

# A Domain Specific Language (DSL) for Legal Rules

---

## Introduction

### Overview

This notebook defines a package of Wolfram Language functions that facilitates the modeling of legal rules. This is a domain specific language (DSL) - a high-level programming vocabulary tailored to a specific purpose. The larger context of why one would want to do this and numerous other aspects of policy automation are explained elsewhere.

A DSL for legal rules has to do three basic things:

1. **Information-seeking:** Dynamically seek out missing information in order to make a determination;
2. **Proof trees:** Provide justifications for its determinations; and
3. **Time series operators:** Handle certain time series computations.

These features are defined below, along with unit tests that verify their correctness and demonstrate how to use them.

The functions in this library fall into two categories. Some of the functions generalize ordinary Wolfram Language functions to time series operators. For example, the arithmetic operations are generalized to time series objects, so you can add, subtract, multiply, and divide entire time series together, just as you would ordinarily add two scalar quantities (like  $2+2$ ) together. Other functions, while also applying to time series objects, have no counterpart in the Wolfram Language. For example, a group of functions determine how much time has elapsed during which a Boolean time series has been true. So this DSL generalizes a small subset of the Wolfram Language, and also extends it to include certain time-related operations. This DSL also provides functions for easily generalizing a Wolfram Language function to a time series, so that it can be extended even further when necessary, with very little effort.

### Development Conventions

Where a function in this DSL generalizes an existing Wolfram Language function, the naming and syntax generally follow the Wolfram Language conventions. However, this is not always the case. Some function signatures have not been generalized, and occasionally it was not possible for the names to be used consistently. All of this is explained in a separate syntax reference document.

Typically, user-defined Wolfram Language functions start with lower case letters, to avoid conflicts with functions that are part of the core language. However, in this library, function names start with capital letters, for two reasons. First, this ensures that they will stand out from functions defined by users of this DSL. And second, functions that start with upper case letters are omitted from proof trees (see

below). If the function names in this DSL end up conflicting with functions released in future versions of the Wolfram Language, adjustments will have to be made.

## Interface to this Package

This defines the package's interface to the outside world. In other words, these are the functions that are expected to be used to model the legal/policy rules. (Note that overloaded Wolfram Language functions are tentatively not included in this list. Also, many functions that have not yet been defined are not currently included in this list.)

```
BeginPackage["DSL`"];

AlwaysQ::usage = "";
AnnualTimeLine::usage = "";
ApplyRules::usage = "";
Fact::usage = "";
AsOf::usage = "";
Date::usage = "";
DawnOfTime::usage = "";
EndOfTime::usage = "";
EverQ::usage = "";
IfThen::usage = "asdfasdf";
InterviewPopup::usage = "";
InterviewAPI::usage = "";
InterviewWidget::usage = "";
MissingData::usage = "";
Numeric::usage = "";
ProofTree::usage = "";
SwitchThen::usage = "";
TimeLine::usage = "";
TimeLineMap::usage = "";
TimeLinePlot::usage = "";
TrueBefore::usage = "";
TrueBetween::usage = "";
TrueOnOrAfter::usage = "";

Begin["`Private`"];
```

## Background: Modeling Legal Rules

### Facts

Facts are represented as relations (symbolic expressions), with associated values. For example:

```
gender["Mary"] → "Female"
married["Mary", "Tom"] → True
datePurchased["Joan", "Whiteacre"] → DateObject[{2016, 4, 1}]
```

## Rules

Rules derive new facts from existing ones, using `Fact` to mark inputs. Base-level facts are undefined symbols. Example:

```
qualifyingRelativeOf[a_, b_] :=
  age[a] < 18 &&
  age[b] ≥ 18 &&
  Fact[gender[b]] == "Female" &&
  Fact[pregnantQ[b]] &&
  Fact[familyRelationship[a, b]] == "Child";
```

# Information-Seeking

## Identifying Needed Inputs

### Ask for a Fact

The `Fact` function doesn't take any action on its own; it's just a wrapper used to mark inputs, for example `Fact[doB[p]]`.

### ApplyRules

This function applies rules (a list of “goals”) to facts (an association of fact-value pairs). It returns determinations for the goals, a list of any missing facts that are still needed, and an estimate of the progress towards resolving the goals.

In[498]:=

```
ApplyRules[goals_List, facts_Association] :=
  Quiet[Block[{apply, missingData, progress},
    apply = goals //. Normal[KeyMap[Fact[#] &, facts]];
    missingData = DeleteDuplicates[Cases[apply, Fact[_], All] /. Fact[f_] → f];
    progress = N[Length[facts] / Max[Length[facts] + Length[missingData], 1]];

    If[
      missingData == {}, <|"Results" → apply, "MissingData" → {}, "Progress" → 1|>,
      <|"Results" → {}, "MissingData" → missingData, "Progress" → progress|>
    ]
  ]
SetAttributes[ApplyRules, HoldAll];
```

Example:

```

qualifyingRelativeOf[a_, b_] :=
  age[a] < 18 &&
  age[b] ≥ 18 &&
  Fact[gender[b]] == "Female" &&
  Fact[pregnantQ[b]] &&
  Fact[familyRelationship[a, b]] == "Child";

age[p_] := QuantityMagnitude[DateDifference[Fact[doB[p]], Now, "Year"]];

ApplyRules[{qualifyingRelativeOf["Jim", "Tony"]}, <|
  doB["Jim"] → {1902, 3, 3}, doB["Tony"] → {1990, 3, 3}|>]
<|Results → {False}, MissingData → {}, Progress → 1|>

ApplyRules[{qualifyingRelativeOf["Jim", "Tony"]}, <|
  doB["Jim"] → {2002, 3, 3}, doB["Tony"] → {1990, 3, 3}|>]
<|Results → {}, MissingData →
  {gender[Tony], pregnantQ[Tony], familyRelationship[Jim, Tony]}, Progress → 0.4|>

ApplyRules[{qualifyingRelativeOf["Jim", "Tony"]}, <| doB["Tony"] → {1990, 3, 3}|>]
<|Results → {}, MissingData → {doB[Jim], gender[Tony],
  pregnantQ[Tony], familyRelationship[Jim, Tony]}, Progress → 0.2|>

```

## Interactive Interviews

### InterviewPopup

This function creates a pop up interview (in a Wolfram Language notebook) that can be used to test rules interactively. It is recursively driven by the list of missing facts.

In[500]:=

```

InterviewPopup[goals_List, facts_Association] :=
  Quiet[Block[{re, remd, newFactValue},
    re = ApplyRules[goals, facts];
    remd = MissingData[re];

    (* If there's no more missing data, return the evaluated goal. Otherwise,
    ask the next question and resubmit. *)
    If[remd === {}, Return[re],
      newFactValue = Input[ToString[remd[[1]]] <> "?"];
      If[newFactValue === quit, Return[re],
        Return[InterviewPopup[goals, Append[facts, remd[[1]] → newFactValue]]]]]
  ];
SetAttributes[InterviewPopup, HoldFirst];

```

There are no unit tests because this is an interactive feature. Evaluating the code below will trigger an

interview. Typing `quit` will abort it.

```
InterviewPopup[{qualifyingRelationship["Lucy", "Sam"]}, <||>]
{$Canceled > 18 && $Canceled == Female}
```

If you seed `PopupInterview` with facts, those will not be asked:

```
InterviewPopup[{qualifyingRelationship["Lucy", "Sam"]}, <|age["Lucy"] → 26|>]
{True}
```

## Interview API

TODO: Given goals and facts, this function returns information about what is known and what needs to be known in order to reach a determination.

