighth new moon of the year. If new moon is before sunset, then the eighth month begins at sunset; if the new moon is after sunset, the month begins one day later.

References

[1] Email communication from the Bahá'í World Centre, July 22, 2015.

[2] Letter written on behalf of the Universal House of Justice to the National Spiritual Assembly of the Bahá'í of the United States, October 30, 1974.

[3] Universal House of Justice, *The Bahá'í World: An International Record*, vol. xviii, Bahá'í World Center, Haifa, pp. 598–601, 1986.

[4] Universal House of Justice, "Regarding the implementation of the Badí' calendar," July 10, 2014. Available at `universalhouseofjustice.bahai.org/activities-bahai-community/20140710_001`.

Print of the French Revolutionary calendar month of Vendémiaire by Laurent Guyot, after Jean-Jacques Lagrenée, the younger, Paris. (Courtesy of Bibliothèque Nationale de France, Paris.)

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`, ..., `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, `<=` ($\leqslant$) 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 `=`, `/=` (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 ... en)` the expression

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

is used. The function `nth`, used as `(nth i l)`, extracts the ith 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,

```
(defconstant sunday                                   (1.53)
  ;; TYPE day-of-week
  ;; Residue class for Sunday.
  0)
```

(1.54)

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

(1.55)

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

(1.56)

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

(1.57)

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

(1.58)

```
(defconstant friday
  ;; TYPE day-of-week
  ;; Residue class for Friday.
  5)
```

(1.59)

```
(defconstant saturday
  ;; TYPE day-of-week
  ;; Residue class for Saturday.
  6)
```

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 prefered 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

```
(defconstant false
  ;; TYPE boolean
  ;; Constant representing false.
  nil)
```

For “true,” we define

```
(defconstant true
  ;; TYPE boolean
  ;; Constant representing true.
  t)
```

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

(1.97)

```
(defconstant bogus
  ;; TYPE string
  ;; Used to denote nonexistent dates.
  "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:

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

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

(1.60)

```
(defun day-of-week-from-fixed (date)
  ;; TYPE fixed-date -> day-of-week
  ;; The residue class of the day of the week of date.
  (mod (- date (rd 0) sunday) 7))
```

and we implement julian day calculations by writing

(1.3)

```
(defconstant jd-epoch
  ;; TYPE moment
  ;; Fixed time of start of the julian day number.
  (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

(1.1)

```
1 (defun rd (tee)
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.

(1.4)

```
(defun moment-from-jd (jd)
  ;; TYPE julian-day-number -> moment
  ;; Moment of julian day number jd.
  (+ jd jd-epoch))
```

(1.5)

```
(defun jd-from-moment (tee)
  ;; TYPE moment -> julian-day-number
  ;; Julian day number of moment tee.
  (- tee jd-epoch))
```

(1.6)

```
(defconstant mjd-epoch
  ;; TYPE fixed-date
  ;; Fixed time of start of the modified julian day number.
  (rd 678576))
```

(1.7)

```
(defun fixed-from-mjd (mjd)
  ;; TYPE julian-day-number -> fixed-date
  ;; Fixed date of modified julian day number mjd.
  (+ mjd mjd-epoch))
```

(1.8)

```
(defun mjd-from-fixed (date)
  ;; TYPE fixed-date -> julian-day-number
  ;; Modified julian day number of fixed date.
  (- date mjd-epoch))
```

(1.9)

```
(defconstant unix-epoch
  ;; TYPE fixed-date
  ;; Fixed date of the start of the Unix second count.
  (rd 719163))
```

(1.10)

```
(defun moment-from-unix (s)
  ;; TYPE second -> moment
  ;; Fixed date from Unix second count s
  (+ unix-epoch (/ s 24 60 60)))
```

(1.11)

```
(defun unix-from-moment (tee)
  ;; TYPE moment -> second
  ;; Unix second count from moment tee
  (* 24 60 60 (- tee unix-epoch)))
```

(1.13)

```
(defun fixed-from-jd (jd)
  ;; TYPE julian-day-number -> fixed-date
  ;; Fixed date of julian day number jd.
  (floor (moment-from-jd jd)))
```

(1.14)

```
(defun jd-from-fixed (date)
  ;; TYPE fixed-date -> julian-day-number
  ;; Julian day number of fixed date.
  (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, $\lfloor m/n \rfloor$:

```
(defun quotient (m n)
  ;; TYPE (real nonzero-real) -> integer
  ;; Whole part of m/n.
  (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

(1.29)

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

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

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

(1.24)

```
(defun mod3 (x a b)
  ;; TYPE (real real real) -> real
  ;; The value of x shifted into the range
  ;; [a..b). Returns x if a=b.
  (if (= a b)
      x
    (+ a (mod (- x a) (- b a)))))
```

This is typeset as $x \bmod [a \,..\, b)$; 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

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

that is, the expression $f(i)$ is summed for all $i = k, k+1, \ldots,$ 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:

(1.30)

```
(defmacro sum (expression index initial condition)
  ;; TYPE ((integer->real) * integer (integer->boolean))
  ;; TYPE  -> real
  ;; Sum expression for index = initial and successive
  ;; integers, as long as condition holds.
  `(loop for ,index from ,initial
         while ,condition
         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:

(1.31)

```
(defmacro prod (expression index initial condition)
  ;; TYPE ((integer->real) * integer (integer->boolean))
  ;; TYPE  -> real
  ;; Product of expression for index = initial and successive
  ;; integers, as long as condition holds.
  `(apply '*
          (loop for ,index from ,initial
                while ,condition
                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:

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

```
(defun poly (x a)
  ;; TYPE (real list-of-reals) -> real
  ;; Sum powers of x with coefficients (from order 0 up)
  ;; in list a.
  (if (equal a nil)
      0
    (+ (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):

(1.32)

```
(defmacro next (index initial condition)
  ;; TYPE (* integer (integer->boolean)) -> integer
  ;; First integer greater or equal to initial such that
  ;; condition holds.
  `(loop for ,index from ,initial
         when ,condition
         return ,index))
```

(1.33)

```
(defmacro final (index initial condition)
  ;; TYPE (* integer (integer->boolean)) -> integer
  ;; Last integer greater or equal to initial such that
  ;; condition holds.
  `(loop for ,index from ,initial
         when (not ,condition)
         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**:

(1.35)

```
(defmacro binary-search (l lo h hi x test end)
  ;; TYPE (* real * real * (real->boolean)
  ;; TYPE  ((real real)->boolean)) -> real
  ;; Bisection search for x in [lo..hi] such that
  ;; end holds.  test determines when to go left.
  (let* ((left (gensym)))
    `(do* ((,x false (/ (+ ,h ,l) 2))
           (,left false ,test)
           (,l ,lo (if ,left ,l ,x))
           (,h ,hi (if ,left ,x ,h)))
          (,end (/ (+ ,h ,l) 2)))))
```

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

Binary search is used mainly for function inversion:

(1.36)

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

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

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

```
(defun standard-year (date)
  ;; TYPE standard-date -> standard-year
  ;; Year field of date = (year month day).
  (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

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

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

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

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

(1.12)

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

(1.16)

```
(defun sign (y)
  ;; TYPE real -> {-1,0,+1}
  ;; Sign of y.
  (cond
   ((< y 0) -1)
   ((> y 0) +1)
   (t 0)))
```

(1.18)

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

(1.37)

```
(defun list-of-fixed-from-moments (ell)
  ;; TYPE list-of-moments -> list-of-fixed-dates
  ;; List of fixed dates corresponding to list ell
  ;; of moments.
  (if (equal ell nil)
      nil
    (append (list (fixed-from-moment (first ell)))
            (list-of-fixed-from-moments (rest ell)))))
```

```
(defun interval (t0 t1)
  ;; TYPE (moment moment) -> interval
  ;; Half-open interval [t0..t1).
  (list t0 t1))
```

```
(defun interval-closed (t0 t1)
  ;; TYPE (moment moment) -> interval
  ;; Closed interval [t0..t1].
  (list t0 t1))
```

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

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

(1.38)

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

(1.39)

```
(defun list-range (ell range)
  ;; TYPE (list-of-moments interval) -> list-of-moments
  ;; Those moments in list ell that occur in range.
  (if (equal ell nil)
      nil
    (let* ((r (list-range (rest ell) range)))
      (if (in-range? (first ell) range)
          (append (list (first ell)) r)
        r))))
```

(1.40)

```
(defun positions-in-range (p c cap-Delta range)
  ;; TYPE (nonegative-real positive-real
  ;; TYPE  nonegative-real interval) -> list-of-moments
  ;; List of occurrences of moment p of c-day cycle
  ;; within range.
  ;; cap-Delta is position in cycle of RD moment 0.
  (let* ((a (begin range))
         (b (end range))
         (date (mod3 (- p cap-Delta) a (+ a c))))
    (if (>= date b)
        nil
      (append (list date)
              (positions-in-range p c cap-Delta
                                  (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:

(1.41)

```
(defun from-radix (a b &optional c)
  ;; TYPE (list-of-reals list-of-rationals list-of-rationals)
  ;; TYPE  -> real
  ;; The number corresponding to a in radix notation
  ;; with base b for whole part and c for fraction.
  (/ (sum (* (nth i a)
             (prod (nth j (append b c))
                   j i (< j (+ (length b) (length c)))))
          i 0 (< i (length a)))
     (apply '* c)))
```

where `length` measures the length of a list; and

(1.42)

```
(defun to-radix (x b &optional c)
  ;; TYPE (real list-of-rationals list-of-rationals)
  ;; TYPE  -> list-of-reals
  ;; The radix notation corresponding to x
  ;; with base b for whole part and c for fraction.
  (if (null c)
      (if (null b)
          (list x)
        (append (to-radix (quotient x (nth (1- (length b)) b))
                          (butlast b) nil)
                (list (mod x (nth (1- (length b)) b)))))
    (to-radix (* x (apply '* c)) (append b c))))
```

which is implemented recursively.

(1.43)

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

(1.44)

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

(1.45)

```
(defun angle-from-degrees (alpha)
  ;; TYPE angle -> list-of-reals
  ;; List of degrees-arcminutes-arcseconds from angle alpha
  ;; in degrees.
  (let* ((dms (to-radix (abs alpha) nil (list 60 60))))
    (if (>= alpha 0)
        dms
      (list ; degrees-minutes-seconds
       (- (first dms)) (- (second dms)) (- (third dms))))))
```

D.1.3 The Egyptian and Armenian Calendars

```
(defun egyptian-date (year month day)
  ;; TYPE (egyptian-year egyptian-month egyptian-day)
  ;; TYPE  -> egyptian-date
  (list year month day))
```

(1.46)

```
(defconstant egyptian-epoch
  ;; TYPE fixed-date
  ;; Fixed date of start of the Egyptian (Nabonasser)
  ;; calendar.
  ;; JD 1448638 = February 26, 747 BCE (Julian).
  (fixed-from-jd 1448638))
```

(1.47)

```
(defun fixed-from-egyptian (e-date)
  ;; TYPE egyptian-date -> fixed-date
  ;; Fixed date of Egyptian date e-date.
  (let* ((month (standard-month e-date))
         (day (standard-day e-date))
         (year (standard-year e-date)))
    (+ 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
```

(1.48)

```
(defun alt-fixed-from-egyptian (e-date)
  ;; TYPE egyptian-date -> fixed-date
  ;; Fixed date of Egyptian date e-date.
  (+ egyptian-epoch
     (sigma ((a (list 365 30 1))
             (e-date e-date))
            (* a (1- e-date)))))
```

(1.49)

```
(defun egyptian-from-fixed (date)
  ;; TYPE fixed-date -> egyptian-date
  ;; Egyptian equivalent of fixed date.
  (let* ((days ; Elapsed days since epoch.
          (- date egyptian-epoch))
         (year ; Year since epoch.
          (1+ (quotient days 365)))
         (month; Calculate the month by division.
          (1+ (quotient (mod days 365)
                        30)))
         (day  ; Calculate the day by subtraction.
          (- days
             (* 365 (1- year))
             (* 30 (1- month))
             -1)))
    (egyptian-date year month day)))
```

```
(defun armenian-date (year month day)
  ;; TYPE (armenian-year armenian-month armenian-day)
  ;; TYPE  -> armenian-date
  (list year month day))
```

(1.50)

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

(1.51)

```
(defun fixed-from-armenian (a-date)
  ;; TYPE armenian-date -> fixed-date
  ;; Fixed date of Armenian date a-date.
  (let* ((month (standard-month a-date))
         (day (standard-day a-date))
         (year (standard-year a-date)))
    (+ armenian-epoch
       (- (fixed-from-egyptian
           (egyptian-date year month day))
          egyptian-epoch))))
```

(1.52)

```
(defun armenian-from-fixed (date)
  ;; TYPE fixed-date -> armenian-date
  ;; Armenian equivalent of fixed date.
  (egyptian-from-fixed
   (+ date (- egyptian-epoch armenian-epoch))))
```

D.1.4 Cycles of Days

(1.62)

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

(1.65)

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

(1.66)

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

(1.67)
```
(defun kday-before (k date)
  ;; TYPE (day-of-week fixed-date) -> fixed-date
  ;; Fixed date of the k-day before fixed date.
  ;; k=0 means Sunday, k=1 means Monday, and so on.
  (kday-on-or-before k (- date 1)))
```

(1.68)
```
(defun kday-after (k date)
  ;; TYPE (day-of-week fixed-date) -> fixed-date
  ;; Fixed date of the k-day after fixed date.
  ;; k=0 means Sunday, k=1 means Monday, and so on.
  (kday-on-or-before k (+ date 7)))
```

D.1.5 Akan Calendar

(1.76)
```
(defun akan-day-name (n)
  ;; TYPE integer -> akan-name
  ;; The n-th name of the Akan cycle.
  (akan-name (amod n 6)
             (amod n 7)))
```

```
(defun akan-name (prefix stem)
  ;; TYPE (akan-prefix akan-stem) -> akan-name
  (list prefix stem))
```

```
(defun akan-prefix (name)
  ;; TYPE akan-name -> akan-prefix
  (first name))
```

```
(defun akan-stem (name)
  ;; TYPE akan-name -> akan-stem
  (second name))
```

(1.77)
```
(defun akan-name-difference (a-name1 a-name2)
  ;; TYPE (akan-name akan-name) -> nonnegative-integer
  ;; Number of names from Akan name a-name1 to the
  ;; next occurrence of Akan name a-name2.
  (let* ((prefix1 (akan-prefix a-name1))
         (prefix2 (akan-prefix a-name2))
         (stem1 (akan-stem a-name1))
         (stem2 (akan-stem a-name2))
         (prefix-difference (- prefix2 prefix1))
         (stem-difference (- stem2 stem1)))
    (amod (+ prefix-difference
             (* 36 (- stem-difference
                      prefix-difference)))
          42)))
```

(1.78)
```
(defconstant akan-day-name-epoch
  ;; TYPE fixed-date
  ;; RD date of an epoch (day 0)  of Akan day cycle.
  (rd 37))
```

(1.79)
```
(defun akan-name-from-fixed (date)
  ;; TYPE fixed-date -> akan-name
  ;; Akan name for date.
  (akan-day-name (- date akan-day-name-epoch)))
```

(1.80)
```
(defun akan-day-name-on-or-before (name date)
  ;; TYPE (akan-name fixed-date) -> fixed-date
  ;; Fixed date of latest date on or before fixed date
  ;; that has Akan name.
  (mod3
   (akan-name-difference (akan-name-from-fixed 0) name)
   date (- date 42)))
```

D.2 The Gregorian Calendar

```
(defun gregorian-date (year month day)
  ;; TYPE (gregorian-year gregorian-month gregorian-day)
  ;; TYPE  -> gregorian-date
  (list year month day))
```

(2.3)

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

(2.4)

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

(2.5)

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

(2.6)

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

(2.7)

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

(2.8)

```
(defconstant may
  ;; TYPE standard-month
  ;; May on Julian/Gregorian calendar.
  5)
```

(2.9)

```
(defconstant june
  ;; TYPE standard-month
  ;; June on Julian/Gregorian calendar.
  6)
```

(2.10)

```
(defconstant july
  ;; TYPE standard-month
  ;; July on Julian/Gregorian calendar.
  7)
```

(2.11)

```
(defconstant august
  ;; TYPE standard-month
  ;; August on Julian/Gregorian calendar.
  8)
```

(2.12)

```
(defconstant september
  ;; TYPE standard-month
  ;; September on Julian/Gregorian calendar.
  9)
```

(2.13)

```
(defconstant october
  ;; TYPE standard-month
  ;; October on Julian/Gregorian calendar.
  10)
```

(2.14)

```
(defconstant november
  ;; TYPE standard-month
  ;; November on Julian/Gregorian calendar.
  11)
```

(2.15)

```
(defconstant december
  ;; TYPE standard-month
  ;; December on Julian/Gregorian calendar.
  12)
```

(2.16)

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

(2.17)

```
(defun fixed-from-gregorian (g-date)
  ;; TYPE gregorian-date -> fixed-date
  ;; Fixed date equivalent to the Gregorian date g-date.
  (let* ((month (standard-month g-date))
         (day (standard-day g-date))
         (year (standard-year g-date)))
    (+ (1- gregorian-epoch); Days before start of calendar
       (* 365 (1- year)); Ordinary days since epoch
       (quotient (1- year)
                 4); Julian leap days since epoch...
       (-          ; ...minus century years since epoch...
        (quotient (1- year) 100))
       (quotient   ; ...plus years since epoch divisible...
        (1- year) 400)  ; ...by 400.
       (quotient        ; Days in prior months this year...
        (- (* 367 month) 362); ...assuming 30-day Feb
        12)
       (if (<= month 2) ; Correct for 28- or 29-day Feb
           0
         (if (gregorian-leap-year? year)
             -1
           -2))
       day)))          ; Days so far this month.
```

(2.18)

```
(defun gregorian-new-year (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of January 1 in g-year.
  (fixed-from-gregorian
   (gregorian-date g-year january 1)))
```

(2.19)

```
(defun gregorian-year-end (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of December 31 in g-year.
  (fixed-from-gregorian
   (gregorian-date g-year december 31)))
```

(2.20)

```
(defun gregorian-year-range (g-year)
  ;; TYPE gregorian-year -> range
  ;; The range of moments in Gregorian year g-year.
  (interval (gregorian-new-year g-year)
            (gregorian-new-year (1+ g-year))))
```

(2.21)

```
(defun gregorian-year-from-fixed (date)
  ;; TYPE fixed-date -> gregorian-year
  ;; Gregorian year corresponding to the fixed date.
  (let* ((d0        ; Prior days.
          (- date gregorian-epoch))
         (n400      ; Completed 400-year cycles.
          (quotient d0 146097))
```

```
          (d1        ; Prior days not in n400.
           (mod d0 146097))
          (n100      ; 100-year cycles not in n400.
           (quotient d1 36524))
          (d2        ; Prior days not in n400 or n100.
           (mod d1 36524))
          (n4        ; 4-year cycles not in n400 or n100.
           (quotient d2 1461))
          (d3        ; Prior days not in n400, n100, or n4.
           (mod d2 1461))
          (n1        ; Years not in n400, n100, or n4.
           (quotient d3 365))
          (year (+ (* 400 n400)
                   (* 100 n100)
                   (* 4 n4)
                   n1)))
     (if (or (= n100 4) (= n1 4))
         year      ; Date is day 366 in a leap year.
       (1+ year)))); Date is ordinal day (1+ (mod d3 365))
                                       ; in (1+ year).
```

(2.23)

```
(defun gregorian-from-fixed (date)
  ;; TYPE fixed-date -> gregorian-date
  ;; Gregorian (year month day) corresponding to fixed date.
  (let* ((year (gregorian-year-from-fixed date))
         (prior-days; This year
          (- date (gregorian-new-year year)))
         (correction; To simulate a 30-day Feb
          (if (< date (fixed-from-gregorian
                       (gregorian-date year march 1)))
              0
            (if (gregorian-leap-year? year)
                1
              2)))
         (month     ; Assuming a 30-day Feb
          (quotient
           (+ (* 12 (+ prior-days correction)) 373)
           367))
         (day       ; Calculate the day by subtraction.
          (1+ (- date
                 (fixed-from-gregorian
                  (gregorian-date year month 1))))))
    (gregorian-date year month day)))
```

(2.24)

```
(defun gregorian-date-difference (g-date1 g-date2)
  ;; TYPE (gregorian-date gregorian-date) -> integer
  ;; Number of days from Gregorian date g-date1 until
  ;; g-date2.
  (- (fixed-from-gregorian g-date2)
     (fixed-from-gregorian g-date1)))
```

(2.25)

```
(defun day-number (g-date)
  ;; TYPE gregorian-date -> positive-integer
  ;; Day number in year of Gregorian date g-date.
  (gregorian-date-difference
   (gregorian-date (1- (standard-year g-date)) december 31)
   g-date))
```

(2.26)

```
(defun days-remaining (g-date)
  ;; TYPE gregorian-date -> nonnegative-integer
  ;; Days remaining in year after Gregorian date g-date.
  (gregorian-date-difference
   g-date
   (gregorian-date (standard-year g-date) december 31)))
```

(2.27)

```
(defun last-day-of-gregorian-month (g-year g-month)
  ;; TYPE (gregorian-year gregorian-month) -> gregorian-day
  ;; Last day of month g-month in Gregorian year g-year.
  (gregorian-date-difference
```

```
    (gregorian-date g-year g-month 1)
    (gregorian-date (if (= g-month 12)
                        (1+ g-year)
                      g-year)
                    (amod (1+ g-month) 12)
                    1)))
```

(2.28)

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

(2.29)

```
(defun alt-gregorian-from-fixed (date)
  ;; TYPE fixed-date -> gregorian-date
  ;; Alternative calculation of Gregorian (year month day)
  ;; corresponding to fixed date.
  (let* ((y (gregorian-year-from-fixed
             (+ (1- gregorian-epoch)
                date
                306)))
         (prior-days
          (- date (fixed-from-gregorian
                   (gregorian-date (1- y) march 1))))
         (month
          (amod (+ (quotient
                    (+ (* 5 prior-days) 2)
                    153)
                   3)
                12))
         (year (- y (quotient (+ month 9) 12)))
         (day
          (1+ (- date
                 (fixed-from-gregorian
                  (gregorian-date year month 1))))))
    (gregorian-date year month day)))
```

(2.30)

```
(defun alt-gregorian-year-from-fixed (date)
  ;; TYPE fixed-date -> gregorian-year
  ;; Gregorian year corresponding to the fixed date.
  (let* ((approx ; approximate year
          (quotient (- date gregorian-epoch -2)
                    146097/400))
         (start  ; start of next year
          (+ gregorian-epoch
             (* 365 approx)
             (sigma ((y (to-radix approx (list 4 25 4)))
                     (a (list 97 24 1 0)))
                    (* y a)))))
    (if (< date start)
        approx
      (1+ approx))))
```

(2.32)

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

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

(2.33)

```
(defun nth-kday (n k g-date)
  ;; TYPE (integer day-of-week gregorian-date) -> fixed-date
  ;; If n>0, return the n-th k-day on or after
  ;; g-date.  If n<0, return the n-th k-day on or
  ;; before g-date.  If n=0 return bogus.  A k-day of
  ;; 0 means Sunday, 1 means Monday, and so on.
  (cond ((> n 0)
         (+ (* 7 n)
            (kday-before k (fixed-from-gregorian g-date))))
        ((< n 0)
         (+ (* 7 n)
            (kday-after k (fixed-from-gregorian g-date))))
        (t bogus)))
```

(2.34)

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

(2.35)

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

(2.36)

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

(2.37)

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

(2.38)

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

(2.39)

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

(2.40)

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

```
(defun christmas (g-year)                                  (2.41)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Christmas in Gregorian year g-year.
  (fixed-from-gregorian
   (gregorian-date g-year december 25)))
```

```
(defun advent (g-year)                                     (2.42)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Advent in Gregorian year g-year
  ;; (the Sunday closest to November 30).
  (kday-nearest sunday
                (fixed-from-gregorian
                 (gregorian-date g-year november 30))))
```

```
(defun epiphany (g-year)                                   (2.43)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Epiphany in U.S. in Gregorian year
  ;; g-year (the first Sunday after January 1).
  (first-kday sunday (gregorian-date g-year january 2)))
```

```
(defun unlucky-fridays-in-range (range)                    (2.44)
  ;; TYPE range -> list-of-fixed-dates
  ;; List of Fridays within range of dates
  ;; that are day 13 of Gregorian months.
  (let* ((a (begin range))
         (b (end range))
         (fri (kday-on-or-after friday a))
         (date (gregorian-from-fixed fri)))
    (if (in-range? fri range)
        (append
         (if (= (standard-day date) 13)
             (list fri)
           nil)
         (unlucky-fridays-in-range
          (interval (1+ fri) b)))
      nil)))
```

```
(defun unlucky-fridays (g-year)                            (2.45)
  ;; TYPE gregorian-year -> list-of-fixed-dates
  ;; List of Fridays within Gregorian year g-year
  ;; that are day 13 of Gregorian months.
  (unlucky-fridays-in-range
   (gregorian-year-range g-year)))
```

D.3 The Julian Calendar

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

```
(defun bce (n)
  ;; TYPE standard-year -> julian-year
  ;; Negative value to indicate a BCE Julian year.
  (- n))
```

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

```
(defun ce (n)
  ;; TYPE standard-year -> julian-year
  ;; Positive value to indicate a CE Julian year.
  n)
```

```
(defun julian-date (year month day)
  ;; TYPE (julian-year julian-month julian-day)
  ;; TYPE  -> julian-date
  (list year month day))
```

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

(3.2)

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

(3.3)

```
(defun fixed-from-julian (j-date)
  ;; TYPE julian-date -> fixed-date
  ;; Fixed date equivalent to the Julian date j-date.
  (let* ((month (standard-month j-date))
         (day (standard-day j-date))
         (year (standard-year j-date))
         (y (if (< year 0)
                (1+ year) ; No year zero
              year)))
    (+ (1- julian-epoch)  ; Days before start of calendar
       (* 365 (1- y))     ; Ordinary days since epoch.
       (quotient (1- y) 4); Leap days since epoch...
       (quotient          ; Days in prior months this year...
        (- (* 367 month) 362); ...assuming 30-day Feb
        12)
       (if (<= month 2)   ; Correct for 28- or 29-day Feb
           0
         (if (julian-leap-year? year)
             -1
           -2))
       day)))             ; Days so far this month.
```

(3.4)

```
(defun julian-from-fixed (date)
  ;; TYPE fixed-date -> julian-date
  ;; Julian (year month day) corresponding to fixed date.
  (let* ((approx      ; Nominal year.
          (quotient (+ (* 4 (- date julian-epoch)) 1464)
                    1461))
         (year (if (<= approx 0)
                   (1- approx) ; No year 0.
                approx))
         (prior-days; This year
          (- date (fixed-from-julian
                   (julian-date year january 1))))
         (correction; To simulate a 30-day Feb
          (if (< date (fixed-from-julian
                       (julian-date year march 1)))
              0
            (if (julian-leap-year? year)
                1
              2)))
         (month     ; Assuming a 30-day Feb
          (quotient
           (+ (* 12 (+ prior-days correction)) 373)
           367))
         (day       ; Calculate the day by subtraction.
          (1+ (- date
                 (fixed-from-julian
                  (julian-date year month 1))))))
    (julian-date year month day)))
```

(3.5)