## InterviewWidget

TODO: The following code creates a notebook widget that interactively investigates a given goal.

---

# Time Series Operations

## Overview

### Introduction

As explained elsewhere in great detail, modeling time is a particular challenge in building rule-based systems. The following functions create time series objects and operations that are used extensively in legal rules. A time series is a list of date-value pairs (not necessarily regularly spaced), and time series can be added, subtracted, etc., just like scalar values. So as to avoid conflicts with the `TimeSeries` object as defined in the Wolfram Language, we'll create a new object called `TimeLine` and define a variety of operations on them. The core data structure within a `TimeLine` will be compatible with that of a `TimeSeries`, so we can take advantage of the Wolfram Language's existing `TimeSeries` functions when necessary.

### Beginning and End of Time

There's a need for values that represent the outer limits of a time series. These are not represented as `DateObjects`, even though they are dates, because complex time series operations perform faster this way.

```
In[504]:= DawnOfTime = {2000, 1, 1};
          EndOfTime = {2050, 12, 31};
```

The dawn and end of time expressed in absolute time . This is used for internal `TimeLine` processing.

```
In[506]:= absDawnOfTime = AbsoluteTime[DawnOfTime];
          absEndOfTime = AbsoluteTime[EndOfTime];
```



## TimeLine

### TimeLine

A TimeLine is a list of date-value pairs. Here's an example of a Boolean TimeLine, and its internal representation using AbsoluteTime (the number of seconds since 1900-01-01):

```
MakeTimeLine[{{{2000, 1, 1}, False}, {{2010, 1, 1}, True}}]
```

```
TimeLine[{{3 155 673 600, False}, {3 471 292 800, True}}]
```

```
MakeTimeLine[{{ Sat 1 Jan 2000, False}, { Fri 1 Jan 2010, True}}]
```

```
TimeLine[{{3 155 673 600, False}, {3 471 292 800, True}}]
```

```
ShowTimeLine[TimeLine[{{3 155 673 600, False}, {3 471 292 800, True}}]]
```

```
{{{2000, 1, 1}, False}, {{2010, 1, 1}, True}}
```

This function constructs a TimeLine object.

```
In[508]:= MakeTimeLine[pairs_] := TimeLine[Map[{AbsoluteTime#[[1]]], #[[2]]} &, pairs]]
```

A shorthand way of constructing a new TimeLine:

```
In[509]:= T[ts_] := MakeTimeLine[ts];
```

Convert a TimeLine to a human-readable T object:

```
In[510]:= ShowTimeLine[ts_TimeLine] :=  
  Map[{DateList#[[1]][[1 ;; 3]], #[[2]]} &, TimeLineUnbox[ts]];  
ShowTimeLine[symbol_] := symbol;
```

View a TimeSeries in column form, with DateObjects in the output:

```
In[512]:= TimeLineColumnView[ts_TimeLine] :=  
  Map[{DateObject[DateList#[[1]][[1 ;; 3]]], #[[2]]} &, TimeLineUnbox[ts]] // Column;  
TimeLineColumnView[symbol_] := symbol;
```

```
TimeLineColumnView[TimeLine[{{3 155 673 600, False}, {3 471 292 800, True}}]]
```

```
{ Sat 1 Jan 2000, False}
```

```
{ Fri 1 Jan 2010, True}
```

### Internal Machinery

An internal operation to get data out of a TimeLine object:

```
In[514]:= TimeLineUnbox[t_TimeLine] := t /. TimeLine[innards_] -> innards;
```

Convert a scalar to a TimeLine (internal):

```
In[515]:= ToTimeLine[v_TimeLine] := v;
ToTimeLine[v_] := TimeLine[{{absDawnOfTime, v}}];
```

Extract the dates from one or more time series:

```
In[517]:= TimeLineDates[ts_TimeLine] := TimeLineUnbox[ts][[All, 1]];
TimeLineDates[ts1_TimeLine, ts2_TimeLine] :=
  Union[TimeLineDates[ts1], TimeLineDates[ts2]];
TimeLineDates[scalar_] := DawnOfTime;
```

Extract the values from a time series:

```
In[520]:= TimeLineValues[ts_TimeLine] := TimeLineUnbox[ts][[All, 2]];
TimeLineValues[scalar_] := scalar (* or should this be a list? *)
```

Delete redundant time intervals from a time series. If there's only one interval remains, it returns a scalar. Despite the name, the input is assumed to be a List, not a TimeLine. (Avg. execution time = 0.00023s.)

```
In[522]:= TimeLineTrim[t_List] := Block[{trim},
  trim = Map[#[[1]] &, Split[t, #1[[2]] === #2[[2]] &]];
  If[Length[trim] === 1, trim[[1, 2]], TimeLine[trim]]
];
```

Determine whether something is a TimeLine:

```
In[523]:= TimeLineQ[x_] := Head[x] === TimeLine;
```

Retrieves the value of the first time-value pair in the time series.

```
In[524]:= TimeLineFirstValue[ts_] := TimeLineUnbox[ts][[1]][[2]];

```

Applies a function to the list of all values in a TimeLine (treating them all as a single List and then putting the transformed list back onto the dates of the timeline).

```
In[525]:= TimeLineValuesApply[f_, ts_] :=
  zipDatesValues[TimeLineDates[ts], f[TimeLineValues[ts]]]
```

## AsOf

Get the value of a TimeLine at a particular point in time. AsOf accepts either {y,m,d} or absolute times.

```
In[526]:= AsOf[date_, ts_TimeLine] := QuickAsOf[date, ts];
AsOf[date_, scalar_] := scalar;
```

It also accepts temporal inputs:

```
In[528]:= AsOf[date_TimeLine, val_TimeLine] := TimeLineMap[AsOf[#, val] &, date];
```

An internal method that actually does the lookup:

```
In[529]:= QuickAsOf[date_, ts_TimeLine] :=
  Block[{i, t = TimeLineUnbox[ts], len, abTime = AbsoluteTime[date]},
    len = Length[t];
    For[i = 1, i <= len, i++,
      (* If the date is before the index date,
        return the value of the prior interval *)
      If[abTime < t[[i, 1]], Return[t[[i - 1]][[2]]]];
      (* Else, return the value of the last interval *)
      Return[t[[len]][[2]]];
    ];
  QuickAsOf[date_, scalar_] := scalar;
```

## TimeLineMap

This function maps a scalar function (or functions) onto a time series. For performance reasons, the merging/mapping algorithm varies depending on the nature of the arguments. The order of the following definitions may affect the function's behavior.



```

In[531]:= TimeLineMap[f_, ts_TimeLine] :=
  TimeLineTrim[Map[{#[[1]], f#[[2]]}] &, TimeLineUnbox[ts]]];

TimeLineMap[f_, scalar_ : Except[_Fact]] := f[scalar];

(* Merges two time series like a zipper and
  applies a function to the values in each interval. *)
TimeLineMap[f_, ts1_TimeLine, ts2_TimeLine] :=
  TimeLineMapZip[f[#1, #2] &, ts1, ts2];

(* Map the scalar to the time series and apply the function. *)
TimeLineMap[f_, ts_TimeLine, scalar_ : Except[_Fact]] := TimeLineTrim[
  Map[{#[[1]],
    f#[[2]], scalar]
  } &, TimeLineUnbox[ts]]
];
TimeLineMap[f_, scalar_ : Except[_Fact], ts_TimeLine] := TimeLineTrim[
  Map[{#[[1]],
    f[scalar, #[[2]]]
  } &, TimeLineUnbox[ts]]
];

(* ??? *)
TimeLineMap[f_, scalar1_ : Except[_Fact], scalar2_ : Except[_Fact]] :=
  f[scalar1, scalar2];

```

Utility function

```

In[537]:= toList[val_List] := val;
  toList[val_] := {val};

```

## TimeLineMapZip

Combines two time series like a zipper, which is more efficient than using `Union[TimeLineDates[]]` and then applying `f` to each interval. This function is the same as the core `TimeLineMap`.

```

In[539]:= TimeLineMapZip[f_, ts1_TimeLine, ts2_TimeLine] :=
  Block[{result, t1 = TimeLineUnbox[ts1],
    t2 = TimeLineUnbox[ts2], idx1 = 1, idx2 = 1, val1,
    val2, t1Count, t2Count, nextIdx1, nextIdx2,
    nextDate1, nextDate2, pair, lastPair},

  result = Reap[

```

```

val1 = t1[[1, 2]];
val2 = t2[[1, 2]];
t1Count = Length[t1];
t2Count = Length[t2];

(* Add first time-value pair. Assumes all time series start at dawn! *)
lastPair = f[val1, val2];
(* Sow[{DawnOfTime, lastPair}]; *)
(* Time series starts at earlier of input time series *)
Sow[{Min[{t1[[1, 1]], t2[[1, 1]]}], lastPair}];

(* Walk along the two time series,
index by index, until reaching the end of both. *)
While[idx1 < t1Count || idx2 < t2Count,

(* Don't exceed the length of either list *)
nextIdx1 = Min[idx1 + 1, t1Count];
nextIdx2 = Min[idx2 + 1, t2Count];

(* Get the next change date of each time series *)
nextDate1 = t1[[nextIdx1, 1]];
nextDate2 = t2[[nextIdx2, 1]];

Which[
(* Case 1: If the next dates on each series are the same,
advance along both *)
nextDate1 === nextDate2,
idx1 = nextIdx1;
val1 = t1[[idx1, 2]];
idx2 = nextIdx2;
val2 = t2[[idx2, 2]];
pair = f[val1, val2];
(* Only add a value if it differs from the prior value *)
If[pair != lastPair, Sow[{nextDate1, pair}]; lastPair = pair];,

(* Otherwise, advance along the series that's farther behind,
except when the one that's behind is at its end *)
(* Case 2: Advance time series 1 *)
idx2 === t2Count || (idx1 != t1Count && nextDate2 > nextDate1),
idx1 = nextIdx1;
val1 = t1[[idx1, 2]];
pair = f[val1, val2];

```

```

      If[pair != lastPair, Sow[{nextDate1, pair}]; lastPair = pair;];,

      (* Case3: Advance time series 2 *)
      True,
      idx2 = nextIdx2;
      val2 = t2[[idx2, 2]];
      pair = f[val1, val2];

      If[pair != lastPair, Sow[{nextDate2, pair}]; lastPair = pair;];
    ]; (* End Which *)
  ]; (* End While loop *)

] [[2, 1]]; (* End Reap *)

Return[If[Length[result] == 1, result[[1, 2]], TimeLine[result]]];
];

```

This version is necessary in order to ensure that the IfThen function doesn't return a time series that starts at the DawnOfTime:

```

In[540]:= TimeLineMapZip[f_, ts1_TimeLine, scalar_] :=
  TimeLineMapZip[f, ts1, TimeLine[{{TimeLineUnbox[ts1][[1, 1]], scalar}}]]

```

## Visualizing TimeLines

The following functions visually display TimeLines. Note that TimeLinePlot, defined here, is different from TimeLinePlot, defined in the core Wolfram Language.

```

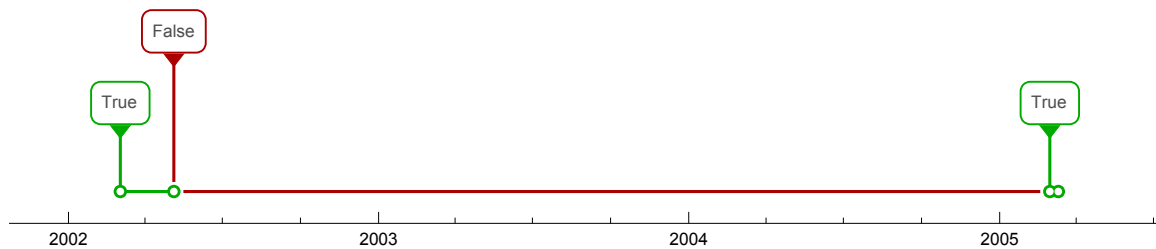
In[541]:= TimeLinePlot[ts_TimeLine] :=
  TimelinePlot[plotData[ts], PlotLayout → "Overlapped"];
TimeLinePlot[list_List] := TimelinePlot[
  plotData[#] & /@ list, PlotLayout → "Overlapped"];

plotData[ts_TimeLine] := intervalStyling[#[[1]], #[[3]], #[[2]]] & /@
  Partition[Flatten[TimeLineUnbox[ts], 1], 3, 2, 1, x]

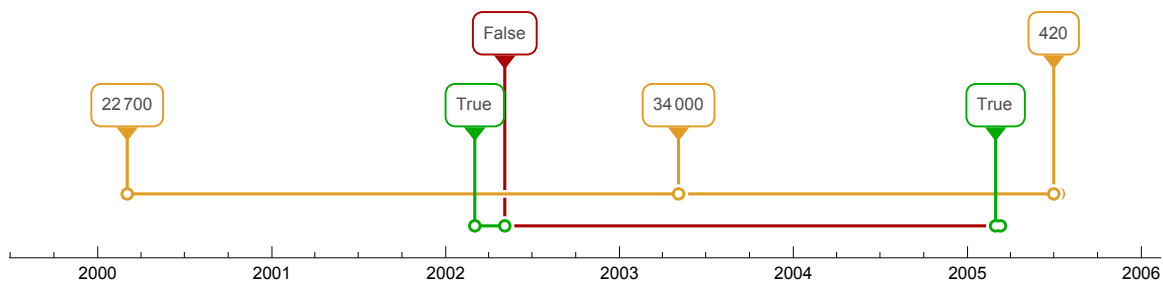
intervalStyling[start_, x, value_] :=
  intervalStyling[start, DatePlus[start, {10, "Day"}], value];
intervalStyling[start_, end_, True] := Style[Labeled[
  Interval[{DateObject[start], DateObject[end]}], True], Darker[Green]];
intervalStyling[start_, end_, False] := Style[
  Labeled[Interval[{DateObject[start], DateObject[end]}], False], Darker[Red]];
intervalStyling[start_, end_, val_] :=
  Labeled[Interval[{DateObject[start], DateObject[end]}], val];

```

```
TimeLinePlot[
  TimeLine[{{{2002, 3, 3}, True}, {{2002, 5, 5}, False}, {{2005, 3, 1}, True}}]]
```



```
TimeLinePlot[
  {TimeLine[{{{2002, 3, 3}, True}, {{2002, 5, 5}, False}, {{2005, 3, 1}, True}}],
  TimeLine[{{{2000, 3, 3}, 22700}, {{2003, 5, 5}, 34000}, {{2005, 7, 1}, 420}}]}]
```



## TimeLineWindow

Extracts a subset of a TimeLine. The input dates can be in the {y,m,f} format or in absolute times.