```
(defconstant kalends
  ;; TYPE roman-event
  ;; Class of Kalends.
  1)
```

(3.6)

```
(defconstant nones
  ;; TYPE roman-event
  ;; Class of Nones.
  2)
```

(3.7)

```
(defconstant ides
  ;; TYPE roman-event
  ;; Class of Ides.
  3)
```

```
(defun roman-date (year month event count leap)
  ;; TYPE (roman-year roman-month roman-event roman-count
  ;; TYPE  roman-leap) -> roman-date
  (list year month event count leap))
```

```
(defun roman-year (date)
  ;; TYPE roman-date -> roman-year
  (first date))
```

```
(defun roman-month (date)
  ;; TYPE roman-date -> roman-month
  (second date))
```

```
(defun roman-event (date)
  ;; TYPE roman-date -> roman-event
  (third date))
```

```
(defun roman-count (date)
  ;; TYPE roman-date -> roman-count
  (fourth date))
```

```
(defun roman-leap (date)
  ;; TYPE roman-date -> roman-leap
  (fifth date))
```

```
(defun ides-of-month (month)                                          (3.8)
  ;; TYPE roman-month -> ides
  ;; Date of Ides in Roman month.
  (if (member month (list march may july october))
      15
    13))
```

```
(defun nones-of-month (month)                                         (3.9)
  ;; TYPE roman-month -> nones
  ;; Date of Nones in Roman month.
  (- (ides-of-month month) 8))
```

```
(defun fixed-from-roman (r-date)                                      (3.10)
  ;; TYPE roman-date -> fixed-date
  ;; Fixed date for Roman name r-date.
  (let* ((leap (roman-leap r-date))
         (count (roman-count r-date))
         (event (roman-event r-date))
         (month (roman-month r-date))
         (year (roman-year r-date)))
    (+ (cond
        ((= event kalends)
         (fixed-from-julian (julian-date year month 1)))
        ((= event nones)
         (fixed-from-julian
          (julian-date year month (nones-of-month month))))
        ((= event ides)
         (fixed-from-julian
          (julian-date year month (ides-of-month month)))))
       (- count)
       (if (and (julian-leap-year? year)
                (= month march)
                (= event kalends)
                (>= 16 count 6))
           0 ; After Ides until leap day
         1) ; Otherwise
       (if leap
           1 ; Leap day
         0)))) ; Non-leap day
```

(3.11)

```
(defun roman-from-fixed (date)
  ;; TYPE fixed-date -> roman-date
  ;; Roman name for fixed date.
  (let* ((j-date (julian-from-fixed date))
         (month (standard-month j-date))
         (day (standard-day j-date))
         (year (standard-year j-date))
         (month-prime (amod (1+ month) 12))
         (year-prime (if (/= month-prime 1)
                         year
                       (if (/= year -1)
                           (1+ year)
                         1)))
         (kalends1 (fixed-from-roman
                    (roman-date year-prime month-prime
                                kalends 1 false))))
    (cond
     ((= day 1) (roman-date year month kalends 1 false))
     ((<= day (nones-of-month month))
      (roman-date year month nones
                  (1+ (- (nones-of-month month) day)) false))
     ((<= day (ides-of-month month))
      (roman-date year month ides
                  (1+ (- (ides-of-month month) day)) false))
     ((or (/= month february)
          (not (julian-leap-year? year)))
      ;; After the Ides, in a month that is not February of a
      ;; leap year
      (roman-date year-prime month-prime kalends
                  (1+ (- kalends1 date)) false))
     ((< day 25)
      ;; February of a leap year, before leap day
      (roman-date year march kalends (- 30 day) false))
     (true
      ;; February of a leap year, on or after leap day
      (roman-date year march kalends
                  (- 31 day) (= day 25))))))
```

(3.12)

```
(defconstant year-rome-founded
  ;; TYPE julian-year
  ;; Year on the Julian calendar of the founding of Rome.
  (bce 753))
```

(3.13)

```
(defun julian-year-from-auc (year)
  ;; TYPE auc-year -> julian-year
  ;; Julian year equivalent to AUC year
  (if (<= 1 year (- year-rome-founded))
      (+ year year-rome-founded -1)
    (+ year year-rome-founded)))
```

(3.14)

```
(defun auc-year-from-julian (year)
  ;; TYPE julian-year -> auc-year
  ;; Year AUC equivalent to Julian year
  (if (<= year-rome-founded year -1)
      (- year year-rome-founded -1)
    (- year year-rome-founded)))
```

```
(defun olympiad (cycle year)
  ;; TYPE (olympiad-cycle olympiad-year) -> olympiad
  (list cycle year))
```

```
(defun olympiad-cycle (o-date)
  ;; TYPE olympiad -> olympiad-cycle
  (first o-date))
```

```
(defun olympiad-year (o-date)
  ;; TYPE olympiad -> olympiad-year
  (second o-date))
```

(3.15)
```
(defconstant olympiad-start
  ;; TYPE julian-year
  ;; Start of the Olympiads.
  (bce 776))
```

(3.16)
```
(defun julian-year-from-olympiad (o-date)
  ;; TYPE olympiad -> julian-year
  ;; Julian year corresponding to Olympian o-date.
  (let* ((cycle (olympiad-cycle o-date))
         (year (olympiad-year o-date))
         (years (+ olympiad-start
                   (* 4 (1- cycle))
                   year -1)))
    (if (< years 0)
        years
      (1+ years))))
```

(3.17)
```
(defun olympiad-from-julian-year (j-year)
  ;; TYPE julian-year -> olympiad
  ;; Olympiad corresponding to Julian year j-year.
  (let* ((years (- j-year olympiad-start
                   (if (< j-year 0) 0 1))))
    (olympiad (1+ (quotient years 4))
              (1+ (mod years 4)))))
```

(3.18)
```
(defconstant spring
  ;; TYPE season
  ;; Longitude of sun at vernal equinox.
  (deg 0))
```

(3.19)
```
(defconstant summer
  ;; TYPE season
  ;; Longitude of sun at summer solstice.
  (deg 90))
```

(3.20)
```
(defconstant autumn
  ;; TYPE season
  ;; Longitude of sun at autumnal equinox.
  (deg 180))
```

(3.21)
```
(defconstant winter
  ;; TYPE season
  ;; Longitude of sun at winter solstice.
  (deg 270))
```

(3.22)
```
(defun cycle-in-gregorian (season g-year cap-L start)
  ;; TYPE (season gregorian-year positive-real moment)
  ;; TYPE  -> list-of-moments
  ;; Moments of season in Gregorian year g-year.
  ;; Seasonal year is cap-L days, seasons are given as
  ;; longitudes and are of equal length,
  ;; and a seasonal year started at moment start.
  (let* ((year (gregorian-year-range g-year))
         (pos (* (/ season (deg 360)) cap-L))
         (cap-Delta (- pos (mod start cap-L))))
    (positions-in-range pos cap-L cap-Delta year)))
```

(3.23)
```
(defun julian-season-in-gregorian (season g-year)
  ;; TYPE (season gregorian-year) -> list-of-moments
  ;; Moment(s) of Julian season in Gregorian year g-year.
  (let* ((cap-Y (+ 365 (hr 6)))
         (offset ; season start
          (* (/ season (deg 360)) cap-Y)))
    (cycle-in-gregorian season g-year cap-Y
                        (+ (fixed-from-julian
                            (julian-date (bce 1) march 23))
                           offset))))
```

(3.24)

```
(defun julian-in-gregorian (j-month j-day g-year)
  ;; TYPE (julian-month julian-day gregorian-year)
  ;; TYPE  -> list-of-fixed-dates
  ;; List of the fixed dates of Julian month j-month, day
  ;; j-day that occur in Gregorian year g-year.
  (let* ((jan1 (gregorian-new-year g-year))
         (y (standard-year (julian-from-fixed jan1)))
         (y-prime (if (= y -1)
                      1
                    (1+ y)))
         ;; The possible occurrences in one year are
         (date0 (fixed-from-julian
                 (julian-date y j-month j-day)))
         (date1 (fixed-from-julian
                 (julian-date y-prime j-month j-day))))
    (list-range (list date0 date1)
                (gregorian-year-range g-year))))
```

(3.25)

```
(defun eastern-orthodox-christmas (g-year)
  ;; TYPE gregorian-year -> list-of-fixed-dates
  ;; List of zero or one fixed dates of Eastern Orthodox
  ;; Christmas in Gregorian year g-year.
  (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 Ethiopic Calendars

```
(defun coptic-date (year month day)
  ;; TYPE (coptic-year coptic-month coptic-day) -> coptic-date
  (list year month day))
```

(4.1)

```
(defconstant coptic-epoch
  ;; TYPE fixed-date
  ;; Fixed date of start of the Coptic calendar.
  (fixed-from-julian (julian-date (ce 284) august 29)))
```

(4.2)

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

(4.3)

```
(defun fixed-from-coptic (c-date)
  ;; TYPE coptic-date -> fixed-date
  ;; Fixed date of Coptic date c-date.
  (let* ((month (standard-month c-date))
         (day (standard-day c-date))
         (year (standard-year c-date)))
    (+ coptic-epoch -1  ; Days before start of calendar
       (* 365 (1- year)); Ordinary days in prior years
       (quotient year 4); Leap days in prior years
       (* 30 (1- month)); Days in prior months this year
       day)))           ; Days so far this month
```

(4.4)

```
(defun coptic-from-fixed (date)
  ;; TYPE fixed-date -> coptic-date
  ;; Coptic equivalent of fixed date.
  (let* ((year ; Calculate the year by cycle-of-years formula
          (quotient (+ (* 4 (- date coptic-epoch)) 1463)
                    1461))
         (month; Calculate the month by division.
          (1+ (quotient
               (- date (fixed-from-coptic
                        (coptic-date year 1 1)))
               30)))
```

```
        (day  ; Calculate the day by subtraction.
         (- date -1
            (fixed-from-coptic
             (coptic-date year month 1)))))
    (coptic-date year month day)))
```

```
(defun ethiopic-date (year month day)
  ;; TYPE (ethiopic-year ethiopic-month ethiopic-day)
  ;; TYPE  -> ethiopic-date
  (list year month day))
```

(4.5)

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

(4.6)

```
(defun fixed-from-ethiopic (e-date)
  ;; TYPE ethiopic-date -> fixed-date
  ;; Fixed date of Ethiopic date e-date.
  (let* ((month (standard-month e-date))
         (day (standard-day e-date))
         (year (standard-year e-date)))
    (+ ethiopic-epoch
       (- (fixed-from-coptic
           (coptic-date year month day))
          coptic-epoch))))
```

(4.7)

```
(defun ethiopic-from-fixed (date)
  ;; TYPE fixed-date -> ethiopic-date
  ;; Ethiopic equivalent of fixed date.
  (coptic-from-fixed
   (+ date (- coptic-epoch ethiopic-epoch))))
```

(4.8)

```
(defun coptic-in-gregorian (c-month c-day g-year)
  ;; TYPE (coptic-month coptic-day gregorian-year)
  ;; TYPE  -> list-of-fixed-dates
  ;; List of the fixed dates of Coptic month c-month, day
  ;; c-day that occur in Gregorian year g-year.
  (let* ((jan1 (gregorian-new-year g-year))
         (y (standard-year (coptic-from-fixed jan1)))
         ;; The possible occurrences in one year are
         (date0 (fixed-from-coptic
                 (coptic-date y c-month c-day)))
         (date1 (fixed-from-coptic
                 (coptic-date (1+ y) c-month c-day))))
    (list-range (list date0 date1)
                (gregorian-year-range g-year))))
```

(4.9)

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

D.5 The ISO Calendar

```
(defun iso-date (year week day)
  ;; TYPE (iso-year iso-week iso-day) -> iso-date
  (list year week day))
```

```
(defun iso-week (date)
  ;; TYPE iso-date -> iso-week
  (second date))
```

```
(defun iso-day (date)
  ;; TYPE iso-date -> day-of-week
  (third date))
```

```
(defun iso-year (date)
  ;; TYPE iso-date -> iso-year
  (first date))
```

```
(defun fixed-from-iso (i-date)                                    (5.1)
  ;; TYPE iso-date -> fixed-date
  ;; Fixed date equivalent to ISO i-date.
  (let* ((week (iso-week i-date))
         (day (iso-day i-date))
         (year (iso-year i-date)))
    ;; Add fixed date of Sunday preceding date plus day
    ;; in week.
    (+ (nth-kday
        week sunday
        (gregorian-date (1- year) december 28)) day)))
```

```
(defun iso-from-fixed (date)                                      (5.2)
  ;; TYPE fixed-date -> iso-date
  ;; ISO (year week day) corresponding to the fixed date.
  (let* ((approx ; Year may be one too small.
          (gregorian-year-from-fixed (- date 3)))
         (year (if (>= date
                       (fixed-from-iso
                        (iso-date (1+ approx) 1 1)))
                   (1+ approx)
                 approx))
         (week (1+ (quotient
                    (- date
                       (fixed-from-iso (iso-date year 1 1)))
                    7)))
         (day (amod (- date (rd 0)) 7)))
    (iso-date year week day)))
```

```
(defun iso-long-year? (i-year)                                    (5.3)
  ;; TYPE iso-year -> boolean
  ;; True if i-year is a long (53-week) year.
  (let* ((jan1 (day-of-week-from-fixed
                (gregorian-new-year i-year)))
         (dec31 (day-of-week-from-fixed
                 (gregorian-year-end i-year))))
    (or (= jan1 thursday)
        (= dec31 thursday))))
```

D.6 The Icelandic Calendar

```
(defun icelandic-date (year season week weekday)
  ;; TYPE (icelandic-year icelandic-season
  ;; TYPE  icelandic-week icelandic-weekday) -> icelandic-date
  (list year season week weekday))
```

```
(defun icelandic-year (i-date)
  ;; TYPE icelandic-date -> icelandic-year
  (first i-date))
```

```
(defun icelandic-season (i-date)
  ;; TYPE icelandic-date -> icelandic-season
  (second i-date))
```

```
(defun icelandic-week (i-date)
  ;; TYPE icelandic-date -> icelandic-week
  (third i-date))
```

```
(defun icelandic-weekday (i-date)
  ;; TYPE icelandic-date -> icelandic-weekday
  (fourth i-date))
```

(6.1)

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

(6.2)

```
(defun icelandic-summer (i-year)
  ;; TYPE icelandic-year -> fixed-date
  ;; Fixed date of start of Icelandic year i-year.
  (let* ((apr19 (+ icelandic-epoch (* 365 (1- i-year))
                  (sigma ((y (to-radix i-year (list 4 25 4)))
                          (a (list 97 24 1 0)))
                         (* y a)))))
    (kday-on-or-after thursday apr19)))
```

(6.3)

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

(6.4)

```
(defun fixed-from-icelandic (i-date)
  ;; TYPE icelandic-date -> fixed-date
  ;; Fixed date equivalent to Icelandic i-date.
  (let* ((year (icelandic-year i-date))
         (season (icelandic-season i-date))
         (week (icelandic-week i-date))
         (weekday (icelandic-weekday i-date))
         (start ; Start of season.
          (if (= season summer)
              (icelandic-summer year)
            (icelandic-winter year)))
         (shift ; First day of week in prior season.
          (if (= season summer) thursday saturday)))
    (+ start
       (* 7 (1- week)) ; Elapsed weeks.
       (mod (- weekday shift) 7))))
```

(6.5)

```
(defun icelandic-from-fixed (date)
  ;; TYPE fixed-date -> icelandic-date
  ;; Icelandic (year season week weekday) corresponding to
  ;; the fixed date.
  (let* ((approx ; approximate year
          (quotient (- date icelandic-epoch -369)
                    146097/400))
         (year (if (>= date (icelandic-summer approx))
                   approx
                 (1- approx)))
         (season (if (< date (icelandic-winter year))
                     summer
                   winter))
         (start ; Start of current season.
          (if (= season summer)
              (icelandic-summer year)
            (icelandic-winter year)))
         (week ; Weeks since start of season.
          (1+ (quotient (- date start) 7)))
         (weekday (day-of-week-from-fixed date)))
    (icelandic-date year season week weekday)))
```

(6.6)

```
(defun icelandic-leap-year? (i-year)
  ;; TYPE icelandic-year -> boolean
  ;; True if Icelandic i-year is a leap year (53 weeks)
  ;; on the Icelandic calendar.
  (/= (- (icelandic-summer (1+ i-year))
         (icelandic-summer i-year))
      364))
```

(6.7)

```
(defun icelandic-month (i-date)
  ;; TYPE icelandic-date -> icelandic-month
  ;; Month of i-date on the Icelandic calendar.
  ;; Epagomenae are "month" 0.
  (let* ((date (fixed-from-icelandic i-date))
         (year (icelandic-year i-date))
         (season (icelandic-season i-date))
         (midsummer (- (icelandic-winter year) 90))
         (start (cond ((= season winter)
                       (icelandic-winter year))
                      ((>= date midsummer)
                       (- midsummer 90))
                      ((< date (+ (icelandic-summer year) 90))
                       (icelandic-summer year))
                      (t ; Epagomenae.
                       midsummer))))
    (1+ (quotient (- date start) 30))))
```

D.7 The Islamic Calendar

```
(defun islamic-date (year month day)
  ;; TYPE (islamic-year islamic-month islamic-day)
  ;; TYPE  -> islamic-date
  (list year month day))
```

(7.1)

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

(7.2)

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

(7.3)

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

(7.4)

```
(defun islamic-from-fixed (date)
  ;; TYPE fixed-date -> islamic-date
  ;; Islamic date (year month day) corresponding to fixed
  ;; date.
  (let* ((year
          (quotient
           (+ (* 30 (- date islamic-epoch)) 10646)
           10631))
         (prior-days
          (- date (fixed-from-islamic
                   (islamic-date year 1 1))))
         (month
          (quotient
           (+ (* 11 prior-days) 330)
           325))
         (day
          (1+ (- date (fixed-from-islamic
                       (islamic-date year month 1))))))
    (islamic-date year month day)))
```

(7.5)
```
(defun islamic-in-gregorian (i-month i-day g-year)
  ;; TYPE (islamic-month islamic-day gregorian-year)
  ;; TYPE  -> list-of-fixed-dates
  ;; List of the fixed dates of Islamic month i-month, day
  ;; i-day that occur in Gregorian year g-year.
  (let* ((jan1 (gregorian-new-year g-year))
         (y (standard-year (islamic-from-fixed jan1)))
         ;; The possible occurrences in one year are
         (date0 (fixed-from-islamic
                 (islamic-date y i-month i-day)))
         (date1 (fixed-from-islamic
                 (islamic-date (1+ y) i-month i-day)))
         (date2 (fixed-from-islamic
                 (islamic-date (+ y 2) i-month i-day))))
    ;; Combine in one list those that occur in current year
    (list-range (list date0 date1 date2)
                (gregorian-year-range g-year))))
```

(7.6)
```
(defun mawlid (g-year)
  ;; TYPE gregorian-year -> list-of-fixed-dates
  ;; List of fixed dates of Mawlid an-Nabi occurring in
  ;; Gregorian year g-year.
  (islamic-in-gregorian 3 12 g-year))
```

D.8 The Hebrew Calendar

```
(defun hebrew-date (year month day)
  ;; TYPE (hebrew-year hebrew-month hebrew-day) -> hebrew-date
  (list year month day))
```

(8.1)
```
(defconstant nisan
  ;; TYPE hebrew-month
  ;; Nisan is month number 1.
  1)
```

(8.2)
```
(defconstant iyyar
  ;; TYPE hebrew-month
  ;; Iyyar is month number 2.
  2)
```

(8.3)
```
(defconstant sivan
  ;; TYPE hebrew-month
  ;; Sivan is month number 3.
  3)
```

(8.4)
```
(defconstant tammuz
  ;; TYPE hebrew-month
  ;; Tammuz is month number 4.
  4)
```

(8.5)
```
(defconstant av
  ;; TYPE hebrew-month
  ;; Av is month number 5.
  5)
```

(8.6)
```
(defconstant elul
  ;; TYPE hebrew-month
  ;; Elul is month number 6.
  6)
```

(8.7)
```
(defconstant tishri
  ;; TYPE hebrew-month
  ;; Tishri is month number 7.
  7)
```

(8.8)
```
(defconstant marheshvan
  ;; TYPE hebrew-month
  ;; Marheshvan is month number 8.
  8)
```

(8.9)
```
(defconstant kislev
  ;; TYPE hebrew-month
  ;; Kislev is month number 9.
  9)
```

(8.10)
```
(defconstant tevet
  ;; TYPE hebrew-month
  ;; Tevet is month number 10.
  10)
```

(8.11)
```
(defconstant shevat
  ;; TYPE hebrew-month
  ;; Shevat is month number 11.
  11)
```

(8.12)
```
(defconstant adar
  ;; TYPE hebrew-month
  ;; Adar is month number 12.
  12)
```

(8.13)
```
(defconstant adarii
  ;; TYPE hebrew-month
  ;; Adar II is month number 13.
  13)
```

(8.14)
```
(defun hebrew-leap-year? (h-year)
  ;; TYPE hebrew-year -> boolean
  ;; True if h-year is a leap year on Hebrew calendar.
  (< (mod (1+ (* 7 h-year)) 19) 7))
```

(8.15)
```
(defun last-month-of-hebrew-year (h-year)
  ;; TYPE hebrew-year -> hebrew-month
  ;; Last month of Hebrew year h-year.
  (if (hebrew-leap-year? h-year)
      adarii
    adar))
```

(8.16)
```
(defun hebrew-sabbatical-year? (h-year)
  ;; TYPE hebrew-year -> boolean
  ;; True if h-year is a sabbatical year on the Hebrew
  ;; calendar.
  (= (mod h-year 7) 0))
```

(8.17)
```
(defconstant hebrew-epoch
  ;; TYPE fixed-date
  ;; Fixed date of start of the Hebrew calendar, that is,
  ;; Tishri 1, 1 AM.
  (fixed-from-julian (julian-date (bce 3761) october 7)))
```

(8.19)
```
(defun molad (h-year h-month)
  ;; TYPE (hebrew-year hebrew-month) -> rational-moment
  ;; Moment of mean conjunction of h-month in Hebrew
  ;; h-year.
  (let* ((y ;; Treat Nisan as start of year.
          (if (< h-month tishri)
              (1+ h-year)
            h-year))
```

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

(8.20)

```
(defun hebrew-calendar-elapsed-days (h-year)
  ;; TYPE hebrew-year -> integer
  ;; Number of days elapsed from the (Sunday) noon prior
  ;; to the epoch of the Hebrew calendar to the mean
  ;; conjunction (molad) of Tishri of Hebrew year h-year,
  ;; or one day later.
  (let* ((months-elapsed  ; Since start of Hebrew calendar.
          (quotient (- (* 235 h-year) 234) 19))
         (parts-elapsed; Fractions of days since prior noon.
          (+ 12084 (* 13753 months-elapsed)))
         (days  ; Whole days since prior noon.
          (+ (* 29 months-elapsed)
             (quotient parts-elapsed 25920)))
         ;; If (* 13753 months-elapsed) causes integers that
         ;; are too large, use instead:
         ;; (parts-elapsed
         ;;  (+ 204 (* 793 (mod months-elapsed 1080))))
         ;; (hours-elapsed
         ;;  (+ 11 (* 12 months-elapsed)
         ;;     (* 793 (quotient months-elapsed 1080))
         ;;     (quotient parts-elapsed 1080)))
         ;; (days
         ;;  (+ (* 29 months-elapsed)
         ;;     (quotient hours-elapsed 24)))
         ;; If even larger integers aren't a problem, use just:
         ;; (days
         ;;  (quotient (+ 12084 (* months-elapsed 765433))
         ;;            25920)))
         )
    (if (< (mod (* 3 (1+ days)) 7) 3); Sun, Wed, or Fri
        (+ days 1) ; Delay one day.
      days)))
```

(8.21)

```
(defun hebrew-year-length-correction (h-year)
  ;; TYPE hebrew-year -> 0-2
  ;; Delays to start of Hebrew year h-year to keep ordinary
  ;; year in range 353-356 and leap year in range 383-386.
  (let* ((ny0 (hebrew-calendar-elapsed-days (1- h-year)))
         (ny1 (hebrew-calendar-elapsed-days h-year))
         (ny2 (hebrew-calendar-elapsed-days (1+ h-year))))
    (cond
     ((= (- ny2 ny1) 356) ; Next year would be too long.
      2)
     ((= (- ny1 ny0) 382) ; Previous year too short.
      1)
     (t 0))))
```

(8.22)

```
(defun hebrew-new-year (h-year)
  ;; TYPE hebrew-year -> fixed-date
  ;; Fixed date of Hebrew new year h-year.
  (+ hebrew-epoch
     (hebrew-calendar-elapsed-days h-year)
     (hebrew-year-length-correction h-year)))
```

(8.23)

```
(defun last-day-of-hebrew-month (h-year h-month)
  ;; TYPE (hebrew-year hebrew-month) -> hebrew-day
  ;; Last day of month h-month in Hebrew year h-year.
  (if (or (member h-month
                  (list iyyar tammuz elul tevet adarii))
          (and (= h-month adar)
               (not (hebrew-leap-year? h-year)))
```

```
          (and (= h-month marheshvan)
               (not (long-marheshvan? h-year)))
          (and (= h-month kislev)
               (short-kislev? h-year)))
      29
    30))
```

(8.24)

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

(8.25)

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

(8.26)

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

(8.27)

```
(defun fixed-from-hebrew (h-date)
  ;; TYPE hebrew-date -> fixed-date
  ;; Fixed date of Hebrew date h-date.
  (let* ((month (standard-month h-date))
         (day (standard-day h-date))
         (year (standard-year h-date)))
    (+ (hebrew-new-year year)
       day -1               ; Days so far this month.
       (if ;; before Tishri
           (< month tishri)
           ;; Then add days in prior months this year before
           ;; and after Nisan.
           (+ (sum (last-day-of-hebrew-month year m)
                   m tishri
                   (<= m (last-month-of-hebrew-year year)))
              (sum (last-day-of-hebrew-month year m)
                   m nisan (< m month)))
         ;; Else add days in prior months this year
         (sum (last-day-of-hebrew-month year m)
              m tishri (< m month))))))
```

(8.28)

```
(defun hebrew-from-fixed (date)
  ;; TYPE fixed-date -> hebrew-date
  ;; Hebrew (year month day) corresponding to fixed date.
  ;; The fraction can be approximated by 365.25.
  (let* ((approx    ; Approximate year
          (1+
           (quotient (- date hebrew-epoch) 35975351/98496)))
         ;; The value 35975351/98496, the average length of
         ;; a Hebrew year, can be approximated by 365.25
         (year      ; Search forward.
          (final y (1- approx)
                 (<= (hebrew-new-year y) date)))
         (start     ; Starting month for search for month.
          (if (< date (fixed-from-hebrew
                       (hebrew-date year nisan 1)))
              tishri
            nisan))
         (month ; Search forward from either Tishri or Nisan.
          (next m start
                (<= date
                    (fixed-from-hebrew
                     (hebrew-date
                      year
```

```
                        m
                        (last-day-of-hebrew-month year m))))))
             (day   ; Calculate the day by subtraction.
              (1+ (- date (fixed-from-hebrew
                           (hebrew-date year month 1))))))
        (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.

(8.29)

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

(This latter function requires 64-bit integers.)

(8.30)