In[548]:=

```
TimeLineWindow[t_, start_List, end_List] :=
  TimeLineWindow[t, AbsoluteTime[start], AbsoluteTime[end]];
TimeLineWindow[t_, start_, end_] := TimeLine[Union[
  {{start, AsOf[start, t]}},
  Select[TimeLineUnbox[t], start < #[[1]] < end &]
]];
```

## TimeLineGenerator

Using the date values of an existing TimeLine, this function applies a generating function to establish the new values for those dates.

In[550]:=

```
TimeLineGenerator[timeLine_, fcn_] := Block[{dates},
  dates = TimeLineDates[timeLine];
  zipDatesValues[dates, Array[fcn, Length[dates]]]
];
SetAttributes[TimeLineGenerator, HoldAll];
```

```
TimeLineGenerator[TheYear, #^2 &]
```

## Helper Functions

Convert a DateObject to {y,m,d} format:

```
In[552]:= toDateList[date_DateObject] := DateList[date][[1 ;; 3]]
```

Zip up a list of dates and a list of values to form a new timeline:

```
In[553]:= zipDatesValues[dates_, vals_] := TimeLine[Transpose[{dates, vals}]]
```

Determines whether the first interval in a time series represents the dawn of time:

```
In[554]:= timeLineStartsAtDawn[ts_] := TimeLineUnbox[ts][[1]][[1]] == absDawnOfTime;
```

## TimeLines that Change Annually

Creates a TimeLine starting on a given date, with values that change annually thereafter per a given list. TODO

```
YearlyValue[startDate_, list_List] := TimeLine[
  Insert[Transpose[{Map[{#, 1, 1} &,
    Range[startDate, startDate + Length[list] - 1]], list}],
    {DawnOfTime, Null}, {1}]
];
```

## Boolean TimeLines

### Boolean Operators

Ordinary logical operators, but they apply to time series objects rather than Boolean objects.

```

In[555]:= and[p_, q_] := TimeLineMap[And, p, q];
Unprotect[And];
And[False, a_] = False;
And[a_, False] = False;
And[a_, b_TimeLine] := and[a, b];
Protect[And];

or[p_, q_] := TimeLineMap[Or, p, q];
Unprotect[Or];
Or[True, a_] = True;
Or[a_, True] = True;
Or[a_, b_TimeLine] := or[a, b];
Protect[Or];

not[p_] := TimeLineMap[!#&, p];
Unprotect[Not];
Not[a_TimeLine] := not[a];
Protect[Not];

```

## Constructing Boolean TimeLines

An easy way to construct time series whose truth value changes on a particular date (or dates).

Creates a Boolean time series that's false before a given date, and true thereafter.

```

In[571]:= TrueOnOrAfter[date_] := If[date === DawnOfTime,
    True, MakeTimeLine[{{DawnOfTime, False}, {date, True}}]];

```

Creates a Boolean time series that's true before a given date, and false thereafter.

```

In[572]:= TrueBefore[date_] := If[date === DawnOfTime,
    False, MakeTimeLine[{{DawnOfTime, True}, {date, False}}]];

```

Creates a Boolean time series that's true between two dates, and otherwise false.

```

In[573]:= TrueBetween[date1_, date2_] := and[TrueOnOrAfter[date1], TrueBefore[date2]];

```

Creates a Boolean time series that's true on a given date, and otherwise false.

```

In[574]:= TrueOn[date_] := TrueBetween[date, AddDays[date, 1]];

```

## Ever or Always True

Does a time series ever have a particular value?

```
In[575]:= EverQ[ts_TimeLine, val_] := MemberQ[TimeLineValues[ts], val];
EverQ[symbol_, val_] := symbol == val;
EverQ[ts_] := EverQ[ts, True];
```

Does a time series always have a particular value?

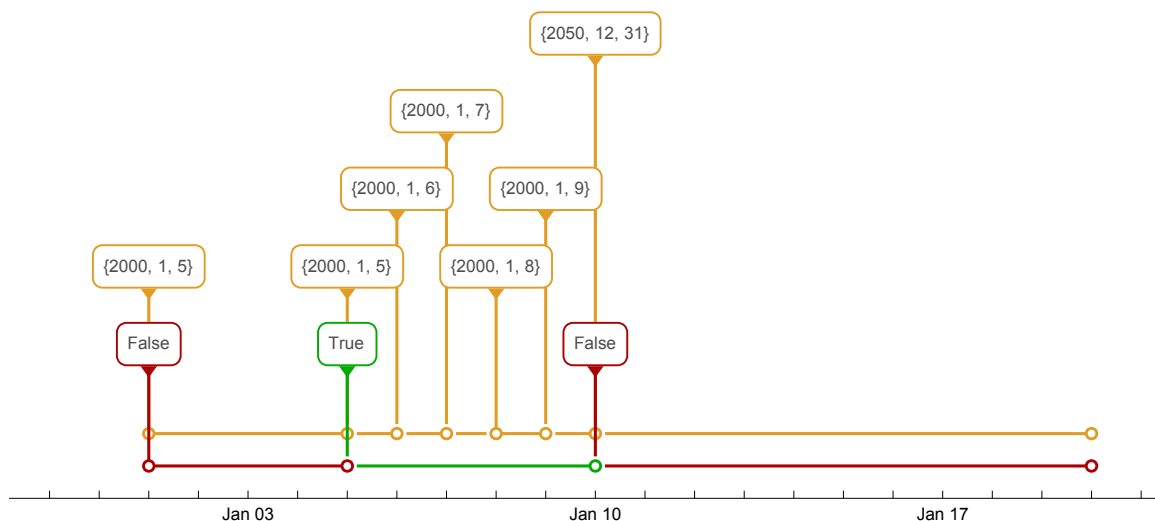
```
In[578]:= AlwaysQ[ts_TimeLine, val_] := Block[{vals = TimeLineValues[ts]},
  and[Length[vals] == 1, MemberQ[vals, val]]]; (* TODO: rewrite *)
AlwaysQ[symbol_, val_] := symbol == val;
AlwaysQ[ts_] := AlwaysQ[ts, True];
```

## When Next or Last True

At any point in time, when will a Boolean time series next be true? If, at a given point in time, the TimeLine is true, the value for that interval is TheDate. TODO: handle scalars, test, performance, and prevent TimeLines as arguments.

```
In[581]:= DateNextTrue[timeLine_] := Block[{currentlyTrue, vals, newVals},
  currentlyTrue = IfThen[timeLine, TheDate, x];
  (* Assign TheDate to True intervals. Slow b/c it uses TheDate. *)
  vals = TimeLineValues[currentlyTrue];
  If[Last[vals] == x, vals[[-1]] = EndOfTime]; (* Deal with last interval *)
  newVals = Map[If[#[[1]] == x, #[[2]], #[[1]]] &, Partition[vals, 2, 1, 1, x]];
  (* Get date of next True interval *)
  zipDatesValues[TimeLineDates[currentlyTrue], newVals]
]
```

```
testval = TrueOnOrAfter[{2000, 1, 5}] && TrueBefore[{2000, 1, 10}];
TimeLinePlot[{testval, DateNextTrue[testval]}]
```



At any point in time, when was a Boolean time series last true?

```
In[582]:= DateLastTrue[timeLine_] := DawnOfTime;
```

## Conditionals

### Usage Note

The following conditional functions implement time series versions of `If`, `Which`, and `Switch`. Due to the difficulty of filtering function definitions so as to cleanly separate the time series version from the core Wolfram Language version, the conditional functions have names that diverge from the Wolfram Language. (Conditionals are delicate since, if implemented carelessly, they can easily spin out into infinite loops.)