```
(defun yom-kippur (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Yom Kippur occurring in Gregorian year
  ;; g-year.
  (let* ((h-year
          (1+ (- g-year
                 (gregorian-year-from-fixed
                  hebrew-epoch)))))
    (fixed-from-hebrew (hebrew-date h-year tishri 10))))
```

(8.31)

```
(defun passover (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Passover occurring in Gregorian year
  ;; g-year.
  (let* ((h-year
          (- g-year
             (gregorian-year-from-fixed hebrew-epoch))))
    (fixed-from-hebrew (hebrew-date h-year nisan 15))))
```

(8.32)

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

(8.33)

```
(defun purim (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Purim occurring in Gregorian year g-year.
  (let* ((h-year
          (- g-year
             (gregorian-year-from-fixed hebrew-epoch)))
         (last-month  ; Adar or Adar II
          (last-month-of-hebrew-year h-year)))
    (fixed-from-hebrew
     (hebrew-date h-year last-month 14))))
```

(8.34)

```
(defun ta-anit-esther (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Ta'anit Esther occurring in
  ;; Gregorian year g-year.
  (let* ((purim-date (purim g-year)))
```

```
    (if ; Purim is on Sunday
        (= (day-of-week-from-fixed purim-date) sunday)
        ;; Then prior Thursday
        (- purim-date 3)
      ;; Else previous day
      (1- purim-date))))
```

(8.35)

```
(defun tishah-be-av (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Tishah be-Av occurring in
  ;; Gregorian year g-year.
  (let* ((h-year ; Hebrew year
          (- g-year
             (gregorian-year-from-fixed hebrew-epoch)))
         (av9
          (fixed-from-hebrew
           (hebrew-date h-year av 9))))
    (if ; Ninth of Av is Saturday
        (= (day-of-week-from-fixed av9) saturday)
        ;; Then the next day
        (1+ av9)
      av9)))
```

(8.36)

```
(defun yom-ha-zikkaron (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Yom ha-Zikkaron occurring in Gregorian
  ;; year g-year.
  (let* ((h-year ; Hebrew year
          (- g-year
             (gregorian-year-from-fixed hebrew-epoch)))
         (iyyar4; Ordinarily Iyyar 4
          (fixed-from-hebrew
           (hebrew-date h-year iyyar 4))))
    (cond ((member (day-of-week-from-fixed iyyar4)
                   (list thursday friday))
           ;; If Iyyar 4 is Thursday or Friday, then Wednesday
           (kday-before wednesday iyyar4))
          ;; If it's on Sunday, then Monday
          ((= sunday (day-of-week-from-fixed iyyar4))
           (1+ iyyar4))
          (t iyyar4))))
```

(8.37)

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

(8.38)

```
(defun birkath-ha-hama (g-year)
  ;; TYPE gregorian-year -> list-of-fixed-dates
  ;; List of fixed date of Birkath ha-Hama occurring in
  ;; Gregorian year g-year, if it occurs.
  (let* ((dates (coptic-in-gregorian 7 30 g-year)))
    (if (and (not (equal dates nil))
             (= (mod (standard-year
                      (coptic-from-fixed (first dates)))
                     28)
                17))
        dates
      nil)))
```

(8.39)

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

```
          (* (/ season (deg 360)) cap-Y)))
    (cycle-in-gregorian season g-year cap-Y
                        (+ (fixed-from-hebrew
                            (hebrew-date 1 adar 21))
                           (hr 18)
                           offset))))
```

(8.40)

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

(8.41)

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

(8.42)

```
(defun hebrew-in-gregorian (h-month h-day g-year)
  ;; TYPE (hebrew-month hebrew-day gregorian-year)
  ;; TYPE  -> list-of-fixed-dates
  ;; List of the fixed dates of Hebrew month h-month, day
  ;; h-day that occur in Gregorian year g-year.
  (let* ((jan1 (gregorian-new-year g-year))
         (y (standard-year (hebrew-from-fixed jan1)))
         ;; The possible occurrences in one year are
         (date0 (fixed-from-hebrew
                 (hebrew-date y h-month h-day)))
         (date1 (fixed-from-hebrew
                 (hebrew-date (1+ y) h-month h-day)))
         (date2 (fixed-from-hebrew
                 (hebrew-date (+ y 2) h-month h-day))))
    (list-range (list date0 date1 date2)
                (gregorian-year-range g-year))))
```

(8.43)

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

(8.44)

```
(defun hebrew-birthday (birthdate h-year)
  ;; TYPE (hebrew-date hebrew-year) -> fixed-date
  ;; Fixed date of the anniversary of Hebrew birthdate
  ;; occurring in Hebrew h-year.
  (let* ((birth-day (standard-day birthdate))
         (birth-month (standard-month birthdate))
         (birth-year (standard-year birthdate)))
    (if ; It's Adar in a normal Hebrew year or Adar II
                                       ; in a Hebrew leap year,
        (= birth-month (last-month-of-hebrew-year birth-year))
        ;; Then use the same day in last month of Hebrew year.
```

```
          (fixed-from-hebrew
           (hebrew-date h-year (last-month-of-hebrew-year h-year)
                        birth-day))
        ;; Else use the normal anniversary of the birth date,
        ;; or the corresponding day in years without that date
        (+ (fixed-from-hebrew
            (hebrew-date h-year birth-month 1))
           birth-day -1))))
```

(8.45)

```
(defun hebrew-birthday-in-gregorian (birthdate g-year)
  ;; TYPE (hebrew-date gregorian-year)
  ;; TYPE  -> list-of-fixed-dates
  ;; List of the fixed dates of Hebrew birthday
  ;; that occur in Gregorian g-year.
  (let* ((jan1 (gregorian-new-year g-year))
         (y (standard-year (hebrew-from-fixed jan1)))
         ;; The possible occurrences in one year are
         (date0 (hebrew-birthday birthdate y))
         (date1 (hebrew-birthday birthdate (1+ y)))
         (date2 (hebrew-birthday birthdate (+ y 2))))
    ;; Combine in one list those that occur in current year.
    (list-range (list date0 date1 date2)
                (gregorian-year-range g-year))))
```

(8.46)

```
(defun yahrzeit (death-date h-year)
  ;; TYPE (hebrew-date hebrew-year) -> fixed-date
  ;; Fixed date of the anniversary of Hebrew death-date
  ;; occurring in Hebrew h-year.
  (let* ((death-day (standard-day death-date))
         (death-month (standard-month death-date))
         (death-year (standard-year death-date)))
    (cond
     ;; If it's Marheshvan 30 it depends on the first
     ;; anniversary; if that was not Marheshvan 30, use
     ;; the day before Kislev 1.
     ((and (= death-month marheshvan)
           (= death-day 30)
           (not (long-marheshvan? (1+ death-year))))
      (1- (fixed-from-hebrew
           (hebrew-date h-year kislev 1))))
     ;; If it's Kislev 30 it depends on the first
     ;; anniversary; if that was not Kislev 30, use
     ;; the day before Tevet 1.
     ((and (= death-month kislev)
           (= death-day 30)
           (short-kislev? (1+ death-year)))
      (1- (fixed-from-hebrew
           (hebrew-date h-year tevet 1))))
     ;; If it's Adar II, use the same day in last
     ;; month of Hebrew year (Adar or Adar II).
     ((= death-month adarii)
      (fixed-from-hebrew
       (hebrew-date
        h-year (last-month-of-hebrew-year h-year)
        death-day)))
     ;; If it's the 30th in Adar I and Hebrew year is not a
     ;; Hebrew leap year (so Adar has only 29 days), use the
     ;; last day in Shevat.
     ((and (= death-day 30)
           (= death-month adar)
           (not (hebrew-leap-year? h-year)))
      (fixed-from-hebrew (hebrew-date h-year shevat 30)))
     ;; In all other cases, use the normal anniversary of
     ;; the date of death.
     (t (+ (fixed-from-hebrew
            (hebrew-date h-year death-month 1))
           death-day -1)))))
```

(8.47)

```
(defun yahrzeit-in-gregorian (death-date g-year)
  ;; TYPE (hebrew-date gregorian-year)
```

```
  ;; TYPE  -> list-of-fixed-dates
  ;; List of the fixed dates of death-date (yahrzeit)
  ;; that occur in Gregorian year g-year.
  (let* ((jan1 (gregorian-new-year g-year))
         (y (standard-year (hebrew-from-fixed jan1)))
         ;; The possible occurrences in one year are
         (date0 (yahrzeit death-date y))
         (date1 (yahrzeit death-date (1+ y)))
         (date2 (yahrzeit death-date (+ y 2))))
    ;; Combine in one list those that occur in current year
    (list-range (list date0 date1 date2)
                (gregorian-year-range g-year))))
```

(8.49)

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

(8.50)

```
(defun possible-hebrew-days (h-month h-day)
  ;; TYPE (hebrew-month hebrew-day) -> list-of-weekdays
  ;; Possible days of week
  (let* ((h-date0 (hebrew-date 5 nisan 1))
         ;; leap year with full pattern
         (h-year (if (> h-month elul) 6 5))
         (h-date (hebrew-date h-year h-month h-day))
         (n (- (fixed-from-hebrew h-date)
               (fixed-from-hebrew h-date0)))
         (basic (list tuesday thursday saturday))
         (extra
          (cond
           ((and (= h-month marheshvan) (= h-day 30))
            nil)
           ((and (= h-month kislev) (< h-day 30))
            (list monday wednesday friday))
           ((and (= h-month kislev) (= h-day 30))
            (list monday))
           ((member h-month (list tevet shevat))
            (list sunday monday))
           ((and (= h-month adar) (< h-day 30))
            (list sunday monday))
           (t (list sunday)))))
    (shift-days (append basic extra) n)))
```

D.9 The Ecclesiastical Calendars

(9.1)

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

(9.2)

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

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

(9.3)

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

(9.4)

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

D.10 The Old Hindu Calendars

(10.1)

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

(10.2)

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

(10.3)

```
(defconstant arya-solar-year
  ;; TYPE rational
  ;; Length of Old Hindu solar year.
  1577917500/4320000)
```

(10.4)

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

(10.5)

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

(10.6)
```
(defconstant arya-solar-month
  ;; TYPE rational
  ;; Length of Old Hindu solar month.
  (/ arya-solar-year 12))
```

(10.7)
```
(defun fixed-from-old-hindu-solar (s-date)
  ;; TYPE hindu-solar-date -> fixed-date
  ;; Fixed date corresponding to Old Hindu solar date s-date.
  (let* ((month (standard-month s-date))
         (day (standard-day s-date))
         (year (standard-year s-date)))
    (ceiling
     (+ hindu-epoch ; Since start of era.
        (* year arya-solar-year) ; Days in elapsed years
        (* (1- month) arya-solar-month) ; ...in months.
        day (hr -30))))) ; Midnight of day.
```

(10.8)
```
(defun old-hindu-solar-from-fixed (date)
  ;; TYPE fixed-date -> hindu-solar-date
  ;; Old Hindu solar date equivalent to fixed date.
  (let* ((sun ; Sunrise on Hindu date.
          (+ (hindu-day-count date) (hr 6)))
         (year    ; Elapsed years.
          (quotient sun arya-solar-year))
         (month (1+ (mod (quotient sun arya-solar-month)
                         12)))
         (day (1+ (floor (mod sun arya-solar-month)))))
    (hindu-solar-date year month day)))
```

```
(defun old-hindu-lunar-date (year month leap day)
  ;; TYPE (old-hindu-lunar-year old-hindu-lunar-month
  ;; TYPE  old-hindu-lunar-leap old-hindu-lunar-day)
  ;; TYPE  -> old-hindu-lunar-date
  (list year month leap day))
```

```
(defun old-hindu-lunar-month (date)
  ;; TYPE old-hindu-lunar-date -> old-hindu-lunar-month
  (second date))
```

```
(defun old-hindu-lunar-leap (date)
  ;; TYPE old-hindu-lunar-date -> old-hindu-lunar-leap
  (third date))
```

```
(defun old-hindu-lunar-day (date)
  ;; TYPE old-hindu-lunar-date -> old-hindu-lunar-day
  (fourth date))
```

```
(defun old-hindu-lunar-year (date)
  ;; TYPE old-hindu-lunar-date -> old-hindu-lunar-year
  (first date))
```

(10.9)
```
(defconstant arya-lunar-month
  ;; TYPE rational
  ;; Length of Old Hindu lunar month.
  1577917500/53433336)
```

(10.10)
```
(defconstant arya-lunar-day
  ;; TYPE rational
  ;; Length of Old Hindu lunar day.
  (/ arya-lunar-month 30))
```

(10.11)

```
(defun old-hindu-lunar-leap-year? (l-year)
  ;; TYPE old-hindu-lunar-year -> boolean
  ;; True if l-year is a leap year on the
  ;; old Hindu calendar.
  (>= (mod (- (* l-year arya-solar-year)
              arya-solar-month)
           arya-lunar-month)
      23902504679/1282400064))
```

(10.13)

```
(defun old-hindu-lunar-from-fixed (date)
  ;; TYPE fixed-date -> old-hindu-lunar-date
  ;; Old Hindu lunar date equivalent to fixed date.
  (let* ((sun ; Sunrise on Hindu date.
          (+ (hindu-day-count date) (hr 6)))
         (new-moon ; Beginning of lunar month.
          (- sun (mod sun arya-lunar-month)))
         (leap ; If lunar contained in solar.
          (and (>= (- arya-solar-month arya-lunar-month)
                   (mod new-moon arya-solar-month))
               (> (mod new-moon arya-solar-month) 0)))
         (month ; Next solar month's name.
          (1+ (mod (ceiling (/ new-moon
                               arya-solar-month))
                   12)))
         (day ; Lunar days since beginning of lunar month.
          (1+ (mod (quotient sun arya-lunar-day) 30)))
         (year ; Solar year at end of lunar month(s).
          (1- (ceiling (/ (+ new-moon arya-solar-month)
                          arya-solar-year)))))
    (old-hindu-lunar-date year month leap day)))
```

(10.14)

```
(defun fixed-from-old-hindu-lunar (l-date)
  ;; TYPE old-hindu-lunar-date -> fixed-date
  ;; Fixed date corresponding to Old Hindu lunar date
  ;; l-date.
  (let* ((year (old-hindu-lunar-year l-date))
         (month (old-hindu-lunar-month l-date))
         (leap (old-hindu-lunar-leap l-date))
         (day (old-hindu-lunar-day l-date))
         (mina ; One solar month before solar new year.
          (* (1- (* 12 year)) arya-solar-month))
         (lunar-new-year ; New moon after mina.
          (* arya-lunar-month
             (1+ (quotient mina arya-lunar-month)))))
    (ceiling
     (+ hindu-epoch
        lunar-new-year
        (* arya-lunar-month
           (if ; If there was a leap month this year.
               (and (not leap)
                    (<= (ceiling (/ (- lunar-new-year mina)
                                    (- arya-solar-month
                                       arya-lunar-month)))
                        month))
               month
             (1- month)))
        (* (1- day) arya-lunar-day) ; Lunar days.
        (hr -6))))) ; Subtract 1 if phase begins before
                    ; sunrise.
```

D.11 The Mayan Calendars

```
(defun mayan-long-count-date (baktun katun tun uinal kin)
  ;; TYPE (mayan-baktun mayan-katun mayan-tun mayan-uinal
  ;; TYPE  mayan-kin) -> mayan-long-count-date
  (list baktun katun tun uinal kin))
```

```
(defun mayan-baktun (date)
  ;; TYPE mayan-long-count-date -> mayan-baktun
  (first date))
```

```lisp
(defun mayan-katun (date)
  ;; TYPE mayan-long-count-date -> mayan-katun
  (second date))
```

```lisp
(defun mayan-tun (date)
  ;; TYPE mayan-long-count-date -> mayan-tun
  (third date))
```

```lisp
(defun mayan-uinal (date)
  ;; TYPE mayan-long-count-date -> mayan-uinal
  (fourth date))
```

```lisp
(defun mayan-kin (date)
  ;; TYPE mayan-long-count-date -> mayan-kin
  (fifth date))
```

(11.1)

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

(11.2)

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

(11.3)

```lisp
(defun mayan-long-count-from-fixed (date)
  ;; TYPE fixed-date -> mayan-long-count-date
  ;; Mayan long count date of fixed date.
  (to-radix (- date mayan-epoch) (list 20 20 18 20)))
```

```lisp
(defun mayan-haab-date (month day)
  ;; TYPE (mayan-haab-month mayan-haab-day) -> mayan-haab-date
  (list month day))
```

```lisp
(defun mayan-haab-day (date)
  ;; TYPE mayan-haab-date -> mayan-haab-day
  (second date))
```

```lisp
(defun mayan-haab-month (date)
  ;; TYPE mayan-haab-date -> mayan-haab-month
  (first date))
```

(11.4)

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

(11.5)

```lisp
(defconstant mayan-haab-epoch
  ;; TYPE fixed-date
  ;; Fixed date of start of haab cycle.
  (- mayan-epoch
     (mayan-haab-ordinal (mayan-haab-date 18 8))))
```

(11.6)

```
(defun mayan-haab-from-fixed (date)
  ;; TYPE fixed-date -> mayan-haab-date
  ;; Mayan haab date of fixed date.
  (let* ((count
          (mod (- date mayan-haab-epoch) 365))
         (day (mod count 20))
         (month (1+ (quotient count 20))))
    (mayan-haab-date month day)))
```

(11.7)

```
(defun mayan-haab-on-or-before (haab date)
  ;; TYPE (mayan-haab-date fixed-date) -> fixed-date
  ;; Fixed date of latest date on or before fixed date
  ;; that is Mayan haab date haab.
  (mod3 (+ (mayan-haab-ordinal haab) mayan-haab-epoch)
        date (- date 365)))
```

```
(defun mayan-tzolkin-date (number name)
  ;; TYPE (mayan-tzolkin-number mayan-tzolkin-name)
  ;; TYPE  -> mayan-tzolkin-date
  (list number name))
```

```
(defun mayan-tzolkin-number (date)
  ;; TYPE mayan-tzolkin-date -> mayan-tzolkin-number
  (first date))
```

```
(defun mayan-tzolkin-name (date)
  ;; TYPE mayan-tzolkin-date -> mayan-tzolkin-name
  (second date))
```

(11.8)

```
(defconstant mayan-tzolkin-epoch
  ;; TYPE fixed-date
  ;; Start of tzolkin date cycle.
  (- mayan-epoch
     (mayan-tzolkin-ordinal (mayan-tzolkin-date 4 20))))
```

(11.9)

```
(defun mayan-tzolkin-from-fixed (date)
  ;; TYPE fixed-date -> mayan-tzolkin-date
  ;; Mayan tzolkin date of fixed date.
  (let* ((count (- date mayan-tzolkin-epoch -1))
         (number (amod count 13))
         (name (amod count 20)))
    (mayan-tzolkin-date number name)))
```

(11.10)

```
(defun mayan-tzolkin-ordinal (t-date)
  ;; TYPE mayan-tzolkin-date -> nonnegative-integer
  ;; Number of days into Mayan tzolkin cycle of t-date.
  (let* ((number (mayan-tzolkin-number t-date))
         (name (mayan-tzolkin-name t-date)))
    (mod (+ number -1
            (* 39 (- number name)))
         260)))
```

(11.11)

```
(defun mayan-tzolkin-on-or-before (tzolkin date)
  ;; TYPE (mayan-tzolkin-date fixed-date) -> fixed-date
  ;; Fixed date of latest date on or before fixed date
  ;; that is Mayan tzolkin date tzolkin.
  (mod3 (+ (mayan-tzolkin-ordinal tzolkin) mayan-tzolkin-epoch)
        date (- date 260)))
```

(11.12)

```
(defun mayan-year-bearer-from-fixed (date)
  ;; TYPE fixed-date -> mayan-tzolkin-name
  ;; Year bearer of year containing fixed date.
  ;; Returns bogus for uayeb.
  (let* ((x (mayan-haab-on-or-before
             (mayan-haab-date 1 0)
             date)))
    (if (= (mayan-haab-month (mayan-haab-from-fixed date))
           19)
        bogus
      (mayan-tzolkin-name (mayan-tzolkin-from-fixed x)))))
```

(11.13)

```
(defun mayan-calendar-round-on-or-before (haab tzolkin date)
  ;; TYPE (mayan-haab-date mayan-tzolkin-date fixed-date)
  ;; TYPE  -> fixed-date
  ;; Fixed date of latest date on or before date, that is
  ;; Mayan haab date haab and tzolkin date tzolkin.
  ;; Returns bogus for impossible combinations.
  (let* ((haab-count
          (+ (mayan-haab-ordinal haab) mayan-haab-epoch))
         (tzolkin-count
          (+ (mayan-tzolkin-ordinal tzolkin)
             mayan-tzolkin-epoch))
         (diff (- tzolkin-count haab-count)))
    (if (= (mod diff 5) 0)
        (mod3 (+ haab-count (* 365 diff))
              date (- date 18980))
      bogus))); haab-tzolkin combination is impossible.
```

(11.14)

```
(defconstant aztec-correlation
  ;; TYPE fixed-date
  ;; Known date of Aztec cycles (Caso's correlation)
  (fixed-from-julian (julian-date 1521 August 13)))
```

```
(defun aztec-xihuitl-date (month day)
  ;; TYPE (aztec-xihuitl-month aztec-xihuitl-day) ->
  ;; TYPE  aztec-xihuitl-date
  (list month day))
```

```
(defun aztec-xihuitl-month (date)
  ;; TYPE aztec-xihuitl-date -> aztec-xihuitl-month
  (first date))
```

```
(defun aztec-xihuitl-day (date)
  ;; TYPE aztec-xihuitl-date -> aztec-xihuitl-day
  (second date))
```

(11.15)

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

(11.16)

```
(defconstant aztec-xihuitl-correlation
  ;; TYPE fixed-date
  ;; Start of a xihuitl cycle.
  (- aztec-correlation
     (aztec-xihuitl-ordinal (aztec-xihuitl-date 11 2))))
```

(11.17)

```
(defun aztec-xihuitl-from-fixed (date)
  ;; TYPE fixed-date -> aztec-xihuitl-date
  ;; Aztec xihuitl date of fixed date.
  (let* ((count (mod (- date aztec-xihuitl-correlation) 365))
         (day (1+ (mod count 20)))
         (month (1+ (quotient count 20))))
    (aztec-xihuitl-date month day)))
```

(11.18)

```
(defun aztec-xihuitl-on-or-before (xihuitl date)
  ;; TYPE (aztec-xihuitl-date fixed-date) -> fixed-date
  ;; Fixed date of latest date on or before fixed date
  ;; that is Aztec xihuitl date xihuitl.
  (mod3 (+ aztec-xihuitl-correlation
           (aztec-xihuitl-ordinal xihuitl))
        date (- date 365)))
```

```
(defun aztec-tonalpohualli-date (number name)
  ;; TYPE (aztec-tonalpohualli-number aztec-tonalpohualli-name)
  ;; TYPE  -> aztec-tonalpohualli-date
  (list number name))
```

```
(defun aztec-tonalpohualli-number (date)
  ;; TYPE aztec-tonalpohualli-date -> aztec-tonalpohualli-number
  (first date))
```

```
(defun aztec-tonalpohualli-name (date)
  ;; TYPE aztec-tonalpohualli-date -> aztec-tonalpohualli-name
  (second date))
```

```
(defun aztec-tonalpohualli-ordinal (t-date)                    (11.19)
  ;; TYPE aztec-tonalpohualli-date -> nonnegative-integer
  ;; Number of days into Aztec tonalpohualli cycle of t-date.
  (let* ((number (aztec-tonalpohualli-number t-date))
         (name (aztec-tonalpohualli-name t-date)))
    (mod (+ number -1
            (* 39 (- number name)))
         260)))
```

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

```
(defun aztec-tonalpohualli-from-fixed (date)                   (11.21)
  ;; TYPE fixed-date -> aztec-tonalpohualli-date
  ;; Aztec tonalpohualli date of fixed date.
  (let* ((count (- date aztec-tonalpohualli-correlation -1))
         (number (amod count 13))
         (name (amod count 20)))
    (aztec-tonalpohualli-date number name)))
```

```
(defun aztec-tonalpohualli-on-or-before (tonalpohualli date)   (11.22)
  ;; TYPE (aztec-tonalpohualli-date fixed-date) -> fixed-date
  ;; Fixed date of latest date on or before fixed date
  ;; that is Aztec tonalpohualli date tonalpohualli.
  (mod3 (+ aztec-tonalpohualli-correlation
           (aztec-tonalpohualli-ordinal tonalpohualli))
        date (- date 260)))
```

```
(defun aztec-xiuhmolpilli-designation (number name)
  ;; TYPE (aztec-xiuhmolpilli-number aztec-xiuhmolpilli-name)
  ;; TYPE  -> aztec-xiuhmolpilli-designation
  (list number name))
```

```
(defun aztec-xiuhmolpilli-number (date)
  ;; TYPE aztec-xiuhmolpilli-designation -> aztec-xiuhmolpilli-number
  (first date))
```

```
(defun aztec-xiuhmolpilli-name (date)
  ;; TYPE aztec-xiuhmolpilli-designation -> aztec-xiuhmolpilli-name
  (second date))
```

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

```
  (let* ((x (aztec-xihuitl-on-or-before
             (aztec-xihuitl-date 18 20)
             (+ date 364)))
         (month (aztec-xihuitl-month
                 (aztec-xihuitl-from-fixed date))))
    (if (= month 19)
        bogus
      (aztec-tonalpohualli-from-fixed x))))
```

(11.24)

```
(defun aztec-xihuitl-tonalpohualli-on-or-before
  (xihuitl tonalpohualli date)
  ;; TYPE (aztec-xihuitl-date aztec-tonalpohualli-date
  ;; TYPE  fixed-date) -> fixed-date
  ;; Fixed date of latest xihuitl-tonalpohualli combination
  ;; on or before date.  That is the date on or before
  ;; date that is Aztec xihuitl date xihuitl and
  ;; tonalpohualli date tonalpohualli.
  ;; Returns bogus for impossible combinations.
  (let* ((xihuitl-count
          (+ (aztec-xihuitl-ordinal xihuitl)
             aztec-xihuitl-correlation))
         (tonalpohualli-count
          (+ (aztec-tonalpohualli-ordinal tonalpohualli)
             aztec-tonalpohualli-correlation))
         (diff (- tonalpohualli-count xihuitl-count)))
    (if (= (mod diff 5) 0)
        (mod3 (+ xihuitl-count (* 365 diff))
              date (- date 18980))
      bogus))); xihuitl-tonalpohualli combination is impossible.
```

D.12 The Balinese Pawukon Calendar

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

```
(defun bali-luang (b-date)
  ;; TYPE balinese-date -> boolean
  (first b-date))
```

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

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

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

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

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

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

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

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

```
(defun bali-dasawara (b-date)
  ;; TYPE balinese-date -> 0-9
  (tenth b-date))
```

(12.1)

```
(defun bali-pawukon-from-fixed (date)
  ;; TYPE fixed-date -> balinese-date
  ;; Positions of date in ten cycles of Balinese Pawukon
  ;; calendar.
  (balinese-date (bali-luang-from-fixed date)
                 (bali-dwiwara-from-fixed date)
                 (bali-triwara-from-fixed date)
                 (bali-caturwara-from-fixed date)
                 (bali-pancawara-from-fixed date)
                 (bali-sadwara-from-fixed date)
                 (bali-saptawara-from-fixed date)
                 (bali-asatawara-from-fixed date)
                 (bali-sangawara-from-fixed date)
                 (bali-dasawara-from-fixed date)))
```

(12.2)

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

(12.3)

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

(12.4)

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

(12.5)

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

(12.6)

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

(12.7)

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

(12.8)

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

(12.9)

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

(12.10)

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

(12.11)

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

(12.12)

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

(12.13)

```
(defun bali-asatawara-from-fixed (date)
  ;; TYPE fixed-date -> 1-8
  ;; Position of date in 8-day Balinese cycle.
  (let* ((day (bali-day-from-fixed date)))
    (1+ (mod
         (max 6
              (+ 4 (mod (- day 70)
                        210)))
         8))))
```

(12.14)

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

(12.15)

```
(defun bali-on-or-before (b-date date)
  ;; TYPE (balinese-date fixed-date) -> fixed-date
  ;; Last fixed date on or before date with Pawukon b-date.
  (let* ((luang (bali-luang b-date))
         (dwiwara (bali-dwiwara b-date))
         (triwara (bali-triwara b-date))
         (caturwara (bali-caturwara b-date))
         (pancawara (bali-pancawara b-date))
         (sadwara (bali-sadwara b-date))
         (saptawara (bali-saptawara b-date))
         (asatawara (bali-asatawara b-date))
         (sangawara (bali-sangawara b-date))
         (dasawara (bali-dasawara b-date))
         (a5 ; Position in 5-day subcycle.
          (1- pancawara))
         (a6 ; Position in 6-day subcycle.
          (1- sadwara))
         (b7 ; Position in 7-day subcycle.
          (1- saptawara))
         (b35 ; Position in 35-day subcycle.
          (mod (+ a5 14 (* 15 (- b7 a5))) 35))
         (days ; Position in full cycle.
          (+ a6 (* 36 (- b35 a6))))
         (cap-Delta (bali-day-from-fixed (rd 0))))
    (- date (mod (- (+ date cap-Delta) days) 210))))
```

(12.16)

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

(12.17)

```
(defun tumpek (g-year)
  ;; TYPE gregorian-year -> list-of-fixed-dates
  ;; Occurrences of Tumpek (14th day of Pawukon and every
  ;; 35th subsequent day) within Gregorian year g-year.
  (let* ((year (gregorian-year-range g-year))
         (cap-Delta (bali-day-from-fixed (rd 0))))
    (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:

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

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

```
(defun sin-degrees (theta)
  ;; TYPE angle -> amplitude
  ;; Sine of theta (given in degrees).
  (sin (radians-from-degrees theta)))
```

```
(defun cos-degrees (theta)
  ;; TYPE angle -> amplitude
  ;; Cosine of theta (given in degrees).
  (cos (radians-from-degrees theta)))
```

```
(defun tan-degrees (theta)
  ;; TYPE angle -> real
  ;; Tangent of theta (given in degrees).
  (tan (radians-from-degrees theta)))
```

(14.7)

```
(defun arctan-degrees (y x)
  ;; TYPE (real real) -> angle
  ;; Arctangent of y/x in degrees.
  ;; Returns bogus if x and y are both 0.
  (if (and (= x y 0))
      bogus
    (mod
     (if (= x 0)
         (* (sign y) (deg 90L0))
       (let* ((alpha (degrees-from-radians
                      (atan (/ y x)))))
         (if (>= x 0)
             alpha
           (+ alpha (deg 180L0)))))
     360)))
```

```
(defun arcsin-degrees (x)
  ;; TYPE amplitude -> angle
  ;; Arcsine of x in degrees.
  (degrees-from-radians (asin x)))
```

```
(defun arccos-degrees (x)
  ;; TYPE amplitude -> angle
  ;; Arccosine of x in degrees.
  (degrees-from-radians (acos x)))
```

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

```
(defun hr (x)
  ;; TYPE real -> duration
  ;; x hours.
  (/ x 24))
```

```
(defun mn (x)
  ;; TYPE real -> duration
  ;; x minutes.
  (/ x 24 60))
```

```
(defun sec (x)
  ;; TYPE real -> duration
  ;; x seconds.
  (/ x 24 60 60))
```

```
(defun mt (x)
  ;; TYPE real -> distance
  ;; x meters.
  ;; For typesetting purposes.
  x)
```

```
(defun deg (x)
  ;; TYPE real -> angle
  ;; TYPE list-of-reals -> list-of-angles
  ;; x degrees.
  ;; For typesetting purposes.
  x)
```

```
(defun mins (x)
  ;; TYPE real -> angle
  ;; x arcminutes
  (/ x 60))
```

```
(defun secs (x)
  ;; TYPE real -> angle
  ;; x arcseconds
  (/ x 3600))
```

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

```
(defun degrees-minutes-seconds (d m s)
  ;; TYPE (degree minute real) -> angle
  (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:

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

```
(defun latitude (location)
  ;; TYPE location -> half-circle
  (first location))
```

```
(defun longitude (location)
  ;; TYPE location -> circle
  (second location))
```

```
(defun elevation (location)
  ;; TYPE location -> distance
  (third location))
```

```
(defun zone (location)
  ;; TYPE location -> real
  (fourth location))
```

(14.3)

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

(14.4)

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

(14.5)

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

(14.6)

```
(defun direction (location focus)
  ;; TYPE (location location) -> angle
  ;; Angle (clockwise from North) to face focus when
  ;; standing in location.  Subject to errors near focus and
  ;; its antipode.
  (let* ((phi (latitude location))
         (phi-prime (latitude focus))
         (psi (longitude location))
         (psi-prime (longitude focus))
         (y (sin-degrees (- psi-prime psi)))
         (x
          (- (* (cos-degrees phi)
                (tan-degrees phi-prime))
             (* (sin-degrees phi)
                (cos-degrees
                 (- psi psi-prime))))))
    (cond ((or (= x y 0) (= phi-prime (deg 90)))
           (deg 0))
          ((= phi-prime (deg -90))
           (deg 180))
          (t (arctan-degrees y x)))))
```

The following functions compute times:

(14.8)

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

(14.9)

```
(defun universal-from-local (tee_ell location)
  ;; TYPE (moment location) -> moment
  ;; Universal time from local tee_ell at location.
  (- tee_ell (zone-from-longitude (longitude location))))
```

(14.10)
```
(defun local-from-universal (tee_rom-u location)
  ;; TYPE (moment location) -> moment
  ;; Local time from universal tee_rom-u at location.
  (+ tee_rom-u (zone-from-longitude (longitude location))))
```

(14.11)
```
(defun standard-from-universal (tee_rom-u location)
  ;; TYPE (moment location) -> moment
  ;; Standard time from tee_rom-u in universal time at
  ;; location.
  (+ tee_rom-u (zone location)))
```

(14.12)
```
(defun universal-from-standard (tee_rom-s location)
  ;; TYPE (moment location) -> moment
  ;; Universal time from tee_rom-s in standard time at
  ;; location.
  (- tee_rom-s (zone location)))
```

(14.13)
```
(defun standard-from-local (tee_ell location)
  ;; TYPE (moment location) -> moment
  ;; Standard time from local tee_ell at location.
  (standard-from-universal
   (universal-from-local tee_ell location)
   location))
```

(14.14)
```
(defun local-from-standard (tee_rom-s location)
  ;; TYPE (moment location) -> moment
  ;; Local time from standard tee_rom-s at location.
  (local-from-universal
   (universal-from-standard tee_rom-s location)
   location))
```

(14.15)
```
(defun ephemeris-correction (tee)
  ;; TYPE moment -> fraction-of-day
  ;; Dynamical Time minus Universal Time (in days) for
  ;; moment tee.  Adapted from "Astronomical Algorithms"
  ;; by Jean Meeus, Willmann-Bell (1991) for years
  ;; 1600-1986 and from polynomials on the NASA
  ;; Eclipse web site for other years.
  (let* ((year (gregorian-year-from-fixed (floor tee)))
         (c (/ (gregorian-date-difference
                (gregorian-date 1900 january 1)
                (gregorian-date year july 1))
               36525))
         (c2051 (* 1/86400
                   (+ -20 (* 32 (expt (/ (- year 1820) 100) 2))
                      (* 0.5628L0 (- 2150 year)))))
         (y2000 (- year 2000))
         (c2006 (* 1/86400
                   (poly y2000
                         (list 62.92L0 0.32217L0 0.005589L0))))
         (c1987 (* 1/86400
                   (poly y2000
                         (list 63.86L0 0.3345L0 -0.060374L0
                               0.0017275L0
                               0.000651814L0 0.00002373599L0))))
         (c1900 (poly c
                      (list -0.00002L0 0.000297L0 0.025184L0
                            -0.181133L0 0.553040L0 -0.861938L0
                            0.677066L0 -0.212591L0)))
         (c1800 (poly c
                      (list -0.000009L0 0.003844L0 0.083563L0
                            0.865736L0
                            4.867575L0 15.845535L0 31.332267L0
                            38.291999L0 28.316289L0 11.636204L0
                            2.043794L0)))
         (y1700 (- year 1700))
         (c1700 (* 1/86400
```

```
                    (poly y1700
                          (list 8.118780842L0 -0.005092142L0
                                0.003336121L0 -0.0000266484L0))))
          (y1600 (- year 1600))
          (c1600 (* 1/86400
                    (poly y1600
                          (list 120 -0.9808L0 -0.01532L0
                                0.000140272128L0))))
          (y1000 (/ (- year 1000) 100L0))
          (c500 (* 1/86400
                   (poly y1000
                         (list 1574.2L0 -556.01L0 71.23472L0 0.319781L0
                               -0.8503463L0 -0.005050998L0
                               0.0083572073L0))))
          (y0 (/ year 100L0))
          (c0 (* 1/86400
                 (poly y0
                       (list 10583.6L0 -1014.41L0 33.78311L0
                             -5.952053L0 -0.1798452L0 0.022174192L0
                             0.0090316521L0))))
          (y1820 (/ (- year 1820) 100L0))
          (other (* 1/86400
                    (poly y1820 (list -20 0 32)))))
     (cond ((<= 2051 year 2150) c2051)
           ((<= 2006 year 2050) c2006)
           ((<= 1987 year 2005) c1987)
           ((<= 1900 year 1986) c1900)
           ((<= 1800 year 1899) c1800)
           ((<= 1700 year 1799) c1700)
           ((<= 1600 year 1699) c1600)
           ((<= 500 year 1599) c500)
           ((< -500 year 500) c0)
           (t other))))
```

(14.16)

```
(defun dynamical-from-universal (tee_rom-u)
  ;; TYPE moment -> moment
  ;; Dynamical time at Universal moment tee_rom-u.
  (+ tee_rom-u (ephemeris-correction tee_rom-u)))
```

(14.17)

```
(defun universal-from-dynamical (tee)
  ;; TYPE moment -> moment
  ;; Universal moment from Dynamical time tee.
  (- tee (ephemeris-correction tee)))
```

(14.18)

```
(defun julian-centuries (tee)
  ;; TYPE moment -> century
  ;; Julian centuries since 2000 at moment tee.
  (/ (- (dynamical-from-universal tee) j2000)
     36525))
```

(14.19)

```
(defconstant j2000
  ;; TYPE moment
  ;; Noon at start of Gregorian year 2000.
  (+ (hr 12L0) (gregorian-new-year 2000)))
```

(14.20)

```
(defun equation-of-time (tee)
  ;; TYPE moment -> fraction-of-day
  ;; Equation of time (as fraction of day) for moment tee.
  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
  ;; Willmann-Bell, 2nd edn., 1998, p. 185.
  (let* ((c (julian-centuries tee))
         (lambda
           (poly c
                 (deg (list 280.46645L0 36000.76983L0
                            0.0003032L0))))
         (anomaly
          (poly c
                (deg (list 357.52910L0 35999.05030L0
                           -0.0001559L0 -0.00000048L0))))
```

```
           (eccentricity
            (poly c
                  (list 0.016708617L0 -0.000042037L0
                        -0.0000001236L0)))
           (varepsilon (obliquity tee))
           (y (expt (tan-degrees (/ varepsilon 2)) 2))
           (equation
            (* (/ 1 2 pi)
               (+ (* y (sin-degrees (* 2 lambda)))
                  (* -2 eccentricity (sin-degrees anomaly))
                  (* 4 eccentricity y (sin-degrees anomaly)
                     (cos-degrees (* 2 lambda)))
                  (* -0.5L0 y y (sin-degrees (* 4 lambda)))
                  (* -1.25L0 eccentricity eccentricity
                     (sin-degrees (* 2 anomaly)))))))
      (* (sign equation) (min (abs equation) (hr 12L0)))))
```

```
(defun apparent-from-local (tee_ell location)                    (14.21)
  ;; TYPE (moment location) -> moment
  ;; Sundial time from local time tee_ell at location.
  (+ tee_ell (equation-of-time
              (universal-from-local tee_ell location))))
```

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

```
(defun apparent-from-universal (tee_rom-u location)              (14.23)
  ;; TYPE (moment location) -> moment
  ;; True (apparent) time at universal time tee at location.
  (apparent-from-local
   (local-from-universal tee_rom-u location)
   location))
```

```
(defun universal-from-apparent (tee location)                    (14.24)
  ;; TYPE (moment location) -> moment
  ;; Universal time from sundial time tee at location.
  (universal-from-local
   (local-from-apparent tee location)
   location))
```

```
(defun midnight (date location)                                  (14.25)
  ;; TYPE (fixed-date location) -> moment
  ;; Universal time of true (apparent)
  ;; midnight of fixed date at location.
  (universal-from-apparent date location))
```

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

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

Additional solar and lunar astronomical functions are:

(14.28)
```
(defun obliquity (tee)
  ;; TYPE moment -> angle
  ;; Obliquity of ecliptic at moment tee.
  (let* ((c (julian-centuries tee)))
    (+ (angle 23 26 21.448L0)
       (poly c (list 0L0
                     (angle 0 0 -46.8150L0)
                     (angle 0 0 -0.00059L0)
                     (angle 0 0 0.001813L0))))))
```

(14.29)
```
(defun declination (tee beta lambda)
  ;; TYPE (moment half-circle circle) -> angle
  ;; Declination at moment UT tee of object at
  ;; latitude beta and longitude lambda.
  (let* ((varepsilon (obliquity tee)))
    (arcsin-degrees (+ (* (sin-degrees beta)
                          (cos-degrees varepsilon))
                       (* (cos-degrees beta)
                          (sin-degrees varepsilon)
                          (sin-degrees lambda))))))
```

(14.30)
```
(defun right-ascension (tee beta lambda)
  ;; TYPE (moment half-circle circle) -> angle
  ;; Right ascension at moment UT tee of object at
  ;; latitude beta and longitude lambda.
  (let* ((varepsilon (obliquity tee)))
    (arctan-degrees ; Cannot be bogus
     (- (* (sin-degrees lambda)
           (cos-degrees varepsilon))
        (* (tan-degrees beta)
           (sin-degrees varepsilon)))
     (cos-degrees lambda))))
```

(14.31)
```
(defconstant mean-tropical-year
  ;; TYPE duration
  365.242189L0)
```

(14.32)
```
(defconstant mean-sidereal-year
  ;; TYPE duration
  365.25636L0)
```

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

```
                34777.243L0 1221.999L0 62894.511L0
                -4442.039L0 107997.909L0 119.066L0 16859.071L0
                -4.578L0 26895.292L0 -39.127L0 12297.536L0
                90073.778L0))
         (addends
          (list 270.54861L0 340.19128L0 63.91854L0 331.26220L0
                317.843L0 86.631L0 240.052L0 310.26L0 247.23L0
                260.87L0 297.82L0 343.14L0 166.79L0 81.53L0
                3.50L0 132.75L0 182.95L0 162.03L0 29.8L0
                266.4L0 249.2L0 157.6L0 257.8L0 185.1L0 69.9L0
                8.0L0 197.1L0 250.4L0 65.3L0 162.7L0 341.5L0
                291.6L0 98.5L0 146.7L0 110.0L0 5.2L0 342.6L0
                230.9L0 256.1L0 45.3L0 242.9L0 115.2L0 151.8L0
                285.3L0 53.3L0 126.6L0 205.7L0 85.9L0
                146.1L0))
         (lambda
           (+ (deg 282.7771834L0)
              (* (deg 36000.76953744L0) c)
              (* (deg 0.000005729577951308232L0)
                 (sigma ((x coefficients)
                         (y addends)
                         (z multipliers))
                        (* x (sin-degrees (+ y (* z c))))))))))
    (mod (+ lambda (aberration tee) (nutation tee))
         360)))
```

(14.34)

```
(defun nutation (tee)
  ;; TYPE moment -> circle
  ;; Longitudinal nutation at moment tee.
  (let* ((c       ; moment in Julian centuries
          (julian-centuries tee))
         (cap-A (poly c (deg (list 124.90L0 -1934.134L0
                                   0.002063L0))))
         (cap-B (poly c (deg (list 201.11L0 72001.5377L0
                                   0.00057L0)))))
    (+ (* (deg -0.004778L0) (sin-degrees cap-A))
       (* (deg -0.0003667L0) (sin-degrees cap-B)))))
```

(14.35)

```
(defun aberration (tee)
  ;; TYPE moment -> circle
  ;; Aberration at moment tee.
  (let* ((c       ; moment in Julian centuries
          (julian-centuries tee)))
    (- (* (deg 0.0000974L0)
          (cos-degrees
           (+ (deg 177.63L0) (* (deg 35999.01848L0) c))))
       (deg 0.005575L0))))
```

(14.36)

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

(14.37)

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

(14.39)

```
(defun precession (tee)
  ;; TYPE moment -> angle
  ;; Precession at moment tee using 0,0 as J2000 coordinates.
  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
  ;; Willmann-Bell, 2nd edn., 1998, pp. 136-137.
  (let* ((c (julian-centuries tee))
         (eta (mod
               (poly c (list 0 (secs 47.0029L0)
                              (secs -0.03302L0)
                              (secs 0.000060L0)))
               360))
         (cap-P (mod (poly c (list (deg 174.876384L0)
                                   (secs -869.8089L0)
                                   (secs 0.03536L0)))
                     360))
         (p (mod (poly c (list 0 (secs 5029.0966L0)
                               (secs 1.11113L0)
                               (secs 0.000006L0)))
                 360))
         (cap-A (* (cos-degrees eta) (sin-degrees cap-P)))
         (cap-B (cos-degrees cap-P))
         (arg (arctan-degrees cap-A cap-B)))
    (mod (- (+ p cap-P) arg) 360)))
```

(14.40)

```
(defun sidereal-solar-longitude (tee)
  ;; TYPE moment -> angle
  ;; Sidereal solar longitude at moment tee
  (mod (+ (solar-longitude tee)
          (- (precession tee))
          sidereal-start)
       360))
```

(14.41)

```
(defun solar-altitude (tee location)
  ;; TYPE (moment location) -> half-circle
  ;; Geocentric altitude of sun at tee at location,
  ;; as a positive/negative angle in degrees, ignoring
  ;; parallax and refraction.
  (let* ((phi ; Local latitude.
          (latitude location))
         (psi ; Local longitude.
          (longitude location))
         (lambda ; Solar longitude.
           (solar-longitude tee))
         (alpha ; Solar right ascension.
          (right-ascension tee 0 lambda))
         (delta ; Solar declination.
          (declination tee 0 lambda))
         (theta0 ; Sidereal time.
          (sidereal-from-moment tee))
         (cap-H ; Local hour angle.
          (mod (- theta0 (- psi) alpha) 360))
         (altitude
          (arcsin-degrees (+ (* (sin-degrees phi)
                                (sin-degrees delta))
                             (* (cos-degrees phi)
                                (cos-degrees delta)
                                (cos-degrees cap-H))))))
    (mod3 altitude -180 180)))
```

(14.42)

```
(defun estimate-prior-solar-longitude (lambda tee)
  ;; TYPE (season moment) -> moment
  ;; Approximate moment at or before tee
  ;; when solar longitude just exceeded lambda degrees.
  (let* ((rate ; Mean change of one degree.
          (/ mean-tropical-year (deg 360)))
         (tau ; First approximation.
          (- tee
             (* rate (mod (- (solar-longitude tee)
                             lambda)
```

```
                          360))))
         (cap-Delta ; Difference in longitude.
          (mod3 (- (solar-longitude tau) lambda)
                -180 180)))
    (min tee (- tau (* rate cap-Delta)))))
```

```
(defconstant mean-synodic-month                                   (14.44)
  ;; TYPE duration
  29.530588861L0)
```

```
(defun nth-new-moon (n)                                           (14.45)
  ;; TYPE integer -> moment
  ;; Moment of n-th new moon after (or before) the new moon
  ;; of January 11, 1.  Adapted from "Astronomical Algorithms"
  ;; by Jean Meeus, Willmann-Bell, corrected 2nd edn., 2005.
  (let* ((n0 24724) ; Months from RD 0 until j2000.
         (k (- n n0)) ; Months since j2000.
         (c (/ k 1236.85L0)) ; Julian centuries.
         (approx (+ j2000
                    (poly c (list 5.09766L0
                                  (* mean-synodic-month
                                     1236.85L0)
                                  0.00015437L0
                                  -0.000000150L0
                                  0.00000000073L0)))))
         (cap-E (poly c (list 1 -0.002516L0 -0.0000074L0)))
         (solar-anomaly
          (poly c (deg (list 2.5534L0
                             (* 1236.85L0 29.10535670L0)
                             -0.0000014L0 -0.00000011L0))))
         (lunar-anomaly
          (poly c (deg (list 201.5643L0 (* 385.81693528L0
                                           1236.85L0)
                             0.0107582L0 0.00001238L0
                             -0.000000058L0))))
         (moon-argument ; Moon's argument of latitude.
          (poly c (deg (list 160.7108L0 (* 390.67050284L0
                                           1236.85L0)
                             -0.0016118L0 -0.00000227L0
                             0.000000011L0))))
         (cap-omega ; Longitude of ascending node.
          (poly c (deg (list 124.7746L0 (* -1.56375588L0 1236.85L0)
                        0.0020672L0 0.00000215L0))))
         (E-factor (list 0 1 0 0 1 1 2 0 0 1 0 1 1 1 0 0 0 0
                         0 0 0 0 0 0))
         (solar-coeff (list 0 1 0 0 -1 1 2 0 0 1 0 1 1 -1 2
                            0 3 1 0 1 -1 -1 1 0))
         (lunar-coeff (list 1 0 2 0 1 1 0 1 1 2 3 0 0 2 1 2
                            0 1 2 1 1 1 3 4))
         (moon-coeff (list 0 0 0 2 0 0 0 -2 2 0 0 2 -2 0 0
                           -2 0 -2 2 2 2 -2 0 0))
         (sine-coeff
          (list -0.40720L0 0.17241L0 0.01608L0 0.01039L0
                0.00739L0 -0.00514L0 0.00208L0
                -0.00111L0 -0.00057L0 0.00056L0
                -0.00042L0 0.00042L0 0.00038L0
                -0.00024L0 -0.00007L0 0.00004L0
                0.00004L0 0.00003L0 0.00003L0
                -0.00003L0 0.00003L0 -0.00002L0
                -0.00002L0 0.00002L0))
         (correction
          (+ (* -0.00017L0 (sin-degrees cap-omega))
             (sigma ((v sine-coeff)
                     (w E-factor)
                     (x solar-coeff)
                     (y lunar-coeff)
                     (z moon-coeff))
                    (* v (expt cap-E w)
                       (sin-degrees
                        (+ (* x solar-anomaly)
```

```
                         (* y lunar-anomaly)
                         (* z moon-argument)))))))
          (add-const
           (list 251.88L0 251.83L0 349.42L0 84.66L0
                 141.74L0 207.14L0 154.84L0 34.52L0 207.19L0
                 291.34L0 161.72L0 239.56L0 331.55L0))
          (add-coeff
           (list 0.016321L0 26.651886L0
                 36.412478L0 18.206239L0 53.303771L0
                 2.453732L0 7.306860L0 27.261239L0 0.121824L0
                 1.844379L0 24.198154L0 25.513099L0
                 3.592518L0))
          (add-factor
           (list 0.000165L0 0.000164L0 0.000126L0
                 0.000110L0 0.000062L0 0.000060L0 0.000056L0
                 0.000047L0 0.000042L0 0.000040L0 0.000037L0
                 0.000035L0 0.000023L0))
          (extra
           (* 0.000325L0
              (sin-degrees
               (poly c
                     (deg (list 299.77L0 132.8475848L0
                                -0.009173L0))))))
          (additional
           (sigma ((i add-const)
                   (j add-coeff)
                   (l add-factor))
                  (* l (sin-degrees (+ i (* j k)))))))
     (universal-from-dynamical
      (+ approx correction extra additional))))
```

(14.46)

```
(defun new-moon-before (tee)
  ;; TYPE moment -> moment
  ;; Moment UT of last new moon before tee.
  (let* ((t0 (nth-new-moon 0))
         (phi (lunar-phase tee))
         (n (round (- (/ (- tee t0) mean-synodic-month)
                      (/ phi (deg 360))))))
    (nth-new-moon (final k (1- n) (< (nth-new-moon k) tee)))))
```

(14.47)

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

(14.48)

```
(defun lunar-longitude (tee)
  ;; TYPE moment -> angle
  ;; Longitude of moon (in degrees) at moment tee.
  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
  ;; Willmann-Bell, 2nd edn., 1998, pp. 338-342.
  (let* ((c (julian-centuries tee))
         (cap-L-prime (mean-lunar-longitude c))
         (cap-D (lunar-elongation c))
         (cap-M (solar-anomaly c))
         (cap-M-prime (lunar-anomaly c))
         (cap-F (moon-node c))
         (cap-E (poly c (list 1 -0.002516L0 -0.0000074L0)))
         (args-lunar-elongation
          (list 0 2 2 0 0 0 2 2 2 2 0 1 0 2 0 0 4 0 4 2 2 1
                1 2 2 4 2 0 2 2 1 2 0 0 2 2 2 4 0 3 2 4 0 2
                2 2 4 0 4 1 2 0 1 3 4 2 0 1 2))
         (args-solar-anomaly
          (list 0 0 0 0 1 0 0 -1 0 -1 1 0 1 0 0 0 0 0 0 1 1
                0 1 -1 0 0 0 1 0 -1 0 -2 1 2 -2 0 0 -1 0 0 1
                -1 2 2 1 -1 0 0 -1 0 1 0 1 0 0 -1 2 1 0))
```