### InternalIf

The if-then-else function for time series, used only within this package. This evaluates lazily to permit temporal recursion.

```
In[583]:= InternalIf[True, val1_, val2_] := val1;
InternalIf[False, val1_, val2_] := val2;
InternalIf[test_, val1_, val2_] :=
  Block[{first = TimeLineMap[ifTrueMap, test, val1]},
    If[!EverQ[first, False], first, TimeLineMapZip[ifFalseMap, first, val2]]
  ];
SetAttributes[InternalIf, HoldRest]

(* Broken out to make if[] slightly easier to read. *)
ifTrueMap[val1_, val2_] := If[val1, val2, val1];
ifFalseMap[val1_, val2_] := If[val1 === False, val2, val1];
```

### IfThen

This is the externally exposed function that implements the time series version of `If` and `Which`. Note that, unlike `Which`, the last argument is the default value.

```
In[589]:= IfThen[test1_, val1_, args___, def_] :=
  InternalIf[test1, val1, IfThen[args, def]];
IfThen[def_] := def;
SetAttributes[IfThen, HoldRest];
```

### SwitchThen

This is the time series version of `Switch`. Note that it's different than `Switch` because here the last argument is the default value.



```
In[592]:= SwitchThen[test_List, t1_, v1_, args___, def_] :=
  InternalIf[listEqual[t1, test], v1, SwitchThen[test, args, def]];
SwitchThen[test_, t1_, v1_, args___, def_] :=
  InternalIf[t1 == test, v1, SwitchThen[test, args, def]];
SwitchThen[irrelevantTest_, def_] := def;
SetAttributes[SwitchThen, HoldRest];
```

Test for the equality of two lists:

```
In[596]:= listEqual[{a1_, a2_}, {b1_, b2_}] := a1 == b1 && a2 == b2;
listEqual[a_List, b_List] := a == b;
```

## Arithmetic

### Implementation Details

The repetitive declaration structures below are necessary because using signatures like `f[a_TimeLine, b_TimeLine]` somehow causes interference with the `Times` operator. This exhibited itself when evaluating the expression `DateDifference[{2000, 1, 1}, {2017, 1, 2}]` immediately after initializing this notebook. Overloading `Plus` and `Times` is not recommended. See <http://stackoverflow.com/questions/5252179/how-to-overload-times-and-plus-for-matrix-multiplication-in-mathematica>. An alternative, if problems persist, would be to use `CircleTimes`, etc.:

```
CircleTimes[a_TimeLine, b_TimeLine] := TimeLineMap[Times, a, b];
CircleTimes[a_TimeLine, b_] := TimeLineMap[Times, a, b];
CircleTimes[a_, b_TimeLine] := TimeLineMap[Times, a, b];
CircleTimes[a_, b_] := a * b;
```

```
7⊗8
```

```
56
```

```
8⊗MakeTimeLine[{{DawnOfTime, 5}, {{2015, 2, 2}, 9}}]
```

```
TimeLine[{{3 155 673 600, 40}, {3 631 824 000, 72}}]
```

### Addition & Subtraction

Note: The order of these overloaded definitions seems to affect performance (or did in a previous implementation from which this was adapted).

Addition & subtraction:

```
In[598]:= Unprotect[Plus];
Plus[a_TimeLine, b_TimeLine] := TimeLineMap[Plus, a, b];
Plus[a_TimeLine, b_] := TimeLineMap[Plus, a, b];
Plus[a_, b_TimeLine] := TimeLineMap[Plus, a, b];
Protect[Plus];
```

From older implementation: Possibly add the code below to the definition of `Plus`. Without this, interview question order sometimes gets rearranged. However, `Orderless` only remains cleared in `Akkadian.nb` upon first initialization. Upon the second evaluation of `Attributes[Plus]`, `Orderless` is mysteriously back.

```
ClearAttributes[Plus, Orderless];
```

## Multiplication

Multiplication.

```
In[603]:= Unprotect[Times];
Times[a_TimeLine, b_TimeLine] := TimeLineMap[Times, a, b];
Times[a_TimeLine, b_] := TimeLineMap[Times, a, b];
Times[a_, b_TimeLine] := TimeLineMap[Times, a, b];
Protect[Times];
```

## Division

Division. Because the Wolfram Language represents  $a/b$  as `Times[a, Power[b, -1]]`, the `Power` function has to be overloaded as well.

```
In[608]:= Unprotect[Power];
Power[t_TimeLine, -1] := TimeLineMap[1/# &, t];
Protect[Power];
```

## Numeric

Numerical form. Like `N[]`, this converts fractions to decimal form.

```
In[611]:= Numeric[ts_] := TimeLineMap[N, ts];
```

## Modulo

Gives the remainder on division of  $m$  by  $n$ :

```
In[612]:= Modulo[m_, n_] := TimeLineMap[Mod, m, n];
```

## Rounding

Round to the nearest multiple (ties broken by rounding toward the even alternative):

```
In[613]:= Unprotect[Round];
Round[a_TimeLine, b_TimeLine] := TimeLineMap[Round, a, b];
Round[a_TimeLine, b_] := TimeLineMap[Round, a, b];
Round[a_, b_TimeLine] := TimeLineMap[Round, a, b];
Protect[Round];
```

Ceiling (round up to the nearest multiple):

```
In[618]:= Unprotect[Ceiling];
Ceiling[a_TimeLine, b_TimeLine] := TimeLineMap[Ceiling, a, b];
Ceiling[a_TimeLine, b_] := TimeLineMap[Ceiling, a, b];
Ceiling[a_, b_TimeLine] := TimeLineMap[Ceiling, a, b];
Protect[Ceiling];
```

Floor (round down to the nearest multiple):

```
In[623]:= Unprotect[Floor];
Floor[a_TimeLine, b_TimeLine] := TimeLineMap[Floor, a, b];
Floor[a_TimeLine, b_] := TimeLineMap[Floor, a, b];
Floor[a_, b_TimeLine] := TimeLineMap[Floor, a, b];
Protect[Floor];
```

## Comparison

Comparison operators. These also perform date comparison on {y,m,d} objects.

Greater than.

```
In[628]:= Unprotect[Greater];
Greater[a_TimeLine, b_TimeLine] := TimeLineMap[Greater, a, b];
Greater[a_TimeLine, b_] := TimeLineMap[Greater, a, b];
Greater[a_, b_TimeLine] := TimeLineMap[Greater, a, b];
Greater[{ay_, am_, ad_}, {by_, bm_, bd_}] :=
  AbsoluteTime[{ay, am, ad}] > AbsoluteTime[{by, bm, bd}];
Protect[Greater];
```

Greater than or equal to.

```
In[634]:= Unprotect[GreaterEqual];
GreaterEqual[a_TimeLine, b_TimeLine] := TimeLineMap[GreaterEqual, a, b];
GreaterEqual[a_TimeLine, b_] := TimeLineMap[GreaterEqual, a, b];
GreaterEqual[a_, b_TimeLine] := TimeLineMap[GreaterEqual, a, b];
GreaterEqual[{ay_, am_, ad_}, {by_, bm_, bd_}] :=
  AbsoluteTime[{ay, am, ad}] ≥ AbsoluteTime[{by, bm, bd}];
Protect[GreaterEqual];
```

Less than.

```
In[640]:= Unprotect[Less];
Less[a_TimeLine, b_TimeLine] := TimeLineMap[Less, a, b];
Less[a_TimeLine, b_] := TimeLineMap[Less, a, b];
Less[a_, b_TimeLine] := TimeLineMap[Less, a, b];
Less[{ay_, am_, ad_}, {by_, bm_, bd_}] :=
  AbsoluteTime[{ay, am, ad}] < AbsoluteTime[{by, bm, bd}];
Protect[Less];
```