```
        (args-lunar-anomaly
         (list 1 -1 0 2 0 0 -2 -1 1 0 -1 0 1 0 1 1 -1 3 -2
               -1 0 -1 0 1 2 0 -3 -2 -1 -2 1 0 2 0 -1 1 0
               -1 2 -1 1 -2 -1 -1 -2 0 1 4 0 -2 0 2 1 -2 -3
               2 1 -1 3))
        (args-moon-node
         (list 0 0 0 0 0 2 0 0 0 0 0 0 0 -2 2 -2 0 0 0 0 0
               0 0 0 0 0 0 2 0 0 0 0 0 0 -2 2 0 2 0 0 0 0
               0 0 -2 0 0 0 0 -2 -2 0 0 0 0 0 0 0 0))
        (sine-coeff
         (list 6288774 1274027 658314 213618 -185116 -114332
               58793 57066 53322 45758 -40923 -34720 -30383
               15327 -12528 10980 10675 10034 8548 -7888
               -6766 -5163 4987 4036 3994 3861 3665 -2689
               -2602 2390 -2348 2236 -2120 -2069 2048 -1773
               -1595 1215 -1110 -892 -810 759 -713 -700 691
               596 549 537 520 -487 -399 -381 351 -340 330
               327 -323 299 294))
        (correction
         (* (deg 1/1000000)
            (sigma ((v sine-coeff)
                    (w args-lunar-elongation)
                    (x args-solar-anomaly)
                    (y args-lunar-anomaly)
                    (z args-moon-node))
                   (* v (expt cap-E (abs x))
                      (sin-degrees
                       (+ (* w cap-D)
                          (* x cap-M)
                          (* y cap-M-prime)
                          (* z cap-F)))))))
        (venus (* (deg 3958/1000000)
                  (sin-degrees
                   (+ (deg 119.75L0) (* c (deg 131.849L0))))))
        (jupiter (* (deg 318/1000000)
                    (sin-degrees
                     (+ (deg 53.09L0)
                        (* c (deg 479264.29L0))))))
        (flat-earth
         (* (deg 1962/1000000)
            (sin-degrees (- cap-L-prime cap-F)))))
    (mod (+ cap-L-prime correction venus jupiter flat-earth
            (nutation tee))
         360)))
```

```
(defun mean-lunar-longitude (c)
  ;; TYPE century -> angle
  ;; Mean longitude of moon (in degrees) at moment
  ;; given in Julian centuries c.
  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
  ;; Willmann-Bell, 2nd edn., 1998, pp. 337-340.
  (mod
   (poly c
         (deg (list 218.3164477L0 481267.88123421L0
                    -0.0015786L0 1/538841 -1/65194000)))
   360))
```
(14.49)

```
(defun lunar-elongation (c)
  ;; TYPE century -> angle
  ;; Elongation of moon (in degrees) at moment
  ;; given in Julian centuries c.
  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
  ;; Willmann-Bell, 2nd edn., 1998, p. 338.
  (mod
   (poly c
         (deg (list 297.8501921L0 445267.1114034L0
                    -0.0018819L0 1/545868 -1/113065000)))
   360))
```
(14.50)

```
(defun solar-anomaly (c)
  ;; TYPE century -> angle
  ;; Mean anomaly of sun (in degrees) at moment
```
(14.51)

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

(14.52)

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

(14.53)

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

(14.54)

```
(defun lunar-node (date)
  ;; TYPE fixed-date -> angle
  ;; Angular distance of the lunar node from the equinoctial
  ;; point on fixed date.
  (mod3 (+ (moon-node (julian-centuries date)))
        -90 90))
```

(14.55)

```
(defun sidereal-lunar-longitude (tee)
  ;; TYPE moment -> angle
  ;; Sidereal lunar longitude at moment tee.
  (mod (+ (lunar-longitude tee)
          (- (precession tee))
          sidereal-start)
       360))
```

(14.56)

```
(defun lunar-phase (tee)
  ;; TYPE moment -> phase
  ;; Lunar phase, as an angle in degrees, at moment tee.
  ;; An angle of 0 means a new moon, 90 degrees means the
  ;; first quarter, 180 means a full moon, and 270 degrees
  ;; means the last quarter.
  (let* ((phi (mod (- (lunar-longitude tee)
                      (solar-longitude tee))
                   360))
         (t0 (nth-new-moon 0))
         (n (round (/ (- tee t0) mean-synodic-month)))
         (phi-prime (* (deg 360)
                       (mod (/ (- tee (nth-new-moon n))
                               mean-synodic-month)
                            1))))
    (if (> (abs (- phi phi-prime)) (deg 180)) ; close call
        phi-prime
      phi)))
```

(14.57)
```
(defun lunar-phase-at-or-before (phi tee)
  ;; TYPE (phase moment) -> moment
  ;; Moment UT of the last time at or before tee
  ;; when the lunar-phase was phi degrees.
  (let* ((tau ; Estimate.
          (- tee
             (* mean-synodic-month (/ 1 (deg 360))
                (mod (- (lunar-phase tee) phi) 360))))
         (a (- tau 2))
         (b (min tee (+ tau 2)))) ; At or before tee.
    (invert-angular lunar-phase phi
                    (interval-closed a b))))
```

(14.58)
```
(defun lunar-phase-at-or-after (phi tee)
  ;; TYPE (phase moment) -> moment
  ;; Moment UT of the next time at or after tee
  ;; when the lunar-phase is phi degrees.
  (let* ((tau ; Estimate.
          (+ tee
             (* mean-synodic-month (/ 1 (deg 360))
                (mod (- phi (lunar-phase tee)) 360))))
         (a (max tee (- tau 2))) ; At or after tee.
         (b (+ tau 2)))
    (invert-angular lunar-phase phi
                    (interval-closed a b))))
```

(14.59)
```
(defconstant new
  ;; TYPE phase
  ;; Excess of lunar longitude over solar longitude at new
  ;; moon.
  (deg 0))
```

(14.61)
```
(defconstant full
  ;; TYPE phase
  ;; Excess of lunar longitude over solar longitude at full
  ;; moon.
  (deg 180))
```

(14.60)
```
(defconstant first-quarter
  ;; TYPE phase
  ;; Excess of lunar longitude over solar longitude at first
  ;; quarter moon.
  (deg 90))
```

(14.62)
```
(defconstant last-quarter
  ;; TYPE phase
  ;; Excess of lunar longitude over solar longitude at last
  ;; quarter moon.
  (deg 270))
```

(14.63)
```
(defun lunar-latitude (tee)
  ;; TYPE moment -> half-circle
  ;; Latitude of moon (in degrees) at moment tee.
  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
  ;; Willmann-Bell, 2nd edn., 1998, pp. 338-342.
  (let* ((c (julian-centuries tee))
         (cap-L-prime (mean-lunar-longitude c))
         (cap-D (lunar-elongation c))
         (cap-M (solar-anomaly c))
         (cap-M-prime (lunar-anomaly c))
         (cap-F (moon-node c))
         (cap-E (poly c (list 1 -0.002516L0 -0.0000074L0)))
         (args-lunar-elongation
          (list 0 0 0 2 2 2 2 0 2 0 2 2 2 2 2 2 2 0 4 0 0 0
                1 0 0 0 1 0 4 4 0 4 2 2 2 2 0 2 2 2 2 4 2 2
                0 2 1 1 0 2 1 2 0 4 4 1 4 1 4 2))
         (args-solar-anomaly
          (list 0 0 0 0 0 0 0 0 0 0 -1 0 0 1 -1 -1 -1 1 0 1
```

```
              0 1 0 1 1 1 0 0 0 0 0 0 0 0 -1 0 0 0 0 1 1
              0 -1 -2 0 1 1 1 1 1 0 -1 1 0 -1 0 0 0 -1 -2))
       (args-lunar-anomaly
        (list 0 1 1 0 -1 -1 0 2 1 2 0 -2 1 0 -1 0 -1 -1 -1
              0 0 -1 0 1 1 0 0 3 0 -1 1 -2 0 2 1 -2 3 2 -3
              -1 0 0 1 0 1 1 0 0 -2 -1 1 -2 2 -2 -1 1 1 -1
              0 0))
       (args-moon-node
        (list 1 1 -1 -1 1 -1 1 1 -1 -1 -1 -1 1 -1 1 1 -1 -1
              -1 1 3 1 1 1 -1 -1 -1 1 -1 1 -3 1 -3 -1 -1 1
              -1 1 -1 1 1 1 1 -1 3 -1 -1 1 -1 -1 1 -1 1 -1
              -1 -1 -1 -1 -1 1))
       (sine-coeff
        (list 5128122 280602 277693 173237 55413 46271 32573
              17198 9266 8822 8216 4324 4200 -3359 2463 2211
              2065 -1870 1828 -1794 -1749 -1565 -1491 -1475
              -1410 -1344 -1335 1107 1021 833 777 671 607
              596 491 -451 439 422 421 -366 -351 331 315
              302 -283 -229 223 223 -220 -220 -185 181
              -177 176 166 -164 132 -119 115 107))
       (beta
        (* (deg 1/1000000)
           (sigma ((v sine-coeff)
                   (w args-lunar-elongation)
                   (x args-solar-anomaly)
                   (y args-lunar-anomaly)
                   (z args-moon-node))
                  (* v (expt cap-E (abs x))
                     (sin-degrees
                      (+ (* w cap-D)
                         (* x cap-M)
                         (* y cap-M-prime)
                         (* z cap-F)))))))
       (venus (* (deg 175/1000000)
                 (+ (sin-degrees
                     (+ (deg 119.75L0) (* c (deg 131.849L0))
                        cap-F))
                    (sin-degrees
                     (+ (deg 119.75L0) (* c (deg 131.849L0))
                        (- cap-F))))))
       (flat-earth
        (+ (* (deg -2235/1000000)
              (sin-degrees cap-L-prime))
           (* (deg 127/1000000) (sin-degrees
                                 (- cap-L-prime cap-M-prime)))
           (* (deg -115/1000000) (sin-degrees
                                  (+ cap-L-prime cap-M-prime)))))
       (extra (* (deg 382/1000000)
                 (sin-degrees
                  (+ (deg 313.45L0)
                     (* c (deg 481266.484L0)))))))
    (+ beta venus flat-earth extra)))
```

```
(defun lunar-altitude (tee location)                              (14.64)
  ;; TYPE (moment location) -> half-circle
  ;; Geocentric altitude of moon at tee at location,
  ;; as a small positive/negative angle in degrees, ignoring
  ;; parallax and refraction.  Adapted from "Astronomical
  ;; Algorithms" by Jean Meeus, Willmann-Bell, 2nd edn.,
  ;; 1998.
  (let* ((phi ; Local latitude.
          (latitude location))
         (psi ; Local longitude.
          (longitude location))
         (lambda ; Lunar longitude.
          (lunar-longitude tee))
         (beta ; Lunar latitude.
          (lunar-latitude tee))
         (alpha ; Lunar right ascension.
          (right-ascension tee beta lambda))
         (delta ; Lunar declination.
```

```
          (declination tee beta lambda))
         (theta0 ; Sidereal time.
          (sidereal-from-moment tee))
         (cap-H ; Local hour angle.
          (mod (- theta0 (- psi) alpha) 360))
         (altitude
          (arcsin-degrees (+ (* (sin-degrees phi)
                                (sin-degrees delta))
                             (* (cos-degrees phi)
                                (cos-degrees delta)
                                (cos-degrees cap-H))))))
    (mod3 altitude -180 180)))
```

(14.65)

```
(defun lunar-distance (tee)
  ;; TYPE moment -> distance
  ;; Distance to moon (in meters) at moment tee.
  ;; Adapted from "Astronomical Algorithms" by Jean Meeus,
  ;; Willmann-Bell, 2nd edn., 1998, pp. 338-342.
  (let* ((c (julian-centuries tee))
         (cap-D (lunar-elongation c))
         (cap-M (solar-anomaly c))
         (cap-M-prime (lunar-anomaly c))
         (cap-F (moon-node c))
         (cap-E (poly c (list 1 -0.002516L0 -0.0000074L0)))
         (args-lunar-elongation
          (list 0 2 2 0 0 0 2 2 2 2 0 1 0 2 0 0 4 0 4 2 2 1
                1 2 2 4 2 0 2 2 1 2 0 0 2 2 2 4 0 3 2 4 0 2
                2 2 4 0 4 1 2 0 1 3 4 2 0 1 2 2))
         (args-solar-anomaly
          (list 0 0 0 0 1 0 0 -1 0 -1 1 0 1 0 0 0 0 0 0 1 1
                0 1 -1 0 0 0 1 0 -1 0 -2 1 2 -2 0 0 -1 0 0 1
                -1 2 2 1 -1 0 0 -1 0 1 0 1 0 0 -1 2 1 0 0))
         (args-lunar-anomaly
          (list 1 -1 0 2 0 0 -2 -1 1 0 -1 0 1 0 1 1 -1 3 -2
                -1 0 -1 0 1 2 0 -3 -2 -1 -2 1 0 2 0 -1 1 0
                -1 2 -1 1 -2 -1 -1 -2 0 1 4 0 -2 0 2 1 -2 -3
                2 1 -1 3 -1))
         (args-moon-node
          (list 0 0 0 0 0 2 0 0 0 0 0 0 0 -2 2 -2 0 0 0 0 0
                0 0 0 0 0 0 0 2 0 0 0 0 0 0 -2 2 0 2 0 0 0 0
                0 0 -2 0 0 0 0 -2 -2 0 0 0 0 0 0 0 -2))
         (cosine-coeff
          (list -20905355 -3699111 -2955968 -569925 48888 -3149
                246158 -152138 -170733 -204586 -129620 108743
                104755 10321 0 79661 -34782 -23210 -21636 24208
                30824 -8379 -16675 -12831 -10445 -11650 14403
                -7003 0 10056 6322 -9884 5751 0 -4950 4130 0
                -3958 0 3258 2616 -1897 -2117 2354 0 0 -1423
                -1117 -1571 -1739 0 -4421 0 0 0 0 1165 0 0
                8752))
         (correction
          (sigma ((v cosine-coeff)
                  (w args-lunar-elongation)
                  (x args-solar-anomaly)
                  (y args-lunar-anomaly)
                  (z args-moon-node))
                 (* v (expt cap-E (abs x))
                    (cos-degrees
                     (+ (* w cap-D)
                        (* x cap-M)
                        (* y cap-M-prime)
                        (* z cap-F)))))))
    (+ (mt 385000560) correction)))
```

(14.66)

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

```
          (cap-Delta (lunar-distance tee))
          (alt (/ (mt 6378140) cap-Delta))
          (arg (* alt (cos-degrees geo))))
     (arcsin-degrees arg)))
```

(14.67)

```
(defun topocentric-lunar-altitude (tee location)
  ;; TYPE (moment location) -> half-circle
  ;; Topocentric altitude of moon at tee at location,
  ;; as a small positive/negative angle in degrees,
  ;; ignoring refraction.
  (- (lunar-altitude tee location)
     (lunar-parallax tee location)))
```

Times of day are computed by the following functions:

(14.68)

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

(14.69)

```
(defun sine-offset (tee location alpha)
  ;; TYPE (moment location half-circle) -> real
  ;; Sine of angle between position of sun at
  ;; local time tee and
  ;; when its depression is alpha at location.
  ;; Out of range when it does not occur.
  (let* ((phi (latitude location))
         (tee-prime (universal-from-local tee location))
         (delta ; Declination of sun.
          (declination tee-prime (deg 0L0)
                       (solar-longitude tee-prime))))
    (+ (* (tan-degrees phi)
          (tan-degrees delta))
       (/ (sin-degrees alpha)
          (* (cos-degrees delta)
             (cos-degrees phi))))))
```

(14.70)

```
(defun moment-of-depression (approx location alpha early?)
  ;; TYPE (moment location half-circle boolean) -> moment
  ;; Moment in local time near approx when depression
  ;; angle of sun is alpha (negative if above horizon) at
  ;; location; early? is true when morning event is
  ;; sought, and false for evening.
  ;; Returns bogus if depression angle is not reached.
  (let* ((tee (approx-moment-of-depression
               approx location alpha early?)))
    (if (equal tee bogus)
        bogus
```

```
      (if (< (abs (- approx tee))
             (sec 30))
          tee
        (moment-of-depression tee location alpha early?)))))
```

(14.71)

```
(defconstant morning
  ;; TYPE boolean
  ;; Signifies morning.
  true)
```

(14.72)

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

(14.73)

```
(defconstant evening
  ;; TYPE boolean
  ;; Signifies evening.
  false)
```

(14.74)

```
(defun dusk (date location alpha)
  ;; TYPE (fixed-date location half-circle) -> moment
  ;; Standard time in evening on fixed date at
  ;; location when depression angle of sun is alpha.
  ;; Returns bogus if there is no dusk on date.
  (let* ((result (moment-of-depression
                  (+ date (hr 18)) location alpha evening)))
    (if (equal result bogus)
        bogus
      (standard-from-local result location))))
```

(14.75)

```
(defun refraction (tee location)
  ;; TYPE (moment location) -> half-circle
  ;; Refraction angle at moment tee at location.
  ;; The moment is not used.
  (let* ((h (max (mt 0) (elevation location)))
         (cap-R (mt 6.372d6)) ; Radius of Earth.
         (dip ; Depression of visible horizon.
          (arccos-degrees (/ cap-R (+ cap-R h)))))
    (+ (mins 34) dip
       (* (secs 19) (sqrt h)))))
```

(14.76)

```
(defun sunrise (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of sunrise on fixed date at
  ;; location.
  (let* ((alpha (+ (refraction (+ date (hr 6)) location)
                   (mins 16))))
    (dawn date location alpha)))
```

(14.77)

```
(defun sunset (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of sunset on fixed date at
  ;; location.
  (let* ((alpha (+ (refraction (+ date (hr 18)) location)
                   (mins 16))))
    (dusk date location alpha)))
```

(14.80)
```
(defun jewish-sabbath-ends (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of end of Jewish sabbath on fixed date
  ;; at location (as per Berthold Cohn).
  (dusk date location (angle 7 5 0)))
```

(14.81)
```
(defun jewish-dusk (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of Jewish dusk on fixed date
  ;; at location (as per Vilna Gaon).
  (dusk date location (angle 4 40 0)))
```

(14.82)
```
(defun observed-lunar-altitude (tee location)
  ;; TYPE (moment location) -> half-circle
  ;; Observed altitude of upper limb of moon at tee at location,
  ;; as a small positive/negative angle in degrees, including
  ;; refraction and elevation.
  (+ (topocentric-lunar-altitude tee location)
     (refraction tee location)
     (mins 16)))
```

(14.83)
```
(defun moonrise (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of moonrise on fixed date at location.
  ;; Returns bogus if there is no moonrise on date.
  (let* ((tee ; Midnight.
          (universal-from-standard date location))
         (waning (> (lunar-phase tee) (deg 180)))
         (alt ; Altitude at midnight.
          (observed-lunar-altitude tee location))
         (lat (latitude location))
         (offset (/ alt (* 4 (- (deg 90) (abs lat)))))
         (approx ; Approximate rising time.
          (if waning
              (if (> offset 0)
                  (- tee -1 offset)
                (- tee offset))
            (+ tee 1/2 offset)))
         (rise (binary-search
                l (- approx (hr 6))
                u (+ approx (hr 6))
                x (> (observed-lunar-altitude x location)
                     (deg 0))
                (< (- u l) (mn 1)))))
    (if (< rise (1+ tee))
        (max (standard-from-universal rise location)
             date) ; May be just before to midnight.
      ;; Else no moonrise this day.
      bogus)))
```

(14.84)
```
(defun moonset (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of moonset on fixed date at location.
  ;; Returns bogus if there is no moonset on date.
  (let* ((tee ; Midnight.
          (universal-from-standard date location))
         (waxing (< (lunar-phase tee) (deg 180)))
         (alt ; Altitude at midnight.
          (observed-lunar-altitude tee location))
         (lat (latitude location))
         (offset (/ alt (* 4 (- (deg 90) (abs lat)))))
         (approx ; Approximate setting time.
          (if waxing
              (if (> offset 0)
                  (+ tee offset)
                (+ tee 1 offset))
            (- tee offset -1/2)))
         (set (binary-search
```

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

(14.85)

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

(14.86)

```
(defun local-zero-hour (tee)
  ;; TYPE moment -> moment
  ;; Local time of dusk in Padua, Italy on date of moment tee.
  (let* ((date (fixed-from-moment tee)))
    (local-from-standard
     (+ (dusk date padua (angle 0 16 0)) ; Sunset.
        (mn 30)) ; Dusk.
     padua)))
```

(14.87)

```
(defun local-from-italian (tee)
  ;; TYPE moment -> moment
  ;; Local time corresponding to Italian time tee.
  (let* ((date (fixed-from-moment tee))
         (z (local-zero-hour (1- tee))))
    (- tee (- date z))))
```

(14.88)

```
(defun italian-from-local (tee_ell)
  ;; TYPE moment -> moment
  ;; Italian time corresponding to local time tee_ell.
  (let* ((date (fixed-from-moment tee_ell))
         (z0 (local-zero-hour (1- tee_ell)))
         (z (local-zero-hour tee_ell)))
    (if (> tee_ell z) ; if after zero hour
        (+ tee_ell (- date -1 z)) ; then next day
      (+ tee_ell (- date z0)))))
```

(14.89)

```
(defun daytime-temporal-hour (date location)
  ;; TYPE (fixed-date location) -> real
  ;; Length of daytime temporal hour on fixed date at location.
  ;; Returns bogus if there no sunrise or sunset on date.
  (if (or (equal (sunrise date location) bogus)
          (equal (sunset date location) bogus))
      bogus
    (/ (- (sunset date location)
          (sunrise date location))
       12)))
```

(14.90)

```
(defun nighttime-temporal-hour (date location)
  ;; TYPE (fixed-date location) -> real
  ;; Length of nighttime temporal hour on fixed date at location.
  ;; Returns bogus if there no sunrise or sunset on date.
  (if (or (equal (sunrise (1+ date) location) bogus)
          (equal (sunset date location) bogus))
      bogus
    (/ (- (sunrise (1+ date) location)
          (sunset date location))
       12)))
```

(14.91)

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

```
  (let* ((date (fixed-from-moment tee))
         (hour (* 24 (time-from-moment tee)))
         (h (cond ((<= 6 hour 18); daytime today
                   (daytime-temporal-hour date location))
                  ((< hour 6)    ; early this morning
                   (nighttime-temporal-hour (1- date) location))
                  (t             ; this evening
                   (nighttime-temporal-hour date location)))))
    (cond ((equal h bogus) bogus)
          ((<= 6 hour 18); daytime today
           (+ (sunrise date location) (* (- hour 6) h)))
          ((< hour 6)    ; early this morning
           (+ (sunset (1- date) location) (* (+ hour 6) h)))
          (t             ; this evening
           (+ (sunset date location) (* (- hour 18) h))))))
```

(14.92)

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

(14.93)

```
(defun asr (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of asr on fixed date at location.
  ;; According to Hanafi rule.
  ;; Returns bogus is no asr occurs.
  (let* ((noon ; Time when sun nearest zenith.
           (midday date location))
         (phi (latitude location))
         (delta ; Solar declination at noon.
          (declination noon (deg 0) (solar-longitude noon)))
         (altitude ; Solar altitude at noon.
          (arcsin-degrees
           (+ (* (cos-degrees delta) (cos-degrees phi))
              (* (sin-degrees delta) (sin-degrees phi)))))
         (h ; Sun's altitude when shadow increases by
          (mod3 (arctan-degrees ; ... double its length.
                 (tan-degrees altitude)
                 (1+ (* 2 (tan-degrees altitude))))
                -90 90)))
    (if (<= altitude (deg 0)) ; No shadow.
        bogus
      (dusk date location (- h)))))
```

(14.94)

```
(defun alt-asr (date location)
  ;; TYPE (fixed-date location) -> moment
  ;; Standard time of asr on fixed date at location.
  ;; According to Shafi'i rule.
  ;; Returns bogus is no asr occurs.
  (let* ((noon ; Time when sun nearest zenith.
           (midday date location))
         (phi (latitude location))
         (delta ; Solar declination at noon.
          (declination noon (deg 0) (solar-longitude noon)))
         (altitude ; Solar altitude at noon.
          (arcsin-degrees
           (+ (* (cos-degrees delta) (cos-degrees phi))
              (* (sin-degrees delta) (sin-degrees phi)))))
         (h ; Sun's altitude when shadow increases by
          (mod3 (arctan-degrees ; ... its length.
                 (tan-degrees altitude)
                 (1+ (tan-degrees altitude)))
                -90 90)))
    (if (<= altitude (deg 0)) ; No shadow.
        bogus
      (dusk date location (- h)))))
```

The functions for lunar visibility are:

(14.95)

```
(defun arc-of-light (tee)
  ;; TYPE moment -> half-circle
  ;; Angular separation of sun and moon
  ;; at moment tee.
  (arccos-degrees
   (* (cos-degrees (lunar-latitude tee))
      (cos-degrees (lunar-phase tee)))))
```

(14.96)

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

(14.97)

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

(14.98)

```
(defun arc-of-vision (tee location)
  ;; TYPE (moment location) -> half-circle
  ;; Angular difference in altitudes of sun and moon
  ;; at moment tee at location.
  (- (lunar-altitude tee location)
     (solar-altitude tee location)))
```

(14.99)

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

(14.100)

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

(14.101)
```
(defun lunar-semi-diameter (tee location)
  ;; TYPE (moment location) -> half-circle
  ;; Topocentric lunar semi-diameter at moment tee and location.
  (let* ((h (lunar-altitude tee location))
         (p (lunar-parallax tee location)))
    (* 0.27245L0 p (1+ (* (sin-degrees h) (sin-degrees p))))))
```

(14.102)
```
(defun lunar-diameter (tee)
  ;; TYPE moment -> angle
  ;; Geocentric apparent lunar diameter of the moon (in
  ;; degrees) at moment tee.  Adapted from "Astronomical
  ;; Algorithms" by Jean Meeus, Willmann-Bell, 2nd edn.,
  ;; 1998.
  (/ (deg 1792367000/9) (lunar-distance tee)))
```

(14.103)
```
(defun visible-crescent (date location)
  ;; TYPE (fixed-date location) -> boolean
  ;; Criterion for possible visibility of crescent moon
  ;; on eve of date at location.
  ;; Shaukat's criterion may be replaced with another.
  (shaukat-criterion date location))
```

(14.104)
```
(defun phasis-on-or-before (date location)
  ;; TYPE (fixed-date location) -> fixed-date
  ;; Closest fixed date on or before date when crescent
  ;; moon first became visible at location.
  (let* ((moon ; Prior new moon.
          (fixed-from-moment
           (lunar-phase-at-or-before new date)))
         (age (- date moon))
         (tau ; Check if not visible yet on eve of date.
          (if (and (<= age 3)
                   (not (visible-crescent date location)))
              (- moon 30) ; Must go back a month.
            moon)))
    (next d tau (visible-crescent d location))))
```

(14.105)
```
(defun phasis-on-or-after (date location)
  ;; TYPE (fixed-date location) -> fixed-date
  ;; Closest fixed date on or after date on the eve
  ;; of which crescent moon first became visible at location.
  (let* ((moon ; Prior new moon.
          (fixed-from-moment
           (lunar-phase-at-or-before new date)))
         (age (- date moon))
         (tau ; Check if not visible yet on eve of date.
          (if (or (<= 4 age)
                  (visible-crescent (1- date) location))
              (+ moon 29) ; Next new moon
            date)))
    (next d tau (visible-crescent d location))))
```

D.15 The Persian Calendar

```
(defun persian-date (year month day)
  ;; TYPE (persian-year persian-month persian-day)
  ;; TYPE  -> persian-date
  (list year month day))
```

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

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

(15.3)

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

(15.4)

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

(15.5)

```
(defun fixed-from-persian (p-date)
  ;; TYPE persian-date -> fixed-date
  ;; Fixed date of Astronomical Persian date p-date.
  (let* ((month (standard-month p-date))
         (day (standard-day p-date))
         (year (standard-year p-date))
         (new-year
          (persian-new-year-on-or-before
           (+ persian-epoch 180; Fall after epoch.
              (floor
               (* mean-tropical-year
                  (if (< 0 year)
                      (1- year)
                    year))))))); No year zero.
    (+ (1- new-year)      ; Days in prior years.
       (if (<= month 7)   ; Days in prior months this year.
           (* 31 (1- month))
         (+ (* 30 (1- month)) 6))
       day)))             ; Days so far this month.
```

(15.6)

```
(defun persian-from-fixed (date)
  ;; TYPE fixed-date -> persian-date
  ;; Astronomical Persian date (year month day)
  ;; corresponding to fixed date.
  (let* ((new-year
          (persian-new-year-on-or-before date))
         (y (1+ (round (/ (- new-year persian-epoch)
                          mean-tropical-year))))
         (year (if (< 0 y)
                   y
                 (1- y))); No year zero
         (day-of-year (1+ (- date
                             (fixed-from-persian
                              (persian-date year 1 1)))))
         (month (if (<= day-of-year 186)
                    (ceiling (/ day-of-year 31))
                  (ceiling (/ (- day-of-year 6) 30))))
         (day           ; Calculate the day by subtraction
          (- date (1- (fixed-from-persian
                       (persian-date year month 1))))))
    (persian-date year month day)))
```

(15.7)

```
(defun arithmetic-persian-leap-year? (p-year)
  ;; TYPE persian-year -> boolean
  ;; True if p-year is a leap year on the Persian calendar.
  (let* ((y ; Years since start of 2820-year cycles
          (if (< 0 p-year)
              (- p-year 474)
            (- p-year 473))); No year zero
         (year ; Equivalent year in the range 474..3263
          (+ (mod y 2820) 474)))
    (< (mod (* (+ year 38)
               31)
            128)
       31)))
```

(15.8)

```
(defun fixed-from-arithmetic-persian (p-date)
  ;; TYPE persian-date -> fixed-date
  ;; Fixed date equivalent to Persian date p-date.
  (let* ((day (standard-day p-date))
         (month (standard-month p-date))
         (p-year (standard-year p-date))
         (y ; Years since start of 2820-year cycle
          (if (< 0 p-year)
              (- p-year 474)
            (- p-year 473))); No year zero
         (year ; Equivalent year in the range 474..3263
          (+ (mod y 2820) 474)))
    (+ (1- persian-epoch); Days before epoch
       (* 1029983        ; Days in 2820-year cycles
                                        ; before Persian year 474
          (quotient y 2820))
       (* 365 (1- year)) ; Nonleap days in prior years this
                                        ; 2820-year cycle
       (quotient         ; Leap days in prior years this
                                        ; 2820-year cycle
        (- (* 31 year) 5) 128)
       (if (<= month 7)  ; Days in prior months this year
           (* 31 (1- month))
         (+ (* 30 (1- month)) 6))
       day)))            ; Days so far this month
```

(15.9)

```
(defun arithmetic-persian-year-from-fixed (date)
  ;; TYPE fixed-date -> persian-year
  ;; Persian year corresponding to the fixed date.
  (let* ((d0      ; Prior days since start of 2820-year cycle
                  ; beginning in Persian year 474
          (- date (fixed-from-arithmetic-persian
                   (persian-date 475 1 1))))
         (n2820   ; Completed prior 2820-year cycles
          (quotient d0 1029983))
         (d1      ; Prior days not in n2820--that is, days
                  ; since start of last 2820-year cycle
          (mod d0 1029983))
         (y2820 ; Years since start of last 2820-year cycle
          (if (= d1 1029982)
              ;; Last day of 2820-year cycle
              2820
            ;; Otherwise use cycle of years formula
            (quotient (+ (* 128 d1) 46878)
                      46751)))
         (year    ; Years since Persian epoch
          (+ 474     ; Years before start of 2820-year cycles
             (* 2820 n2820) ; Years in prior 2820-year cycles
             y2820))); Years since start of last 2820-year
                                        ; cycle
    (if (< 0 year)
        year
      (1- year)))); No year zero
```

(15.10)

```
(defun arithmetic-persian-from-fixed (date)
  ;; TYPE fixed-date -> persian-date
  ;; Persian date corresponding to fixed date.
  (let* ((year (arithmetic-persian-year-from-fixed date))
         (day-of-year (1+ (- date
                             (fixed-from-arithmetic-persian
                              (persian-date year 1 1)))))
         (month (if (<= day-of-year 186)
                    (ceiling (/ day-of-year 31))
                  (ceiling (/ (- day-of-year 6) 30))))
         (day           ; Calculate the day by subtraction
          (- date (1- (fixed-from-arithmetic-persian
                       (persian-date year month 1))))))
    (persian-date year month day)))
```

(15.11)

```
(defun nowruz (g-year)
  ;; TYPE gregorian-year -> fixed-date
```

```
  ;; Fixed date of Persian New Year (Nowruz) in Gregorian
  ;; year g-year.
  (let* ((persian-year
          (1+ (- g-year
                 (gregorian-year-from-fixed
                  persian-epoch))))
         (y (if (<= persian-year 0)
                ;; No Persian year 0
                (1- persian-year)
              persian-year)))
    (fixed-from-persian (persian-date y 1 1))))
```

D.16 The Bahá’í Calendar

```
(defun bahai-date (major cycle year month day)
  ;; TYPE (bahai-major bahai-cycle bahai-year
  ;; TYPE  bahai-month bahai-day) -> bahai-date
  (list major cycle year month day))
```

```
(defun bahai-major (date)
  ;; TYPE bahai-date -> bahai-major
  (first date))
```

```
(defun bahai-cycle (date)
  ;; TYPE bahai-date -> bahai-cycle
  (second date))
```

```
(defun bahai-year (date)
  ;; TYPE bahai-date -> bahai-year
  (third date))
```

```
(defun bahai-month (date)
  ;; TYPE bahai-date -> bahai-month
  (fourth date))
```

```
(defun bahai-day (date)
  ;; TYPE bahai-date -> bahai-day
  (fifth date))
```

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

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

```
(defun fixed-from-bahai (b-date)                             (16.3)
  ;; TYPE bahai-date -> fixed-date
  ;; Fixed date equivalent to the Baha'i date b-date.
  (let* ((major (bahai-major b-date))
         (cycle (bahai-cycle b-date))
         (year (bahai-year b-date))
         (month (bahai-month b-date))
         (day (bahai-day b-date))
         (g-year; Corresponding Gregorian year.
          (+ (* 361 (1- major))
             (* 19 (1- cycle)) year -1
             (gregorian-year-from-fixed bahai-epoch))))
    (+ (fixed-from-gregorian ; Prior years.
        (gregorian-date g-year march 20))
```

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

(16.4)

```
(defun bahai-from-fixed (date)
  ;; TYPE fixed-date -> bahai-date
  ;; Baha'i (major cycle year month day) corresponding to fixed
  ;; date.
  (let* ((g-year (gregorian-year-from-fixed date))
         (start   ; 1844
          (gregorian-year-from-fixed bahai-epoch))
         (years ; Since start of Baha'i calendar.
          (- g-year start
             (if (<= date
                     (fixed-from-gregorian
                      (gregorian-date g-year march 20)))
                 1 0)))
         (major (1+ (quotient years 361)))
         (cycle (1+ (quotient (mod years 361) 19)))
         (year (1+ (mod years 19)))
         (days; Since start of year
          (- date (fixed-from-bahai
                   (bahai-date major cycle year 1 1))))
         (month
          (cond ((>= date
                     (fixed-from-bahai
                      (bahai-date major cycle year 19 1)))
                 19) ; Last month of year.
                ((>= date ; Intercalary days.
                     (fixed-from-bahai
                      (bahai-date major cycle year
                                  ayyam-i-ha 1)))
                 ayyam-i-ha) ; Intercalary period.
                (t (1+ (quotient days 19)))))
         (day (- date -1
                 (fixed-from-bahai
                  (bahai-date major cycle year month 1)))))
    (bahai-date major cycle year month day)))
```

(16.5)

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

(16.6)

```
(defun bahai-sunset (date)
  ;; TYPE fixed-date -> moment
  ;; Universal time of sunset on fixed date
  ;; in Bahai-Location.
  (universal-from-standard
   (sunset date bahai-location)
   bahai-location))
```

(16.7)

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

(16.8)

```
(defun fixed-from-astro-bahai (b-date)
  ;; TYPE bahai-date -> fixed-date
  ;; Fixed date of Baha'i date b-date.
  (let* ((major (bahai-major b-date))
         (cycle (bahai-cycle b-date))
         (year (bahai-year b-date))
         (month (bahai-month b-date))
         (day (bahai-day b-date))
         (years; Years from epoch
          (+ (* 361 (1- major))
             (* 19 (1- cycle))
             year)))
    (cond ((= month 19); last month of year
           (+ (astro-bahai-new-year-on-or-before
               (+ bahai-epoch
                  (floor (* mean-tropical-year
                            (+ years 1/2)))))
              -20 day))
          ((= month ayyam-i-ha)
           ;; intercalary month, between 18th & 19th
           (+ (astro-bahai-new-year-on-or-before
               (+ bahai-epoch
                  (floor (* mean-tropical-year
                            (- years 1/2)))))
              341 day))
          (t (+ (astro-bahai-new-year-on-or-before
                 (+ bahai-epoch
                    (floor (* mean-tropical-year
                              (- years 1/2)))))
                (* (1- month) 19)
                day -1)))))
```

(16.9)

```
(defun astro-bahai-from-fixed (date)
  ;; TYPE fixed-date -> bahai-date
  ;; Astronomical Baha'i date corresponding to fixed date.
  (let* ((new-year (astro-bahai-new-year-on-or-before date))
         (years (round (/ (- new-year bahai-epoch)
                          mean-tropical-year)))
         (major (1+ (quotient years 361)))
         (cycle (1+ (quotient (mod years 361) 19)))
         (year (1+ (mod years 19)))
         (days; Since start of year
          (- date new-year))
         (month
          (cond
           ((>= date (fixed-from-astro-bahai
                      (bahai-date major cycle year 19 1)))
                                  ; last month of year
            19)
           ((>= date
                (fixed-from-astro-bahai
                 (bahai-date major cycle year ayyam-i-ha 1)))
                                  ; intercalary month
            ayyam-i-ha)
           (t (1+ (quotient days 19)))))
         (day (- date -1
                 (fixed-from-astro-bahai
                  (bahai-date major cycle year month 1)))))
    (bahai-date major cycle year month day)))
```

(16.10)

```
(defun bahai-new-year (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Baha'i New Year in Gregorian year g-year.
  (fixed-from-gregorian
   (gregorian-date g-year march 21)))
```

(16.11)

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

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

(16.12)
```
(defun feast-of-ridvan (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Feast of Ridvan in Gregorian year g-year.
  (+ (naw-ruz g-year) 31))
```

(16.13)
```
(defun birth-of-the-bab (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of the Birthday of the Bab
  ;; in Gregorian year g-year.
  (let* ((ny ; Beginning of Baha'i year.
          (naw-ruz g-year))
         (set1 (bahai-sunset ny))
         (m1 (new-moon-at-or-after set1))
         (m8 (new-moon-at-or-after (+ m1 190)))
         (day (fixed-from-moment m8))
         (set8 (bahai-sunset day)))
    (if (< m8 set8)
        (1+ day)
      (+ day 2))))
```

D.17 The French Revolutionary Calendar

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

(17.1)
```
(defconstant paris
  ;; TYPE location
  ;; Location of Paris Observatory.  Longitude corresponds
  ;; to difference of 9m 21s between Paris time zone and
  ;; Universal Time.
  (location (angle 48 50 11) (angle 2 20 15) (mt 27) (hr 1)))
```

(17.2)
```
(defun midnight-in-paris (date)
  ;; TYPE fixed-date -> moment
  ;; Universal time of true midnight at end of fixed date
  ;; in Paris.
  (midnight (+ date 1) paris))
```

(17.3)
```
(defun french-new-year-on-or-before (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Fixed date of French Revolutionary New Year on or
  ;; before fixed date.
  (let* ((approx ; Approximate time of solstice.
          (estimate-prior-solar-longitude
           autumn (midnight-in-paris date))))
    (next day (- (floor approx) 1)
          (<= autumn (solar-longitude
                      (midnight-in-paris day))))))
```

(17.4)
```
(defconstant french-epoch
  ;; TYPE fixed-date
  ;; Fixed date of start of the French Revolutionary
  ;; calendar.
  (fixed-from-gregorian (gregorian-date 1792 september 22)))
```

(17.5)
```
(defun fixed-from-french (f-date)
  ;; TYPE french-date -> fixed-date
  ;; Fixed date of French Revolutionary date.
  (let* ((month (standard-month f-date))
         (day (standard-day f-date))
         (year (standard-year f-date))
         (new-year
```

```
          (french-new-year-on-or-before
           (floor (+ french-epoch 180; Spring after epoch.
                     (* mean-tropical-year
                        (1- year)))))))
     (+ new-year -1      ;  Days in prior years
        (* 30 (1- month)); Days in prior months
        day)))           ;  Days this month
```

```
(defun french-from-fixed (date)
  ;; TYPE fixed-date -> french-date
  ;; French Revolutionary date of fixed date.
  (let* ((new-year
          (french-new-year-on-or-before date))
         (year (1+ (round (/ (- new-year french-epoch)
                             mean-tropical-year))))
         (month (1+ (quotient (- date new-year) 30)))
         (day (1+ (mod (- date new-year) 30))))
    (french-date year month day)))
```
(17.6)

```
(defun french-leap-year? (f-year)
  ;; TYPE french-year -> boolean
  ;; True if f-year is a leap year on the
  ;; French Revolutionary calendar.
  (> (- (fixed-from-french
         (french-date (1+ f-year) 1 1))
        (fixed-from-french
         (french-date f-year 1 1)))
     365))
```
(17.7)

```
(defun arithmetic-french-leap-year? (f-year)
  ;; TYPE french-year -> boolean
  ;; True if f-year is a leap year on the
  ;; Arithmetic French Revolutionary calendar.
  (and (= (mod f-year 4) 0)
       (not (member (mod f-year 400) (list 100 200 300)))
       (not (= (mod f-year 4000) 0))))
```
(17.8)

```
(defun fixed-from-arithmetic-french (f-date)
  ;; TYPE french-date -> fixed-date
  ;; Fixed date of Arithmetic French Revolutionary
  ;; date f-date.
  (let* ((month (standard-month f-date))
         (day (standard-day f-date))
         (year (standard-year f-date)))
    (+ french-epoch -1; Days before start of calendar.
       (* 365 (1- year)); Ordinary days in prior years.
                        ; Leap days in prior years.
       (quotient (1- year) 4)
       (- (quotient (1- year) 100))
       (quotient (1- year) 400)
       (- (quotient (1- year) 4000))
       (* 30 (1- month)); Days in prior months this year.
       day))); Days this month.
```
(17.9)

```
(defun arithmetic-french-from-fixed (date)
  ;; TYPE fixed-date -> french-date
  ;; Arithmetic French Revolutionary date (year month day)
  ;; of fixed date.
  (let* ((approx   ; Approximate year (may be off by 1).
          (1+ (quotient (- date french-epoch -2)
                        1460969/4000)))
         (year (if (< date
                      (fixed-from-arithmetic-french
                       (french-date approx 1 1)))
                   (1- approx)
                 approx))
         (month    ; Calculate the month by division.
          (1+ (quotient
               (- date (fixed-from-arithmetic-french
```
(17.10)

```
                         (french-date year 1 1)))
                30)))
         (day      ; Calculate the day by subtraction.
          (1+ (- date
                  (fixed-from-arithmetic-french
                   (french-date year month 1))))))
    (french-date year month day)))
```

D.18 Astronomical Lunar Calendars

```
(defun babylonian-date (year month leap day)
  ;; TYPE (babylonian-year babylonian-month
  ;; TYPE  babylonian-leap babylonian-day)
  ;; TYPE  -> babylonian-date
  (list year month leap day))
```

```
(defun babylonian-year (date)
  ;; TYPE babylonian-date -> babylonian-year
  (first date))
```

```
(defun babylonian-month (date)
  ;; TYPE babylonian-date -> babylonian-month
  (second date))
```

```
(defun babylonian-leap (date)
  ;; TYPE babylonian-date -> babylonian-leap
  (third date))
```

```
(defun babylonian-day (date)
  ;; TYPE babylonian-date -> babylonian-day
  (fourth date))
```

```
(defun moonlag (date location)                              (18.1)
  ;; TYPE (fixed-date location) -> duration
  ;; Time between sunset and moonset on date at location.
  ;; Returns bogus if there is no sunset on date.
  (let* ((sun (sunset date location))
         (moon (moonset date location)))
    (cond ((equal sun bogus) bogus)
          ((equal moon bogus) (hr 24)) ; Arbitrary.
          (t (- moon sun)))))
```

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

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

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

```
          (age (- date moon))
          (tau ; Check if not visible yet on eve of date.
           (if (and (<= age 3)
                    (not (babylonian-criterion date)))
               (- moon 30) ; Must go back a month.
             moon)))
     (next d tau (babylonian-criterion d))))
```

(18.5)

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

(18.6)

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

(18.7)

```
(defun fixed-from-babylonian (b-date)
  ;; TYPE babylonian-date -> fixed-date
  ;; Fixed date equivalent to Babylonian date.
  (let* ((month (babylonian-month b-date))
         (leap (babylonian-leap b-date))
         (day (babylonian-day b-date))
         (year (babylonian-year b-date))
         (month1 ;  Elapsed months this year.
          (if (or leap
                  (and (= (mod year 19) 18)
                       (> month 6)))
              month (1- month)))
         (months ; Elapsed months since epoch.
          (+ (quotient (+ (* (1- year) 235) 13) 19)
             month1))
         (midmonth ; Middle of given month.
          (+ babylonian-epoch
             (round (* mean-synodic-month months)) 15)))
    (+ (babylonian-new-month-on-or-before midmonth)
       day -1)))
```

(18.8)

```
(defun babylonian-from-fixed (date)
  ;; TYPE fixed-date -> babylonian-date
  ;; Babylonian date corresponding to fixed date.
  (let* ((crescent ; Most recent new month.
          (babylonian-new-month-on-or-before date))
         (months ; Elapsed months since epoch.
          (round (/ (- crescent babylonian-epoch)
                    mean-synodic-month)))
         (year (1+ (quotient (+ (* 19 months) 5) 235)))
         (approx ; Approximate date of new year.
          (+ babylonian-epoch
             (round (* (quotient (+ (* (1- year) 235) 13) 19)
                       mean-synodic-month))))
         (new-year (babylonian-new-month-on-or-before
                    (+ approx 15)))
         (month1 (1+ (round (/ (- crescent new-year) 29.5L0))))
         (special (= (mod year 19) 18))
         (leap (if special (= month1 7) (= month1 13)))
         (month (if (or leap (and special (> month1 6)))
                    (1- month1)
                  month1))
         (day (- date crescent -1)))
    (babylonian-date year month leap day)))
```

(18.9)

```
(defun astronomical-easter (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Date of (proposed) astronomical Easter in Gregorian
  ;; year g-year.
  (let* ((equinox ; Spring equinox.
```

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

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

```
(defun fixed-from-observational-islamic (i-date)                (18.11)
  ;; TYPE islamic-date -> fixed-date
  ;; Fixed date equivalent to Observational Islamic date
  ;; i-date.
  (let* ((month (standard-month i-date))
         (day (standard-day i-date))
         (year (standard-year i-date))
         (midmonth ; Middle of given month.
          (+ islamic-epoch
             (floor (* (+ (* (1- year) 12)
                          month -1/2)
                       mean-synodic-month)))))
    (+ (phasis-on-or-before ; First day of month.
        midmonth islamic-location)
       day -1)))
```

```
(defun observational-islamic-from-fixed (date)                  (18.12)
  ;; TYPE fixed-date -> islamic-date
  ;; Observational Islamic date (year month day)
  ;; corresponding to fixed date.
  (let* ((crescent ; Most recent new moon.
          (phasis-on-or-before date islamic-location))
         (elapsed-months
          (round (/ (- crescent islamic-epoch)
                    mean-synodic-month)))
         (year (1+ (quotient elapsed-months 12)))
         (month (1+ (mod elapsed-months 12)))
         (day (1+ (- date crescent))))
    (islamic-date year month day)))
```

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

```
(defun early-month? (date location)                             (18.14)
  ;; TYPE (fixed-date location) -> boolean
  ;; Fixed date in location is in a month that was forced to
  ;; start early.
  (let* ((start (phasis-on-or-before date location))
         (prev (- start 15)))
    (or (>= (- date start) 30)
        (> (month-length prev location) 30)
        (and (= (month-length prev location) 30)
             (early-month? prev location)))))
```

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

```
       (day (standard-day i-date))
       (year (standard-year i-date))
       (midmonth ; Middle of given month.
        (+ islamic-epoch
           (floor (* (+ (* (1- year) 12)
                        month -1/2)
                     mean-synodic-month))))
       (moon (phasis-on-or-before ; First day of month.
              midmonth islamic-location))
       (date (+ moon day -1)))
  (if (early-month? midmonth islamic-location) (1- date) date)))
```

(18.16)

```
(defun alt-observational-islamic-from-fixed (date)
  ;; TYPE fixed-date -> islamic-date
  ;; Observational Islamic date (year month day)
  ;; corresponding to fixed date.
  ;; Months are never longer than 30 days.
  (let* ((early (early-month? date islamic-location))
         (long (and early
                    (> (month-length date islamic-location) 29)))
         (date-prime
          (if long (1+ date) date))
         (moon ; Most recent new moon.
          (phasis-on-or-before date-prime islamic-location))
         (elapsed-months
          (round (/ (- moon islamic-epoch)
                    mean-synodic-month)))
         (year (1+ (quotient elapsed-months 12)))
         (month (1+ (mod elapsed-months 12)))
         (day (- date-prime moon
                 (if (and early (not long)) -2 -1))))
    (islamic-date year month day)))
```

(18.17)

```
(defun saudi-criterion (date)
  ;; TYPE fixed-date -> boolean
  ;; Saudi visibility criterion on eve of fixed date in Mecca.
  (let* ((set (sunset (1- date) mecca))
         (tee (universal-from-standard set mecca))
         (phase (lunar-phase tee)))
    (and (< new phase first-quarter)
         (> (moonlag (1- date) mecca) 0))))
```

(18.18)

```
(defun saudi-new-month-on-or-before (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Closest fixed date on or before date when Saudi
  ;; visibility criterion held.
  (let* ((moon ; Prior new moon.
          (fixed-from-moment
           (lunar-phase-at-or-before new date)))
         (age (- date moon))
         (tau ; Check if not visible yet on eve of date.
          (if (and (<= age 3)
                   (not (saudi-criterion date)))
              (- moon 30) ; Must go back a month.
            moon)))
    (next d tau (saudi-criterion d))))
```

(18.19)

```
(defun fixed-from-saudi-islamic (s-date)
  ;; TYPE islamic-date -> fixed-date
  ;; Fixed date equivalent to Saudi Islamic date s-date.
  (let* ((month (standard-month s-date))
         (day (standard-day s-date))
         (year (standard-year s-date))
         (midmonth ; Middle of given month.
          (+ islamic-epoch
             (floor (* (+ (* (1- year) 12)
                          month -1/2)
                       mean-synodic-month)))))
    (+ (saudi-new-month-on-or-before ; First day of month.
        midmonth)
       day -1)))
```

(18.20)

```
(defun saudi-islamic-from-fixed (date)
  ;; TYPE fixed-date -> islamic-date
  ;; Saudi Islamic date (year month day) corresponding to
  ;; fixed date.
  (let* ((crescent ; Most recent new moon.
          (saudi-new-month-on-or-before date))
         (elapsed-months
          (round (/ (- crescent islamic-epoch)
                    mean-synodic-month)))
         (year (1+ (quotient elapsed-months 12)))
         (month (1+ (mod elapsed-months 12)))
         (day (1+ (- date crescent))))
    (islamic-date year month day)))
```

(18.21)

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

(18.22)

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

(18.23)

```
(defun observational-hebrew-from-fixed (date)
  ;; TYPE fixed-date -> hebrew-date
  ;; Observational Hebrew date (year month day)
  ;; corresponding to fixed date.
  (let* ((crescent ; Most recent new moon.
          (phasis-on-or-before date hebrew-location))
         (g-year (gregorian-year-from-fixed date))
         (ny (observational-hebrew-first-of-nisan g-year))
         (new-year (if (< date ny)
                       (observational-hebrew-first-of-nisan
                        (1- g-year))
                     ny))
         (month (1+ (round (/ (- crescent new-year) 29.5L0))))
         (year (+ (standard-year (hebrew-from-fixed new-year))
                  (if (>= month tishri) 1 0)))
         (day (- date crescent -1)))
    (hebrew-date year month day)))
```

(18.24)

```
(defun fixed-from-observational-hebrew (h-date)
  ;; TYPE hebrew-date -> fixed-date
  ;; Fixed date equivalent to Observational Hebrew date.
  (let* ((month (standard-month h-date))
         (day (standard-day h-date))
         (year (standard-year h-date))
         (year1 (if (>= month tishri) (1- year) year))
         (start (fixed-from-hebrew
                 (hebrew-date year1 nisan 1)))
         (g-year (gregorian-year-from-fixed
                  (+ start 60)))
         (new-year (observational-hebrew-first-of-nisan g-year))
         (midmonth ; Middle of given month.
          (+ new-year (round (* 29.5L0 (1- month))) 15)))
    (+ (phasis-on-or-before ; First day of month.
        midmonth hebrew-location)
       day -1)))
```

```
(defun classical-passover-eve (g-year)                                  (18.25)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Classical (observational) Passover Eve
  ;; (Nisan 14) occurring in Gregorian year g-year.
  (+ (observational-hebrew-first-of-nisan g-year) 13))
```