Less than or equal to.

```
In[646]:= Unprotect[LessEqual];
LessEqual[a_TimeLine, b_TimeLine] := TimeLineMap[LessEqual, a, b];
LessEqual[a_TimeLine, b_] := TimeLineMap[LessEqual, a, b];
LessEqual[a_, b_TimeLine] := TimeLineMap[LessEqual, a, b];
LessEqual[{ay_, am_, ad_}, {by_, bm_, bd_}] :=
  AbsoluteTime[{ay, am, ad}] ≤ AbsoluteTime[{by, bm, bd}];
Protect[LessEqual];
```

Equal to.

```
In[652]:= Unprotect[Equal];
Equal[a_TimeLine, b_TimeLine] := TimeLineMap[Equal, a, b];
Equal[a_TimeLine, b_] := TimeLineMap[Equal, a, b];
Equal[a_, b_TimeLine] := TimeLineMap[Equal, a, b];
Equal[{ay_, am_, ad_}, {by_, bm_, bd_}] :=
  AbsoluteTime[{ay, am, ad}] == AbsoluteTime[{by, bm, bd}];
Protect[Equal];
```

Not equal to.

```
In[658]:= Unprotect[Unequal];
Unequal[a_TimeLine, b_TimeLine] := TimeLineMap[Unequal, a, b];
Unequal[a_TimeLine, b_] := TimeLineMap[Unequal, a, b];
Unequal[a_, b_TimeLine] := TimeLineMap[Unequal, a, b];
Unequal[{ay_, am_, ad_}, {by_, bm_, bd_}] :=
  AbsoluteTime[{ay, am, ad}] != AbsoluteTime[{by, bm, bd}];
Protect[Unequal];
```

## Date Operations

### Composition

At least one of the inputs must be a scalar. (Need TimeLineMap to handle three inputs in order to

implement this fully.)

```
In[664]:= MakeDate[y_TimeLine, m_ , d_] := TimeLineMap[{#, m, d} &, y];
MakeDate[y_TimeLine, m_TimeLine, d_] := TimeLineMap[{#1, #2, d} &, y, m];
MakeDate[y_TimeLine, m_ , d_TimeLine] := TimeLineMap[{#1, m, #2} &, y, d];
MakeDate[y_ , m_TimeLine, d_] := TimeLineMap[{y, #, d} &, m];
MakeDate[y_ , m_TimeLine, d_TimeLine] := TimeLineMap[{y, #1, #2} &, m, d];
MakeDate[y_ , m_ , d_TimeLine] := TimeLineMap[{y, m, #} &, d];
MakeDate[y_ , m_ , d_] := {y, m, d};
```

## Decomposition

Extract the components of a (possibly temporal) date:

```
In[670]:= GetYear[date_] := TimeLineMap[#[[1]] &, date];
GetMonth[date_] := TimeLineMap[#[[2]] &, date];
GetDay[date_] := TimeLineMap[#[[3]] &, date];
```

## Differences Between Dates

These functions do not follow the naming pattern of their counterparts in the Wolfram Language.

Gives the number of days between two dates (without units of measure):

```
In[673]:= DayDiff[a_ , b_] :=
  TimeLineMap[QuantityMagnitude[DateDifference[#1, #2]] &, a, b];
```

Gives the number of weeks between two dates (without units of measure):

```
In[674]:= WeekDiff[a_ , b_] :=
  TimeLineMap[QuantityMagnitude[DateDifference[#1, #2, "Week"]] &, a, b];
```

Gives the number of months between two dates (without units of measure):

```
In[675]:= MonthDiff[a_ , b_] :=
  TimeLineMap[QuantityMagnitude[DateDifference[#1, #2, "Month"]] &, a, b];
```

Gives the number of years between two dates (without units of measure):

```
In[676]:= YearDiff[a_ , b_] :=
  TimeLineMap[QuantityMagnitude[DateDifference[#1, #2, "Year"]] &, a, b];
```

## A Date Plus Some Interval

These functions do not follow the naming pattern of their counterparts in the Wolfram Language.

```
In[677]:= AddDays[date_ , n_] := TimeLineMap[DatePlus[#1, #2] &, date, n];
```

```
In[678]:= AddWeeks[date_, n_] := TimeLineMap[DatePlus[#1, {#2, "Week"}] &, date, n];
```

```
In[679]:= AddMonths[date_, n_] := TimeLineMap[DatePlus[#1, {#2, "Month"}] &, date, n];
```

```
In[680]:= AddYears[date_, n_] := TimeLineMap[DatePlus[#1, {#2, "Year"}] &, date, n];
```

## List Operations

These are named differently than their counterparts in the Wolfram Language to avoid conflicts. Maybe name these `List___` to standardize them and clarify that they apply to temporal lists rather than entire time lines per se?

### Length

Number of items in a (temporal) list:

```
In[681]:= Len[ts_] := TimeLineMap[Length, ts];
```

### Minimum & Maximum

Minimum value in a (temporal) list:

```
In[682]:= Minimum[ts_] := TimeLineMap[Min, ts];
```

Maximum value in a (temporal) list:

```
In[683]:= Maximum[ts_] := TimeLineMap[Max, ts];
```

Ensures that a value remains in between specified minimum and maximum bounds. The use case is not clear for this.

```
Bounded[minimum_, val_, maximum_] := Maximum[Minimum[val, maximum], minimum]
```

### Total

Total of values in a (temporal) list:

```
In[684]:= Tot[ts_] := TimeLineMap[Total, ts];
```

### MemberQ

Is an item a member of a temporal list?

```
In[685]:= ListMemberQ[tsList_, tsMember_] := TimeLineMap[MemberQ, tsList, tsMember];
```

## Higher-Order Functions that Manipulate Lists

Maps a function to the elements of a temporal list.

```
In[686]:= ListMap[f_, ts_] := TimeLineMap[Map[f, #] &, ts];
```

Determines whether all members of a temporal set (a list) meet a certain test.

```
In[687]:= ListForAll[ts_, f_] := TimeLineMap[Total[Boole[Map[f, #]]] == Length[#] &, ts];
```

Determines whether any members of a temporal set (a list) meet a certain test.

```
In[688]:= ListExists[ts_, f_] := TimeLineMap[Total[Boole[Map[f, #]]] > 0 &, ts];
```

Filters the elements of a temporal set that meet a certain test.

```
In[689]:= ListSelect[ts_, f_] := TimeLineMap[Select[#, f] &, ts];
```

## Other

TODO:

Union

Intersection

ContainsAny

ContainsAll

## All-Time High or Low

### High / Low

What is the all-time maximum value of a TimeLine?

```
In[690]:= High[ts_] := Max[TimeLineValues[ts]]
```

What is the all-time minimum value of a TimeLine?

```
In[691]:= Low[ts_] := Min[TimeLineValues[ts]]
```

## Interval Count Relative to a Point in Time

### YearsSince

Creates a timeline that counts years until/from a given date. Essentially, this returns a number line that has the interval of a year, with the origin located at the input date. Date input is assumed not to be a time series. Avg. execution time = 0.0002s when time range is 70 years.

```
In[692]:= YearsSince[date_] :=
  Block[{ts = TimeLine[Table[{{i, date[[2]], date[[3]]}, i - date[[1]]],
    {i, DawnOfTime[[1]], EndOfTime[[1]]}]],
    If[timeLineStartsAtDawn[ts], ts,
      TimeLine[Insert[TimeLineUnbox[ts],
        {absDawnOfTime, TimeLineFirstValue[ts] - 1}, {1}]]
  ];
```

## DaysSince

Creates a timeline that counts days until/from a given date. Date input is assumed not to be a time series. Avg. execution time = 0.007s when time range is 70 years.

```
In[693]:= DaysSince[date_] :=
  Block[{start = QuantityMagnitude[DateDifference[DawnOfTime, date]] * -1,
    diff = Length[absoluteTimeDates] - 1},
    zipDatesValues[absoluteTimeDates, Range[start, start + diff]]
  ];
```

## MonthsSince

## WeeksSince

Counts weeks until/from a given date:

```
In[694]:= WeeksSince[date_] := Floor[DaysSince[date] / 7];
```

## IntervalsSince

IN PROGRESS. This creates a number line with the origin at a particular point in time:

```
IntervalsSince[origin_, interval_] := TimeLineGenerator[interval, #^2 &];
```

```
IntervalsSince[{2017, 1, 1}, TheYear]
```

```
Array[# - 20 &, 50]
```

```
{-19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8,
 -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
```

## Absolute Time

### Internal Machinery

List of dates from the dawn of time to the end of time. This exists for performance reasons, as an internal representation of absolute time; it is not expected to be referenced externally.



```
In[695]:= allDates = DateRange[DawnOfTime, EndOfTime];
absoluteTimeDates = Map[AbsoluteTime, allDates];
```

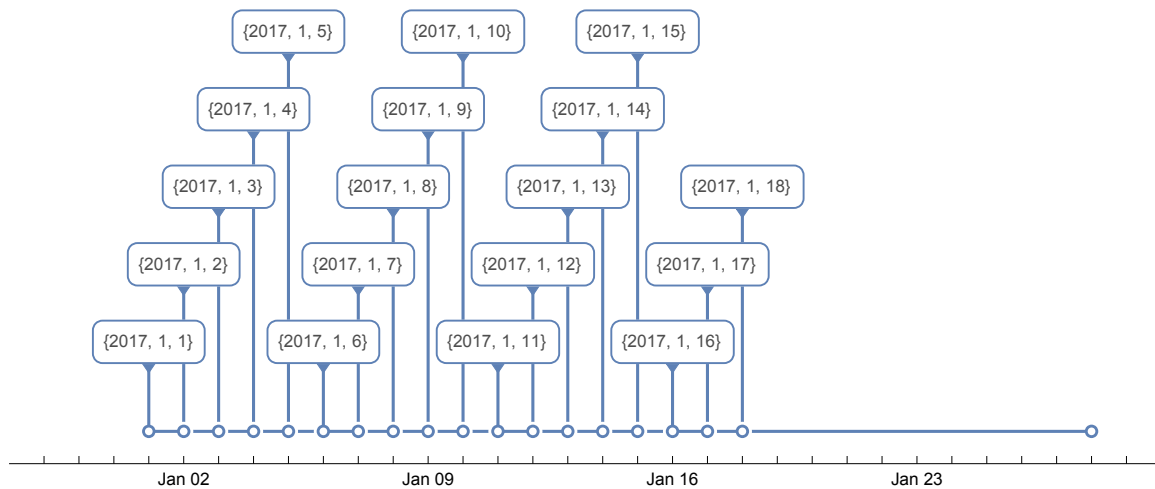
Because the following variables are set using `=` instead of `:=`, they must be ordered in the notebook so dependencies are evaluated first (i.e. be further up the page).

## Calendar-Based TimeLines

A time series in which each day has the value of the date itself. For example, on July 11, 2034, the time series value is `{2034,7,11}`.

```
In[697]:= TheDate = TimeLine[Transpose[{absoluteTimeDates, allDates}]];
```

```
TimeLinePlot[TimeLineWindow[TheDate, {2017, 1, 1}, {2017, 1, 19}]]
```



```
In[698]:= TheMonth = GetMonth[TheDate];
```

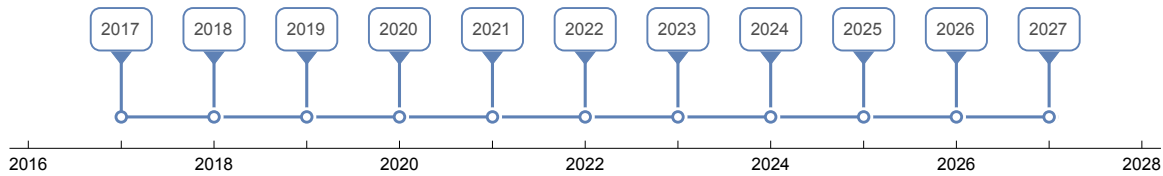
```
In[699]:= TheDayOfTheMonth = GetDay[TheDate];
```

```
In[700]:= TheCalendarQuarter = TimeLineMap[Which[
  # < 4, 1,
  # < 7, 2,
  # < 10, 3,
  True, 4] &,
  TheMonth];
```

A time series representing the calendar year.

```
In[701]:= TheYear = GetYear[TheDate];
```

```
TimeLinePlot[TimeLineWindow[TheYear, {2017, 1, 1}, {2027, 1, 19}]]
```



```
In[702]:= LeapYrQ = TimeLineMap[LeapYearQ[{#}] &, TheYear];
```

TODO:

TheCalendarWeek =

## First/Last Day of Interval

Time series representing the first and last day of every month. It may be the case in practice that a function like `firstDayOfMonth[date]` does the trick and that these are unnecessary.

```
In[703]:= FirstDayOfMonth = MakeDate[TheYear, TheMonth, 1];
```

```
In[704]:= LastDayOfMonth = AddDays[AddMonths[FirstDayOfMonth, 1], -1];
```

## Interval Ratios

```
In[705]:= DaysPerYear = TimeLineMap[If[LeapYrQ[{#}], 366, 365] &, TheYear];
```

```
In[706]:= DaysPerMonth = GetDay[LastDayOfMonth];
```

TODO:

DaysPerCalendarQuarter =

```
In[707]:= WeeksPerMonth = Numeric[DaysPerMonth / 7];
```

## Weekdays

This is an internal helper function. Even though ISO 8601 considers Monday as the first day of the week, the U.S. follows a different convention:

```
In[708]:= DayNameToNumber = <|Sunday → 1, Monday → 2,  
    Tuesday → 3, Wednesday → 4, Thursday → 5, Friday → 6, Saturday → 7|>;
```

Day of the week (as a temporal number):

```
In[709]:= DayOfWeek = zipDatesValues[TimeLineDates[TheDate],
  Array[Mod[#, 7] + 1 &, Length[absoluteTimeDates],
    DayNameToNumber[DateValue[DawnOfTime, "DayName"]] - 1]];

```

Is the day a weekday?

```
In[710]:= WeekdayQ = ! (DayOfWeek == 1 || DayOfWeek == 7);