```
(defun alt-observational-hebrew-from-fixed (date)                       (18.26)
  ;; TYPE fixed-date -> hebrew-date
  ;; Observational Hebrew date (year month day)
  ;; corresponding to fixed date.
  ;; Months are never longer than 30 days.
  (let* ((early (early-month? date hebrew-location))
         (long (and early (> (month-length date hebrew-location) 29)))
         (date-prime
          (if long (1+ date) date))
         (moon ; Most recent new moon.
          (phasis-on-or-before date-prime hebrew-location))
         (g-year (gregorian-year-from-fixed date-prime))
         (ny (observational-hebrew-first-of-nisan g-year))
         (new-year (if (< date-prime ny)
                       (observational-hebrew-first-of-nisan
                        (1- g-year))
                     ny))
         (month (1+ (round (/ (- moon new-year) 29.5L0))))
         (year (+ (standard-year (hebrew-from-fixed new-year))
                  (if (>= month tishri) 1 0)))
         (day (- date-prime moon
                 (if (and early (not long)) -2 -1))))
    (hebrew-date year month day)))
```

```
(defun alt-fixed-from-observational-hebrew (h-date)                     (18.27)
  ;; TYPE hebrew-date -> fixed-date
  ;; Fixed date equivalent to Observational Hebrew h-date.
  ;; Months are never longer than 30 days.
  (let* ((month (standard-month h-date))
         (day (standard-day h-date))
         (year (standard-year h-date))
         (year1 (if (>= month tishri) (1- year) year))
         (start (fixed-from-hebrew
                 (hebrew-date year1 nisan 1)))
         (g-year (gregorian-year-from-fixed
                  (+ start 60)))
         (new-year (observational-hebrew-first-of-nisan g-year))
         (midmonth ; Middle of given month.
          (+ new-year (round (* 29.5L0 (1- month))) 15))
         (moon (phasis-on-or-before ; First day of month.
                midmonth hebrew-location))
         (date (+ moon day -1)))
    (if (early-month? midmonth hebrew-location) (1- date) date)))
```

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

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

```
(defun samaritan-new-moon-after (tee)                                   (18.30)
  ;; TYPE moment -> fixed-date
  ;; Fixed date of first new moon after UT moment tee.
  ;; Modern calculation.
  (ceiling
   (- (apparent-from-universal (new-moon-at-or-after tee)
                               samaritan-location)
      (hr 12))))
```

(18.31)
```
(defun samaritan-new-moon-at-or-before (tee)
  ;; TYPE moment -> fixed-date
  ;; Fixed-date of last new moon before UT moment tee.
  ;; Modern calculation.
  (ceiling
   (- (apparent-from-universal (new-moon-before tee)
                               samaritan-location)
      (hr 12))))
```

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

(18.33)
```
(defun samaritan-new-year-on-or-before (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Fixed date of Samaritan New Year on or before fixed
  ;; date.
  (let* ((g-year (gregorian-year-from-fixed date))
         (dates ; All possible March 11's.
          (append
           (julian-in-gregorian march 11 (1- g-year))
           (julian-in-gregorian march 11 g-year)
           (list (1+ date)))) ; Extra to stop search.
         (n
          (final i 0
                 (<= (samaritan-new-moon-after
                      (samaritan-noon (nth i dates)))
                     date))))
    (samaritan-new-moon-after (samaritan-noon (nth n dates)))))
```

(18.34)
```
(defun fixed-from-samaritan (s-date)
  ;; TYPE hebrew-date -> fixed-date
  ;; Fixed date of Samaritan date h-date.
  (let* ((month (standard-month s-date))
         (day (standard-day s-date))
         (year (standard-year s-date))
         (ny (samaritan-new-year-on-or-before
              (floor (+ samaritan-epoch 50
                        (* 365.25L0 (- year
                                       (ceiling (- month 5) 8)))))))
         (nm (samaritan-new-moon-at-or-before
              (+ ny (* 29.5L0 (1- month)) 15))))
    (+ nm day -1)))
```

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

D.19 The Chinese Calendar

```
(defun chinese-date (cycle year month leap day)
  ;; TYPE (chinese-cycle chinese-year chinese-month
  ;; TYPE  chinese-leap chinese-day) -> chinese-date
  (list cycle year month leap day))
```

```
(defun chinese-cycle (date)
  ;; TYPE chinese-date -> chinese-cycle
  (first date))
```

```
(defun chinese-year (date)
  ;; TYPE chinese-date -> chinese-year
  (second date))
```

```
(defun chinese-month (date)
  ;; TYPE chinese-date -> chinese-month
  (third date))
```

```
(defun chinese-leap (date)
  ;; TYPE chinese-date -> chinese-leap
  (fourth date))
```

```
(defun chinese-day (date)
  ;; TYPE chinese-date -> chinese-day
  (fifth date))
```

(19.1)

```
(defun current-major-solar-term (date)
  ;; TYPE fixed-date -> integer
  ;; Last Chinese major solar term (zhongqi) before fixed
  ;; date.
  (let* ((s (solar-longitude
             (universal-from-standard
              date
              (chinese-location date)))))
    (amod (+ 2 (quotient s (deg 30))) 12)))
```

(19.2)

```
(defun chinese-location (tee)
  ;; TYPE moment -> location
  ;; Location of Beijing; time zone varies with tee.
  (let* ((year (gregorian-year-from-fixed (floor tee))))
    (if (< year 1929)
        (location (angle 39 55 0) (angle 116 25 0)
                  (mt 43.5) (hr 1397/180))
      (location (angle 39 55 0) (angle 116 25 0)
                (mt 43.5) (hr 8)))))
```

(19.3)

```
(defun chinese-solar-longitude-on-or-after (lambda tee)
  ;; TYPE (season moment) -> moment
  ;; Moment (Beijing time) of the first time at or after
  ;; tee (Beijing time) when the solar longitude
  ;; will be lambda degrees.
  (let* ((sun (solar-longitude-after
               lambda
               (universal-from-standard
                tee
                (chinese-location tee)))))
    (standard-from-universal
     sun
     (chinese-location sun))))
```

(19.4)

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

(19.5)

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

```
              (universal-from-standard
               date
               (chinese-location date)))))
     (amod (+ 3 (quotient (- s (deg 15)) (deg 30)))
           12)))
```

(19.6)
```
(defun minor-solar-term-on-or-after (date)
  ;; TYPE fixed-date -> moment
  ;; Moment (in Beijing) of the first Chinese minor solar
  ;; term (jieqi) on or after fixed date.  The minor terms
  ;; begin when the sun's longitude is an odd multiple of 15
  ;; degrees.
  (let* ((s (solar-longitude (midnight-in-china date)))
         (l (mod
             (+ (* 30
                   (ceiling
                    (/ (- s (deg 15)) 30)))
                (deg 15))
             360)))
    (chinese-solar-longitude-on-or-after l date)))
```

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

(19.8)
```
(defun chinese-winter-solstice-on-or-before (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Fixed date, in the Chinese zone, of winter solstice
  ;; on or before fixed date.
  (let* ((approx ; Approximate time of solstice.
          (estimate-prior-solar-longitude
           winter (midnight-in-china (+ date 1)))))
    (next day (1- (floor approx))
          (< winter (solar-longitude
                     (midnight-in-china (1+ day)))))))
```

(19.9)
```
(defun chinese-new-moon-on-or-after (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Fixed date (Beijing) of first new moon on or after
  ;; fixed date.
  (let* ((tee (new-moon-at-or-after
               (midnight-in-china date))))
    (floor
     (standard-from-universal
      tee
      (chinese-location tee)))))
```

(19.10)
```
(defun chinese-new-moon-before (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Fixed date (Beijing) of first new moon before fixed
  ;; date.
  (let* ((tee (new-moon-before
               (midnight-in-china date))))
    (floor
     (standard-from-universal
      tee
      (chinese-location tee)))))
```

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

(19.12)

```
(defun chinese-prior-leap-month? (m-prime m)
  ;; TYPE (fixed-date fixed-date) -> boolean
  ;; True if there is a Chinese leap month on or after lunar
  ;; month starting on fixed day m-prime and at or before
  ;; lunar month starting at fixed date m.
  (and (>= m m-prime)
       (or (chinese-no-major-solar-term? m)
           (chinese-prior-leap-month?
            m-prime
            (chinese-new-moon-before m)))))
```

(19.13)

```
(defun chinese-new-year-in-sui (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Fixed date of Chinese New Year in sui (period from
  ;; solstice to solstice) containing date.
  (let* ((s1; prior solstice
          (chinese-winter-solstice-on-or-before date))
         (s2; following solstice
          (chinese-winter-solstice-on-or-before
           (+ s1 370)))
         (m12 ; month after 11th month--either 12 or leap 11
          (chinese-new-moon-on-or-after (1+ s1)))
         (m13 ; month after m12--either 12 (or leap 12) or 1
          (chinese-new-moon-on-or-after (1+ m12)))
         (next-m11 ; next 11th month
          (chinese-new-moon-before (1+ s2))))
    (if ; Either m12 or m13 is a leap month if there are
        ; 13 new moons (12 full lunar months) and
        ; either m12 or m13 has no major solar term
        (and (= (round (/ (- next-m11 m12)
                          mean-synodic-month))
                12)
             (or (chinese-no-major-solar-term? m12)
                 (chinese-no-major-solar-term? m13)))
        (chinese-new-moon-on-or-after (1+ m13))
      m13)))
```

(19.14)

```
(defun chinese-new-year-on-or-before (date)
  ;; TYPE fixed-date -> fixed-date
  ;; Fixed date of Chinese New Year on or before fixed date.
  (let* ((new-year (chinese-new-year-in-sui date)))
    (if (>= date new-year)
        new-year
      ;; Got the New Year after--this happens if date is
      ;; after the solstice but before the new year.
      ;; So, go back half a year.
      (chinese-new-year-in-sui (- date 180)))))
```

(19.15)

```
(defconstant chinese-epoch
  ;; TYPE fixed-date
  ;; Fixed date of start of the Chinese calendar.
  (fixed-from-gregorian (gregorian-date -2636 february 15)))
```

(19.16)

```
(defun chinese-from-fixed (date)
  ;; TYPE fixed-date -> chinese-date
  ;; Chinese date (cycle year month leap day) of fixed date.
  (let* ((s1; Prior solstice
          (chinese-winter-solstice-on-or-before date))
         (s2; Following solstice
          (chinese-winter-solstice-on-or-before (+ s1 370)))
         (m12     ; month after last 11th month
          (chinese-new-moon-on-or-after (1+ s1)))
         (next-m11; next 11th month
          (chinese-new-moon-before (1+ s2)))
         (m      ; start of month containing date
          (chinese-new-moon-before (1+ date)))
         (leap-year; if there are 13 new moons (12 full
                                              ; lunar months)
          (= (round (/ (- next-m11 m12)
                       mean-synodic-month))
```

```
              12))
          (month  ; month number
           (amod
            (-
             ;; ordinal position of month in year
             (round (/ (- m m12) mean-synodic-month))
             ;; minus 1 during or after a leap month
             (if (and leap-year
                      (chinese-prior-leap-month? m12 m))
                 1
               0))
            12))
          (leap-month    ; it's a leap month if...
           (and
            leap-year; ...there are 13 months
            (chinese-no-major-solar-term?
             m)                             ; no major solar term
            (not (chinese-prior-leap-month? ; and no prior leap
                                            ; month
                  m12 (chinese-new-moon-before m))))))
          (elapsed-years  ; Approximate since the epoch
           (floor (+ 1.5L0  ; 18 months (because of truncation)
                     (- (/ month 12)); after at start of year
                     (/ (- date chinese-epoch)
                        mean-tropical-year))))
          (cycle (1+ (quotient (1- elapsed-years) 60)))
          (year (amod elapsed-years 60))
          (day (1+ (- date m))))
    (chinese-date cycle year month leap-month day)))
```

(19.17)

```
(defun fixed-from-chinese (c-date)
  ;; TYPE chinese-date -> fixed-date
  ;; Fixed date of Chinese date c-date.
  (let* ((cycle (chinese-cycle c-date))
         (year (chinese-year c-date))
         (month (chinese-month c-date))
         (leap (chinese-leap c-date))
         (day (chinese-day c-date))
         (mid-year      ;  Middle of the Chinese year
          (floor
           (+ chinese-epoch
              (* (+ (* (1- cycle) 60); years in prior cycles
                    (1- year)        ; prior years this cycle
                    1/2)             ; half a year
                 mean-tropical-year))))
         (new-year (chinese-new-year-on-or-before mid-year))
         (p  ; new moon before date--a month too early if
             ; there was prior leap month that year
          (chinese-new-moon-on-or-after
           (+ new-year (* (1- month) 29))))
         (d (chinese-from-fixed p))
         (prior-new-moon
          (if  ; If the months match...
              (and (= month (chinese-month d))
                   (equal leap (chinese-leap d)))
              p; ...that's the right month
            ;; otherwise, there was a prior leap month that
            ;; year, so we want the next month
            (chinese-new-moon-on-or-after (1+ p)))))
    (+ prior-new-moon day -1)))
```

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

```
(defun chinese-stem (name)
  ;; TYPE chinese-name -> chinese-stem
  (first name))
```

```
(defun chinese-branch (name)
  ;; TYPE chinese-name -> chinese-branch
  (second name))
```

(19.18)

```
(defun chinese-sexagesimal-name (n)
  ;; TYPE integer -> chinese-name
  ;; The n-th name of the Chinese sexagesimal cycle.
  (chinese-name (amod n 10)
                (amod n 12)))
```

(19.19)

```
(defun chinese-name-difference (c-name1 c-name2)
  ;; TYPE (chinese-name chinese-name) -> nonnegative-integer
  ;; Number of names from Chinese name c-name1 to the
  ;; next occurrence of Chinese name c-name2.
  (let* ((stem1 (chinese-stem c-name1))
         (stem2 (chinese-stem c-name2))
         (branch1 (chinese-branch c-name1))
         (branch2 (chinese-branch c-name2))
         (stem-difference (- stem2 stem1))
         (branch-difference (- branch2 branch1)))
    (amod (+ stem-difference
             (* 25 (- branch-difference
                      stem-difference)))
          60)))
```

(19.20)

```
(defun chinese-year-name (year)
  ;; TYPE chinese-year -> chinese-name
  ;; Sexagesimal name for Chinese year of any cycle.
  (chinese-sexagesimal-name year))
```

(19.21)

```
(defconstant chinese-month-name-epoch
  ;; TYPE integer
  ;; Elapsed months at start of Chinese sexagesimal month
  ;; cycle.
  57)
```

(19.22)

```
(defun chinese-month-name (month year)
  ;; TYPE (chinese-month chinese-year) -> chinese-name
  ;; Sexagesimal name for month month of Chinese year
  ;; year.
  (let* ((elapsed-months (+ (* 12 (1- year))
                            (1- month))))
    (chinese-sexagesimal-name
     (- elapsed-months chinese-month-name-epoch))))
```

(19.23)

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

(19.24)

```
(defun chinese-day-name (date)
  ;; TYPE fixed-date -> chinese-name
  ;; Chinese sexagesimal name for date.
  (chinese-sexagesimal-name
   (- date chinese-day-name-epoch)))
```

(19.25)

```
(defun chinese-day-name-on-or-before (name date)
  ;; TYPE (chinese-name fixed-date) -> fixed-date
  ;; Fixed date of latest date on or before fixed date
  ;; that has Chinese name.
  (mod3 (chinese-name-difference
         (chinese-day-name 0) name)
        date (- date 60)))
```

(19.26)

```
(defun chinese-new-year (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Chinese New Year in Gregorian year g-year.
  (chinese-new-year-on-or-before
   (fixed-from-gregorian
    (gregorian-date g-year july 1))))
```

(19.27)

```
(defun dragon-festival (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of the Dragon Festival occurring in
  ;; Gregorian year g-year.
  (let* ((elapsed-years
          (1+ (- g-year
                 (gregorian-year-from-fixed
                  chinese-epoch))))
         (cycle (1+ (quotient (1- elapsed-years) 60)))
         (year (amod elapsed-years 60)))
    (fixed-from-chinese (chinese-date cycle year 5 false 5))))
```

(19.28)

```
(defun qing-ming (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Qingming occurring in Gregorian year
  ;; g-year.
  (floor
   (minor-solar-term-on-or-after
    (fixed-from-gregorian
     (gregorian-date g-year march 30)))))
```

(19.29)

```
(defun chinese-age (birthdate date)
  ;; TYPE (chinese-date fixed-date) -> nonnegative-integer
  ;; Age at fixed date, given Chinese birthdate,
  ;; according to the Chinese custom.  Returns bogus if
  ;; date is before birthdate.
  (let* ((today (chinese-from-fixed date)))
    (if (>= date (fixed-from-chinese birthdate))
        (+ (* 60 (- (chinese-cycle today)
                    (chinese-cycle birthdate)))
           (- (chinese-year today)
              (chinese-year birthdate))
           1)
      bogus)))
```

(19.30)

```
(defconstant double-bright
  ;; TYPE augury
  ;; Lichun occurs twice (double-happiness).
  3)
```

(19.31)

```
(defconstant bright
  ;; TYPE augury
  ;; Lichun occurs once at the start.
  2)
```

(19.32)

```
(defconstant blind
  ;; TYPE augury
  ;; Lichun occurs once at the end.
  1)
```

(19.33)

```
(defconstant widow
  ;; TYPE augury
  ;; Lichun does not occur (double-blind year).
  0)
```

(19.34)

```
(defun chinese-year-marriage-augury (cycle year)
  ;; TYPE (chinese-cycle chinese-year) -> augury
  ;; The marriage augury type of Chinese year in cycle.
  (let* ((new-year (fixed-from-chinese
```

```
                     (chinese-date cycle year 1 false 1)))
          (c (if (= year 60); next year's cycle
                 (1+ cycle)
               cycle))
          (y (if (= year 60); next year's number
                 1
               (1+ year)))
          (next-new-year (fixed-from-chinese
                          (chinese-date c y 1 false 1)))
          (first-minor-term
           (current-minor-solar-term new-year))
          (next-first-minor-term
           (current-minor-solar-term next-new-year)))
     (cond
      ((and
        (= first-minor-term 1)        ; no lichun at start...
        (= next-first-minor-term 12)) ; ...or at end
       widow)
      ((and
        (= first-minor-term 1)        ; no lichun at start...
        (/= next-first-minor-term 12)); ...only at end
       blind)
      ((and
        (/= first-minor-term 1)       ; lichun at start...
        (= next-first-minor-term 12)) ; ... not at end
       bright)
      (t double-bright))))            ; lichun at start and end
```

(19.35)

```
(defun japanese-location (tee)
  ;; TYPE moment -> location
  ;; Location for Japanese calendar; varies with tee.
  (let* ((year (gregorian-year-from-fixed (floor tee))))
    (if (< year 1888)
        ;; Tokyo (139 deg 46 min east) local time
        (location (deg 35.7L0) (angle 139 46 0)
                  (mt 24) (hr (+ 9 143/450)))
                                      ; Longitude 135 time zone
      (location (deg 35) (deg 135) (mt 0) (hr 9)))))
```

(19.36)

```
(defun korean-location (tee)
  ;; TYPE moment -> location
  ;; Location for Korean calendar; varies with tee.
  ;; Seoul city hall at a varying time zone.
  (let* ((z (cond
             ((< tee
                 (fixed-from-gregorian
                  (gregorian-date 1908 april 1)))
              ;; local mean time for longitude 126 deg 58 min
              3809/450)
             ((< tee
                 (fixed-from-gregorian
                  (gregorian-date 1912 january 1)))
              8.5)
             ((< tee
                 (fixed-from-gregorian
                  (gregorian-date 1954 march 21)))
              9)
             ((< tee
                 (fixed-from-gregorian
                  (gregorian-date 1961 august 10)))
              8.5)
             (t 9))))
    (location (angle 37 34 0) (angle 126 58 0)
              (mt 0) (hr z))))
```

(19.37)

```
(defun korean-year (cycle year)
  ;; TYPE (chinese-cycle chinese-year) -> integer
  ;; Equivalent Korean year to Chinese cycle and year
  (+ (* 60 cycle) year -364))
```

(19.38)

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

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.

(20.1)

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

(20.2)

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

(20.3)

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

(20.4)

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

(20.5)

```
(defun hindu-sine (theta)
  ;; TYPE rational-angle -> rational-amplitude
  ;; Linear interpolation for theta in Hindu table.
  (let* ((entry
          (/ theta (angle 0 225 0))); Interpolate in table.
         (fraction (mod entry 1)))
    (+ (* fraction
          (hindu-sine-table (ceiling entry)))
       (* (- 1 fraction)
          (hindu-sine-table (floor entry))))))
```

(20.6)

```
(defun hindu-arcsin (amp)
  ;; TYPE rational-amplitude -> rational-angle
  ;; Inverse of Hindu sine function of amp.
  (if (< amp 0) (- (hindu-arcsin (- amp)))
    (let* ((pos (next k 0 (<= amp (hindu-sine-table k))))
           (below ; Lower value in table.
            (hindu-sine-table (1- pos))))
      (* (angle 0 225 0)
         (+ pos -1  ; Interpolate.
            (/ (- amp below)
               (- (hindu-sine-table pos) below)))))))
```

(20.7)

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

(20.8)

```
(defconstant hindu-creation
  ;; TYPE fixed-date
  ;; Fixed date of Hindu creation.
  (- hindu-epoch (* 1955880000 hindu-sidereal-year)))
```

(20.9)

```
(defconstant hindu-anomalistic-year
  ;; TYPE rational
  ;; Time from aphelion to aphelion.
  (/ 1577917828000 (- 4320000000 387)))
```

(20.10)

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

(20.11)

```
(defun hindu-true-position (tee period size anomalistic change)
  ;; TYPE (rational-moment rational rational rational
  ;; TYPE  rational) -> rational-angle
  ;; Longitudinal position at moment tee.  period is
  ;; period of mean motion in days.  size is ratio of
  ;; radii of epicycle and deferent.  anomalistic is the
  ;; period of retrograde revolution about epicycle.
  ;; change is maximum decrease in epicycle size.
  (let* ((lambda ; Position of epicycle center
           (hindu-mean-position tee period))
         (offset ; Sine of anomaly
          (hindu-sine (hindu-mean-position tee anomalistic)))
         (contraction (* (abs offset) change size))
         (equation ; Equation of center
          (hindu-arcsin (* offset (- size contraction)))))
    (mod (- lambda equation) 360)))
```

(20.12)

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

(20.13)

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

(20.14)

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

(20.15)

```
(defun hindu-lunar-phase (tee)
  ;; TYPE rational-moment -> rational-angle
  ;; Longitudinal distance between the sun and moon
  ;; at moment tee.
  (mod (- (hindu-lunar-longitude tee)
          (hindu-solar-longitude tee))
       360))
```

(20.16)
```
(defun hindu-lunar-day-from-moment (tee)
  ;; TYPE rational-moment -> hindu-lunar-day
  ;; Phase of moon (tithi) at moment tee, as an integer in
  ;; the range 1..30.
  (1+ (quotient (hindu-lunar-phase tee) (deg 12))))
```

(20.17)
```
(defun hindu-new-moon-before (tee)
  ;; TYPE rational-moment -> rational-moment
  ;; Approximate moment of last new moon preceding moment
  ;; tee, close enough to determine zodiacal sign.
  (let* ((varepsilon (expt 2 -1000)) ; Safety margin.
         (tau  ; Can be off by almost a day.
          (- tee (* (/ 1 (deg 360)) (hindu-lunar-phase tee)
                    hindu-synodic-month))))
    (binary-search ; Search for phase start.
     l (1- tau)
     u (min tee (1+ tau))
     x (< (hindu-lunar-phase x) (deg 180))
     (or (= (hindu-zodiac l) (hindu-zodiac u))
         (< (- u l) varepsilon)))))
```

```
(defun hindu-solar-date (year month day)
  ;; TYPE (hindu-solar-year hindu-solar-month hindu-solar-day)
  ;; TYPE  -> hindu-solar-date
  (list year month day))
```

(20.18)
```
(defun hindu-calendar-year (tee)
  ;; TYPE rational-moment -> hindu-solar-year
  ;; Determine solar year at given moment tee.
  (round (- (/ (- tee hindu-epoch)
               hindu-sidereal-year)
            (/ (hindu-solar-longitude tee)
               (deg 360)))))
```

(20.19)
```
(defconstant hindu-solar-era
  ;; TYPE standard-year
  ;; Years from Kali Yuga until Saka era.
  3179)
```

(20.20)
```
(defun hindu-solar-from-fixed (date)
  ;; TYPE fixed-date -> hindu-solar-date
  ;; Hindu (Orissa) solar date equivalent to fixed date.
  (let* ((critical    ; Sunrise on Hindu date.
          (hindu-sunrise (1+ date)))
         (month (hindu-zodiac critical))
         (year (- (hindu-calendar-year critical)
                  hindu-solar-era))
         (approx ; 3 days before start of mean month.
          (- date 3
             (mod (floor (hindu-solar-longitude critical))
                  (deg 30))))
         (start ; Search forward for beginning...
          (next i approx ; ... of month.
                (= (hindu-zodiac (hindu-sunrise (1+ i)))
                   month)))
         (day (- date start -1)))
    (hindu-solar-date year month day)))
```

(20.21)
```
(defun fixed-from-hindu-solar (s-date)
  ;; TYPE hindu-solar-date -> fixed-date
  ;; Fixed date corresponding to Hindu solar date s-date
  ;; (Saka era; Orissa rule.)
  (let* ((month (standard-month s-date))
         (day (standard-day s-date))
         (year (standard-year s-date))
         (start ; Approximate start of month
                                          ; by adding days...
          (+ (floor (* (+ year hindu-solar-era
```

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

```
(defun hindu-lunar-date (year month leap-month day leap-day)
  ;; TYPE (hindu-lunar-year hindu-lunar-month
  ;; TYPE  hindu-lunar-leap-month hindu-lunar-day
  ;; TYPE  hindu-lunar-leap-day) -> hindu-lunar-date
  (list year month leap-month day leap-day))
```

```
(defun hindu-lunar-month (date)
  ;; TYPE hindu-lunar-date -> hindu-lunar-month
  (second date))
```

```
(defun hindu-lunar-leap-month (date)
  ;; TYPE hindu-lunar-date -> hindu-lunar-leap-month
  (third date))
```

```
(defun hindu-lunar-day (date)
  ;; TYPE hindu-lunar-date -> hindu-lunar-day
  (fourth date))
```

```
(defun hindu-lunar-leap-day (date)
  ;; TYPE hindu-lunar-date -> hindu-lunar-leap-day
  (fifth date))
```

```
(defun hindu-lunar-year (date)
  ;; TYPE hindu-lunar-date -> hindu-lunar-year
  (first date))
```

(20.22)

```
(defconstant hindu-lunar-era
  ;; TYPE standard-year
  ;; Years from Kali Yuga until Vikrama era.
  3044)
```

(20.23)

```
(defun hindu-lunar-from-fixed (date)
  ;; TYPE fixed-date -> hindu-lunar-date
  ;; Hindu lunar date, new-moon scheme,
  ;; equivalent to fixed date.
  (let* ((critical (hindu-sunrise date)) ; Sunrise that day.
         (day (hindu-lunar-day-from-moment
               critical)); Day of month.
         (leap-day        ; If previous day the same.
          (= day (hindu-lunar-day-from-moment
                  (hindu-sunrise (- date 1)))))
         (last-new-moon
          (hindu-new-moon-before critical))
         (next-new-moon
          (hindu-new-moon-before
           (+ (floor last-new-moon) 35)))
         (solar-month         ; Solar month name.
          (hindu-zodiac last-new-moon))
         (leap-month       ; If begins and ends in same sign.
          (= solar-month (hindu-zodiac next-new-moon)))
         (month                    ; Month of lunar year.
          (amod (1+ solar-month) 12))
         (year ; Solar year at end of month.
          (- (hindu-calendar-year
              (if (<= month 2) ; date might precede solar
                                         ; new year.
```

```
                  (+ date 180)
                date))
             hindu-lunar-era)))
     (hindu-lunar-date year month leap-month day leap-day)))
```

(20.24)

```
(defun fixed-from-hindu-lunar (l-date)
  ;; TYPE hindu-lunar-date -> fixed-date
  ;; Fixed date corresponding to Hindu lunar date l-date.
  (let* ((year (hindu-lunar-year l-date))
         (month (hindu-lunar-month l-date))
         (leap-month (hindu-lunar-leap-month l-date))
         (day (hindu-lunar-day l-date))
         (leap-day (hindu-lunar-leap-day l-date))
         (approx
          (+ hindu-epoch
             (* hindu-sidereal-year
                (+ year hindu-lunar-era
                   (/ (1- month) 12)))))
         (s (floor
             (- approx
                (* hindu-sidereal-year
                   (mod3 (- (/ (hindu-solar-longitude approx)
                               (deg 360))
                            (/ (1- month) 12))
                         -1/2 1/2)))))
         (k (hindu-lunar-day-from-moment (+ s (hr 6))))
         (est
          (- s (- day)
             (cond
              ((< 3 k 27) ; Not borderline case.
               k)
              ((let* ((mid ; Middle of preceding solar month.
                       (hindu-lunar-from-fixed
                        (- s 15))))
                 (or ; In month starting near s.
                  (/= (hindu-lunar-month mid) month)
                  (and (hindu-lunar-leap-month mid)
                       (not leap-month))))
               (mod3 k -15 15))
              (t ; In preceding month.
               (mod3 k 15 45)))))
         (tau ; Refined estimate.
          (- est (mod3 (- (hindu-lunar-day-from-moment
                           (+ est (hr 6)))
                          day)
                       -15 15)))
         (date (next d (1- tau)
                     (member (hindu-lunar-day-from-moment
                              (hindu-sunrise d))
                             (list day (amod (1+ day) 30)))))))
    (if leap-day (1+ date) date)))
```

(20.25)

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

(20.26)

```
(defconstant hindu-location
  ;; TYPE location
  ;; Location (Ujjain) for determining Hindu calendar.
  ujjain)
```

(20.27)

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

```
        (phi (latitude location))
        (diurnal-radius
         (hindu-sine (+ (deg 90) (hindu-arcsin sin_delta))))
        (tan_phi ; Tangent of latitude as rational number.
         (/ (hindu-sine phi)
            (hindu-sine (+ (deg 90) phi))))
        (earth-sine (* sin_delta tan_phi)))
   (hindu-arcsin (- (/ earth-sine diurnal-radius)))))
```

(20.28)

```
(defun hindu-tropical-longitude (date)
  ;; TYPE fixed-date -> rational-angle
  ;; Hindu tropical longitude on fixed date.
  ;; Assumes precession with maximum of 27 degrees
  ;; and period of 7200 sidereal years
  ;; (= 1577917828/600 days).
  (let* ((days (- date hindu-epoch)) ; Whole days.
         (precession
          (- (deg 27)
             (abs
              (* (deg 108)
                 (mod3 (- (* 600/1577917828 days)
                          1/4)
                       -1/2 1/2))))))
    (mod (- (hindu-solar-longitude date) precession)
         360)))
```

(20.29)

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

(20.30)

```
(defun hindu-daily-motion (date)
  ;; TYPE fixed-date -> rational-angle
  ;; Sidereal daily motion of sun on date.
  (let* ((mean-motion ; Mean daily motion in degrees.
          (/ (deg 360) hindu-sidereal-year))
         (anomaly
          (hindu-mean-position date hindu-anomalistic-year))
         (epicycle ; Current size of epicycle.
          (- 14/360 (/ (abs (hindu-sine anomaly)) 1080)))
         (entry (quotient anomaly (angle 0 225 0)))
         (sine-table-step ; Marginal change in anomaly
          (- (hindu-sine-table (1+ entry))
             (hindu-sine-table entry)))
         (factor
          (* -3438/225 sine-table-step epicycle)))
    (* mean-motion (1+ factor))))
```

(20.31)

```
(defun hindu-rising-sign (date)
  ;; TYPE fixed-date -> rational-amplitude
  ;; Tabulated speed of rising of current zodiacal sign on
  ;; date.
  (let* ((i  ; Index.
          (quotient (hindu-tropical-longitude date)
                    (deg 30))))
    (nth (mod i 6)
         (list 1670/1800 1795/1800 1935/1800 1935/1800
               1795/1800 1670/1800))))
```

(20.32)

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

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

(20.33)

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

(20.34)

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

(20.35)

```
(defun hindu-standard-from-sundial (tee)
  ;; TYPE rational-moment -> rational-moment
  ;; Hindu local time of temporal moment tee.
  (let* ((date (fixed-from-moment tee))
         (time (time-from-moment tee))
         (q (floor (* 4 time))) ; quarter of day
         (a (cond ((= q 0)    ; early this morning
                   (hindu-sunset (1- date)))
                  ((= q 3)    ; this evening
                   (hindu-sunset date))
                  (t ;  daytime today
                   (hindu-sunrise date))))
         (b (cond ((= q 0) (hindu-sunrise date))
                  ((= q 3) (hindu-sunrise (1+ date)))
                  (t (hindu-sunset date)))))
    (+ a (* 2 (- b a) (- time
                         (cond ((= q 3) (hr 18))
                               ((= q 0) (hr -6))
                               (t (hr 6))))))))
```

(20.36)

```
(defun hindu-fullmoon-from-fixed (date)
  ;; TYPE fixed-date -> hindu-lunar-date
  ;; Hindu lunar date, full-moon scheme,
  ;; equivalent to fixed date.
  (let* ((l-date (hindu-lunar-from-fixed date))
         (year (hindu-lunar-year l-date))
         (month (hindu-lunar-month l-date))
         (leap-month (hindu-lunar-leap-month l-date))
         (day (hindu-lunar-day l-date))
         (leap-day (hindu-lunar-leap-day l-date))
         (m (if (>= day 16)
                (hindu-lunar-month
                 (hindu-lunar-from-fixed (+ date 20)))
              month)))
    (hindu-lunar-date year m leap-month day leap-day)))
```

(20.37)

```
(defun fixed-from-hindu-fullmoon (l-date)
  ;; TYPE hindu-lunar-date -> fixed-date
  ;; Fixed date equivalent to Hindu lunar l-date
```

```
  ;; in full-moon scheme.
  (let* ((year (hindu-lunar-year l-date))
         (month (hindu-lunar-month l-date))
         (leap-month (hindu-lunar-leap-month l-date))
         (day (hindu-lunar-day l-date))
         (leap-day (hindu-lunar-leap-day l-date))
         (m (cond ((or leap-month (<= day 15))
                   month)
                  ((hindu-expunged? year (amod (1- month) 12))
                   (amod (- month 2) 12))
                  (t (amod (1- month) 12)))))
    (fixed-from-hindu-lunar
     (hindu-lunar-date year m leap-month day leap-day))))
```

(20.38)

```
(defun hindu-expunged? (l-year l-month)
  ;; TYPE (hindu-lunar-year hindu-lunar-month) ->
  ;; TYPE  boolean
  ;; True of Hindu lunar month l-month in l-year
  ;; is expunged.
  (/= l-month
      (hindu-lunar-month
       (hindu-lunar-from-fixed
        (fixed-from-hindu-lunar
         (list l-year l-month false 15 false))))))
```

(20.39)

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

(20.40)

```
(defun ayanamsha (tee)
  ;; TYPE moment -> angle
  ;; Difference between tropical and sidereal solar longitude.
  (- (solar-longitude tee)
     (sidereal-solar-longitude tee)))
```

(20.41)

```
(defconstant sidereal-start
  ;; TYPE angle
  (precession (universal-from-local
               (mesha-samkranti (ce 285))
               hindu-location)))
```

(20.42)

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

(20.43)

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

(20.44)

```
(defun astro-hindu-calendar-year (tee)
  ;; TYPE moment -> hindu-solar-year
  ;; Astronomical Hindu solar year KY at given moment tee.
  (round (- (/ (- tee hindu-epoch)
               mean-sidereal-year)
            (/ (sidereal-solar-longitude tee)
               (deg 360)))))
```

(20.45)

```
(defun astro-hindu-solar-from-fixed (date)
  ;; TYPE fixed-date -> hindu-solar-date
  ;; Astronomical Hindu (Tamil) solar date equivalent to
  ;; fixed date.
  (let* ((critical    ; Sunrise on Hindu date.
          (astro-hindu-sunset date))
         (month (sidereal-zodiac critical))
         (year (- (astro-hindu-calendar-year critical)
                  hindu-solar-era))
         (approx ; 3 days before start of mean month.
          (- date 3
             (mod (floor (sidereal-solar-longitude critical))
                  (deg 30))))
         (start ; Search forward for beginning...
          (next i approx ; ... of month.
                (= (sidereal-zodiac (astro-hindu-sunset i))
                   month)))
         (day (- date start -1)))
    (hindu-solar-date year month day)))
```

(20.46)

```
(defun fixed-from-astro-hindu-solar (s-date)
  ;; TYPE hindu-solar-date -> fixed-date
  ;; Fixed date corresponding to Astronomical
  ;; Hindu solar date (Tamil rule; Saka era).
  (let* ((month (standard-month s-date))
         (day (standard-day s-date))
         (year (standard-year s-date))
         (approx ; 3 days before start of mean month.
          (+ hindu-epoch -3
             (floor (* (+ (+ year hindu-solar-era)
                          (/ (1- month) 12))
                       mean-sidereal-year))))
         (start ; Search forward for beginning...
          (next i approx ; ... of month.
                (= (sidereal-zodiac (astro-hindu-sunset i))
                   month))))
    (+ start day -1)))
```

(20.47)

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

(20.48)

```
(defun astro-hindu-lunar-from-fixed (date)
  ;; TYPE fixed-date -> hindu-lunar-date
  ;; Astronomical Hindu lunar date equivalent to fixed date.
  (let* ((critical
          (alt-hindu-sunrise date)) ; Sunrise that day.
         (day
          (astro-lunar-day-from-moment critical)); Day of month
         (leap-day              ; If previous day the same.
          (= day (astro-lunar-day-from-moment
                  (alt-hindu-sunrise (- date 1)))))
         (last-new-moon
          (new-moon-before critical))
         (next-new-moon
          (new-moon-at-or-after critical))
         (solar-month         ; Solar month name.
          (sidereal-zodiac last-new-moon))
         (leap-month       ; If begins and ends in same sign.
          (= solar-month (sidereal-zodiac next-new-moon)))
         (month                     ; Month of lunar year.
          (amod (1+ solar-month) 12))
         (year ; Solar year at end of month.
          (- (astro-hindu-calendar-year
              (if (<= month 2) ; date might precede solar
                                         ; new year.
                  (+ date 180)
```

```
                date))
             hindu-lunar-era)))
    (hindu-lunar-date year month leap-month day leap-day)))
```

(20.49)

```
(defun fixed-from-astro-hindu-lunar (l-date)
  ;; TYPE hindu-lunar-date -> fixed-date
  ;; Fixed date corresponding to Hindu lunar date l-date.
  (let* ((year (hindu-lunar-year l-date))
         (month (hindu-lunar-month l-date))
         (leap-month (hindu-lunar-leap-month l-date))
         (day (hindu-lunar-day l-date))
         (leap-day (hindu-lunar-leap-day l-date))
         (approx
          (+ hindu-epoch
             (* mean-sidereal-year
                (+ year hindu-lunar-era
                   (/ (1- month) 12)))))
         (s (floor
             (- approx
                (* hindu-sidereal-year
                   (mod3 (- (/ (sidereal-solar-longitude approx)
                               (deg 360))
                            (/ (1- month) 12))
                         -1/2 1/2)))))
         (k (astro-lunar-day-from-moment (+ s (hr 6))))
         (est
          (- s (- day)
             (cond
              ((< 3 k 27) ; Not borderline case.
               k)
              ((let* ((mid ; Middle of preceding solar month.
                       (astro-hindu-lunar-from-fixed
                        (- s 15))))
                 (or ; In month starting near s.
                  (/= (hindu-lunar-month mid) month)
                  (and (hindu-lunar-leap-month mid)
                       (not leap-month))))
               (mod3 k -15 15))
              (t ; In preceding month.
               (mod3 k 15 45)))))
         (tau ; Refined estimate.
          (- est (mod3 (- (astro-lunar-day-from-moment
                           (+ est (hr 6)))
                          day)
                       -15 15)))
         (date (next d (1- tau)
                     (member (astro-lunar-day-from-moment
                              (alt-hindu-sunrise d))
                             (list day (amod (1+ day) 30))))))
    (if leap-day (1+ date) date)))
```

(20.50)

```
(defun hindu-solar-longitude-at-or-after (lambda tee)
  ;; TYPE (season moment) -> moment
  ;; Moment of the first time at or after tee
  ;; when Hindu solar longitude will be lambda degrees.
  (let* ((tau ; Estimate (within 5 days).
          (+ tee
             (* hindu-sidereal-year (/ 1 (deg 360))
                (mod (- lambda (hindu-solar-longitude tee))
                     360))))
         (a (max tee (- tau 5))) ; At or after tee.
         (b (+ tau 5)))
    (invert-angular hindu-solar-longitude lambda
                    (interval-closed a b))))
```

(20.51)

```
(defun mesha-samkranti (g-year)
  ;; TYPE gregorian-year -> rational-moment
  ;; Fixed moment of Mesha samkranti (Vernal equinox)
  ;; in Gregorian g-year.
```

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

(20.52)

```
(defun hindu-lunar-day-at-or-after (k tee)
  ;; TYPE (rational rational-moment) -> rational-moment
  ;; Time lunar-day (tithi) number k begins at or after
  ;; moment tee.  k can be fractional (for karanas).
  (let* ((phase ; Degrees corresponding to k.
          (* (1- k) (deg 12)))
         (tau ; Mean occurrence of lunar-day.
          (+ tee (* (/ 1 (deg 360))
                    (mod (- phase (hindu-lunar-phase tee))
                         360)
                    hindu-synodic-month)))
         (a (max tee (- tau 2)))
         (b (+ tau 2)))
    (invert-angular hindu-lunar-phase phase
                    (interval-closed a b))))
```

(20.53)

```
(defun hindu-lunar-new-year (g-year)
  ;; TYPE gregorian-year -> fixed-date
  ;; Fixed date of Hindu lunisolar new year in Gregorian
  ;; g-year.
  (let* ((jan1 (gregorian-new-year g-year))
         (mina ; Fixed moment of solar longitude 330.
          (hindu-solar-longitude-at-or-after (deg 330) jan1))
         (new-moon ; Next new moon.
          (hindu-lunar-day-at-or-after 1 mina))
         (h-day (floor new-moon))
         (critical ; Sunrise that day.
          (hindu-sunrise h-day)))
    (+ h-day
       ;; Next day if new moon after sunrise,
       ;; unless lunar day ends before next sunrise.
       (if (or (< new-moon critical)
               (= (hindu-lunar-day-from-moment
                   (hindu-sunrise (1+ h-day))) 2))
           0 1))))
```

(20.54)

```
(defun hindu-lunar-on-or-before? (l-date1 l-date2)
  ;; TYPE (hindu-lunar-date hindu-lunar-date) -> boolean
  ;; True if Hindu lunar date l-date1 is on or before
  ;; Hindu lunar date l-date2.
  (let* ((month1 (hindu-lunar-month l-date1))
         (month2 (hindu-lunar-month l-date2))
         (leap1 (hindu-lunar-leap-month l-date1))
         (leap2 (hindu-lunar-leap-month l-date2))
         (day1 (hindu-lunar-day l-date1))
         (day2 (hindu-lunar-day l-date2))
         (leap-day1 (hindu-lunar-leap-day l-date1))
         (leap-day2 (hindu-lunar-leap-day l-date2))
         (year1 (hindu-lunar-year l-date1))
         (year2 (hindu-lunar-year l-date2)))
    (or (< year1 year2)
        (and (= year1 year2)
             (or (< month1 month2)
                 (and (= month1 month2)
                      (or (and leap1 (not leap2))
                          (and (equal leap1 leap2)
                               (or (< day1 day2)
                                   (and (= day1 day2)
                                        (or (not leap-day1)
                                            leap-day2)))))
                      ))))))
```

(20.55)

```
(defun hindu-date-occur (l-year l-month l-day)
  ;; TYPE (hindu-lunar-year hindu-lunar-month
  ;; TYPE  hindu-lunar-day) -> fixed-date
  ;; Fixed date of occurrence of Hindu lunar l-month,
  ;; l-day in Hindu lunar year l-year, taking leap and
```

```
  ;; expunged days into account.  When the month is
  ;; expunged, then the following month is used.
  (let* ((lunar (hindu-lunar-date l-year l-month false
                                  l-day false))
         (try (fixed-from-hindu-lunar lunar))
         (mid (hindu-lunar-from-fixed
               (if (> l-day 15) (- try 5) try)))
         (expunged? (/= l-month (hindu-lunar-month mid)))
         (l-date ; day in next month
          (hindu-lunar-date (hindu-lunar-year mid)
                            (hindu-lunar-month mid)
                            (hindu-lunar-leap-month mid)
                            l-day false)))
    (cond (expunged?
           (1- (next d try
                     (not
                      (hindu-lunar-on-or-before?
                       (hindu-lunar-from-fixed d) l-date)))))
          ((/= l-day (hindu-lunar-day
                      (hindu-lunar-from-fixed try)))
           (1- try))
          (t try))))
```

(20.56)

```
(defun hindu-lunar-holiday (l-month l-day g-year)
  ;; TYPE (hindu-lunar-month hindu-lunar-day
  ;; TYPE  gregorian-year) -> list-of-fixed-dates
  ;; List of fixed dates of occurrences of Hindu lunar
  ;; month, day in Gregorian year g-year.
  (let* ((l-year (hindu-lunar-year
                  (hindu-lunar-from-fixed
                   (gregorian-new-year g-year))))
         (date0 (hindu-date-occur l-year l-month l-day))
         (date1 (hindu-date-occur (1+ l-year) l-month l-day)))
    (list-range (list date0 date1)
                (gregorian-year-range g-year))))
```

(20.57)

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

(20.58)

```
(defun hindu-tithi-occur (l-month tithi tee l-year)
  ;; TYPE (hindu-lunar-month rational rational
  ;; TYPE  hindu-lunar-year) -> fixed-date
  ;; Fixed date of occurrence of Hindu lunar tithi prior
  ;; to sundial time tee, in Hindu lunar l-month, l-year.
  (let* ((approx
          (hindu-date-occur l-year l-month (floor tithi)))
         (lunar
          (hindu-lunar-day-at-or-after tithi (- approx 2)))
         (try (fixed-from-moment lunar))
         (tee_h (standard-from-sundial (+ try tee) ujjain)))
    (if (or (<= lunar tee_h)
            (> (hindu-lunar-phase
                (standard-from-sundial (+ try 1 tee) ujjain))
               (* 12 tithi)))
        try
      (1+ try))))
```

(20.59)

```
(defun hindu-lunar-event (l-month tithi tee g-year)
  ;; TYPE (hindu-lunar-month rational rational
  ;; TYPE  gregorian-year) -> list-of-fixed-dates
  ;; List of fixed dates of occurrences of Hindu lunar tithi
  ;; prior to sundial time tee, in Hindu lunar l-month,
  ;; in Gregorian year g-year.
  (let* ((l-year (hindu-lunar-year
                  (hindu-lunar-from-fixed
                   (gregorian-new-year g-year))))
         (date0 (hindu-tithi-occur l-month tithi tee l-year))
```

```
          (date1 (hindu-tithi-occur
                  l-month tithi tee (1+ l-year))))
     (list-range (list date0 date1)
                 (gregorian-year-range g-year))))
```

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

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

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

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

(20.64)
```
(defun yoga (date)
  ;; TYPE fixed-date -> 1-27
  ;; Hindu yoga on date.
  (1+ (floor (mod (/ (+ (hindu-solar-longitude date)
                        (hindu-lunar-longitude date))
                     (angle 0 800 0))
                  27))))
```

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

(20.66)
```
(defun sacred-wednesdays-in-range (range)
  ;; TYPE range -> list-of-fixed-dates
  ;; List of Wednesdays within range of dates
  ;; that are day 8 of Hindu lunar months.
  (let* ((a (begin range))
         (b (end range))
         (wed (kday-on-or-after wednesday a))
         (h-date (hindu-lunar-from-fixed wed)))
    (if (in-range? wed range)
        (append
         (if (= (hindu-lunar-day h-date) 8)
             (list wed)
           nil)
         (sacred-wednesdays-in-range
          (interval (1+ wed) b)))
      nil)))
```

D.21 The Tibetan Calendar

```lisp
(defun tibetan-date (year month leap-month day leap-day)
  ;; TYPE (tibetan-year tibetan-month
  ;; TYPE  tibetan-leap-month tibetan-day
  ;; TYPE  tibetan-leap-day) -> tibetan-date
  (list year month leap-month day leap-day))
```

```lisp
(defun tibetan-year (date)
  ;; TYPE tibetan-date -> tibetan-year
  (first date))
```

```lisp
(defun tibetan-month (date)
  ;; TYPE tibetan-date -> tibetan-month
  (second date))
```

```lisp
(defun tibetan-leap-month (date)
  ;; TYPE tibetan-date -> tibetan-leap-month
  (third date))
```

```lisp
(defun tibetan-day (date)
  ;; TYPE tibetan-date -> tibetan-day
  (fourth date))
```

```lisp
(defun tibetan-leap-day (date)
  ;; TYPE tibetan-date -> tibetan-leap-day
  (fifth date))
```

(21.1)

```lisp
(defconstant tibetan-epoch
  ;; TYPE fixed-date
  (fixed-from-gregorian (gregorian-date -127 december 7)))
```

(21.2)

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

(21.3)

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

(21.4)

```lisp
(defun fixed-from-tibetan (t-date)
  ;; TYPE tibetan-date -> fixed-date
  ;; Fixed date corresponding to Tibetan lunar date t-date.
  (let* ((year (tibetan-year t-date))
         (month (tibetan-month t-date))
         (leap-month (tibetan-leap-month t-date))
         (day (tibetan-day t-date))
         (leap-day (tibetan-leap-day t-date))
```

```
          (months ; Lunar month count.
           (floor (+ (* 804/65 (1- year)) (* 67/65 month)
                     (if leap-month -1 0) 64/65)))
          (days ; Lunar day count.
           (+ (* 30 months) day))
          (mean ; Mean civil days since epoch.
           (+ (* days 11135/11312) -30
              (if leap-day 0 -1) 1071/1616))
          (solar-anomaly
           (mod (+ (* days 13/4824) 2117/4824) 1))
          (lunar-anomaly
           (mod (+ (* days 3781/105840) 2837/15120) 1))
          (sun (- (tibetan-sun-equation (* 12 solar-anomaly))))
          (moon (tibetan-moon-equation (* 28 lunar-anomaly))))
     (floor (+ tibetan-epoch mean sun moon))))
```

(21.5)

```
(defun tibetan-from-fixed (date)
  ;; TYPE fixed-date -> tibetan-date
  ;; Tibetan lunar date corresponding to fixed date.
  (let* ((cap-Y (+ 365 4975/18382)) ; Average Tibetan year.
         (years (ceiling (/ (- date tibetan-epoch) cap-Y)))
         (year0 ; Search for year.
          (final y years
                 (>= date
                     (fixed-from-tibetan
                      (tibetan-date y 1 false 1 false)))))
         (month0 ; Search for month.
          (final m 1
                 (>= date
                     (fixed-from-tibetan
                      (tibetan-date year0 m false 1 false)))))
         (est ; Estimated day.
          (- date (fixed-from-tibetan
                   (tibetan-date year0 month0 false 1 false))))
         (day0 ; Search for day.
          (final
           d (- est 2)
           (>= date
               (fixed-from-tibetan
                (tibetan-date year0 month0 false d false)))))
         (leap-month (> day0 30))
         (day (amod day0 30))
         (month (amod (cond ((> day day0) (1- month0))
                            (leap-month (1+ month0))
                            (t month0))
                      12))
         (year (cond ((and (> day day0) (= month0 1))
                      (1- year0))
                     ((and leap-month (= month0 12))
                      (1+ year0))
                     (t year0)))
         (leap-day
          (= date
             (fixed-from-tibetan
              (tibetan-date year month leap-month day true)))))
    (tibetan-date year month leap-month day leap-day)))
```

(21.6)

```
(defun tibetan-leap-month? (t-year t-month)
  ;; TYPE (tibetan-year tibetan-month) -> boolean
  ;; True if t-month is leap in Tibetan year t-year.
  (= t-month
     (tibetan-month
      (tibetan-from-fixed
       (fixed-from-tibetan
        (tibetan-date t-year t-month true 2 false))))))
```

(21.7)

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

```
  ;; month t-month and year t-year.
  (or
   (= t-day
      (tibetan-day
       (tibetan-from-fixed
        (fixed-from-tibetan
         (tibetan-date t-year t-month false t-day true)))))
   ;; Check also in leap month if there is one.
   (= t-day
      (tibetan-day
       (tibetan-from-fixed
        (fixed-from-tibetan
         (tibetan-date t-year t-month
                       (tibetan-leap-month? t-year t-month)
                       t-day true)))))))
```

```
(defun losar (t-year)                                          (21.8)
  ;; TYPE tibetan-year -> fixed-date
  ;; Fixed date of Tibetan New Year (Losar)
  ;; in Tibetan year t-year.
  (let* ((t-leap (tibetan-leap-month? t-year 1)))
    (fixed-from-tibetan
     (tibetan-date t-year 1 t-leap 1 false))))
```

```
(defun tibetan-new-year (g-year)                               (21.9)
  ;; TYPE gregorian-year -> list-of-fixed-dates
  ;; List of fixed dates of Tibetan New Year in
  ;; Gregorian year g-year.
  (let* ((dec31 (gregorian-year-end g-year))
         (t-year (tibetan-year (tibetan-from-fixed dec31))))
    (list-range
     (list (losar (1- t-year))
           (losar t-year))
     (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

Oo

First page of the index to Joseph Scaliger's *De Emendatione Temporum* (Frankfort edition, 1593). (Courtesy of the University of Illinois, Urbana, IL.)