```

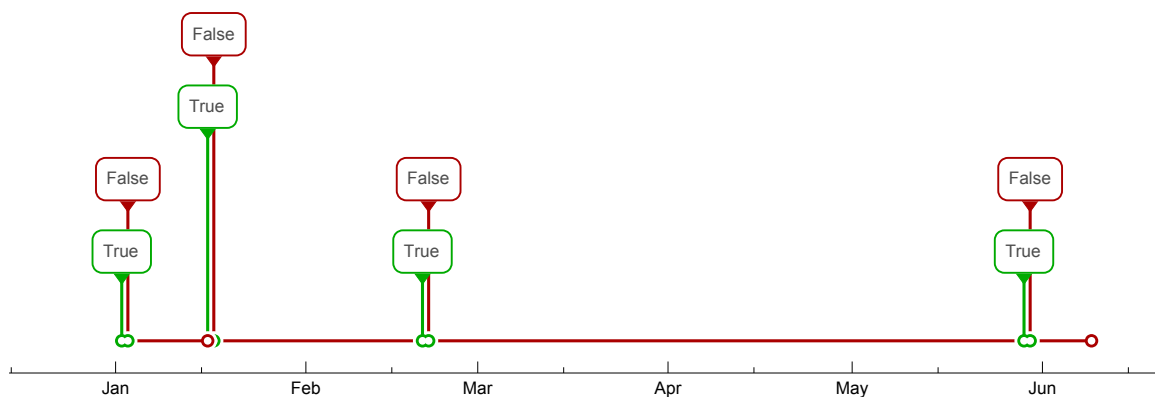
Is the day a U.S. Federal holiday? Note: The legal authority for this is 5 U.S.C. 6103, but we're relying on the Wolfram Language HolidayCalendar implementation. This implementation currently (Sept. 2016) has a bug that fails to recognize Veterans Days that fall on a Saturday but that are observed on the preceding Friday. But has been reported and assuming it will eventually get fixed.

```
In[711]:= USFederalHolidayQ = Block[{holidays, firstVal},
  holidays = DayRange[DawnOfTime, EndOfTime, "Holiday"];
  firstVal = {{AbsoluteTime[DatePlus[DawnOfTime, -1]], False}};
  TimeLine[Union[firstVal, Map[{AbsoluteTime[#], True} &, holidays],
    Map[{AbsoluteTime[#], False} &, DatePlus[holidays, 1]]]
];

```

```
TimeLinePlot[TimeLineWindow[USFederalHolidayQ, {2017, 1, 1}, {2017, 6, 1}]]

```



```
In[712]:= BizDayQ = WeekdayQ && ! USFederalHolidayQ;

```

```
In[713]:= NextBusinessDay = DateNextTrue[BizDayQ];

```

## Per Interval

### Machinery

For internal use: For each interval in a TimeLine, returns the beginning and end date (in absolute time). Result is a List.

```
In[714]:= TimeLineDateSpans[ts_] :=
  Partition[TimeLineDates[ts], 2, 1, 1, x] /. x → absEndOfTime;
```

Apply a function to a TimeLine per some other interval:

```
In[715]:= ApplyPerInterval[f_, ts_, interval_] :=
  TimeLineTrim[Map[{#[[1]], f[TimeLineWindow[ts, #[[1]], #[[2]]]]} &,
    TimeLineDateSpans[interval]]];
```

## All Values in a Subinterval

For each interval in a larger time period, gather all of the time series values of a TimeLine with smaller intervals. This is pretty slow.

```
In[716]:= GatherValuesPer[ts_, interval_] := ApplyPerInterval[TimeLineValues, ts, interval]
```

## EverPer / AlwaysPer

Is a Boolean TimeLine (ts) ever true in a time interval?

```
In[717]:= EverPer[ts_, interval_] := ApplyPerInterval[EverQ, ts, interval]
```

Is a Boolean TimeLine (ts) always true in a time interval?

```
In[718]:= AlwaysPer[ts_, interval_] := ApplyPerInterval[AlwaysQ, ts, interval]
```

## Is a Given Date in an Interval?

In which interval (of a given type) does a given date fall?

```
In[719]:= DateInPeriodQ[date_, interval_] := EverPer[TrueOn[date], interval];
```

## Number of Subintervals in a Larger Interval

Count how many subintervals there are in each larger interval:

```
In[720]:= SubintervalsPer[smaller_, larger_] := Len[GatherValuesPer[smaller, larger]];
```

## Date of Next Interval

For each interval in a TimeLine, returns the date that that the next interval begins.

```
In[721]:= DateOfNextInterval[ts_] := TimeLineTrim[
  Map[{#[[1]], toDateList[DateObject[#[[2]]]]} &, TimeLineDateSpans[ts]]];
```

## Value at End of Interval

What is the value of a TimeLine at the end of each reference interval? (For example, how old is a person at the end of each calendar quarter?) The end is taken to be one second before the start of the next interval.

In[722]:=

```
IntervalEndValue[ts_, interval_] :=
  TimeLineTrim[Map[#[[1]], AsOf[#[[2]] - 1, ts]] &, TimeLineDateSpans[interval]]];
```

## Other

SubintervalCountPer

RunningCountPer

## Elapsed Time

### Elapsed Days Per Interval

Returns the number of days a Boolean TimeLine is true in each of the given intervals. TODO: Optimize (currently takes 6-8 ms).

In[723]:=

```
ElapsedDaysPer[False, interval_] = 0;
ElapsedDaysPer[ts_, interval_] :=
  Block[{dates = TimeLineDates[interval], result = {}, i},
    For[i = 1, i <= Length[dates] - 1, i++,
      AppendTo[result,
        {dates[[i]], ElapsedDaysBetween[ts, dates[[i]], dates[[i + 1]]}]];
  ];

  Return[TimeLineTrim[result]];
];
```

### Total Elapsed Days

Total elapsed days over all time that a Boolean TimeLine is true. Perhaps more simply implemented with Regularize?

In[725]:=

```
TotalElapsedDays[ts_] :=
  Block[{result = 0, pairs = TimeLineUnbox[ts], i},
    For[i = 1, i <= Length[pairs] - 1, i++,
      If[pairs[[i, 2]] === True,
        result += AbsoluteTime[pairs[[i + 1, 1]]] - AbsoluteTime[pairs[[i, 1]]]
      ];
    Return[result / 86400];
];
```

## Elapsed Days Between Two Dates

Number of elapsed days between two dates during which a Boolean TimeLine is true. Perhaps more simply implemented with Regularize?

```
In[726]:= ElapsedDaysBetween[False, date1_, date2_] = 0;
ElapsedDaysBetween[ts_, date1_, date2_] :=
  Block[{pairs = Append[TimeLineUnbox[TimeSeries[ts]], {EndOfTime, False}],
    result = 0, i,
    d1 = AbsoluteTime[date1], d2 = AbsoluteTime[date2], t1, t2, left, right},

  For[i = 1, i <= Length[pairs] - 1, i++,
    t1 = AbsoluteTime[pairs[[i, 1]]];
    t2 = AbsoluteTime[pairs[[i + 1, 1]]];
    left = Max[d1, t1];
    right = Min[d2, t2];

    If[pairs[[i, 2]] === True && left < right, result += right - left]
  ];

  Return[result / 86400];
];
```

## Miscellaneous

Insert date-value pairs into a TimeLine, even if they are redundant (i.e. have the same value as the prior pair).

Regularize[T, interval]

Shift the values in a time series to the right or left:

Shift[T, n]

Maybe more general than Shift (applies fcn to all times):

TimeSeriesDatesMap[f, T]

---

## Other Rule Structures

### Tabular Rules

TODO: Use Dataset as a way to define rule logic in table form.

### Topical Crossovers

This function facilitates seamless crossovers from one topic to another. A crossover is a stopping point in the interview where the user is asked about an intermediate node in the function graph before diving

down into it. For example, if the user knows their unearned income, they can supply that information; if they need help calculating it, the interview will proceed down the tree to investigate it. The crossover rule below governs this process. If the user provides an answer, it will return that value. Otherwise, it will evaluate the logic interactively to make a determination.

In[728]:=

```
Crossover[question_, logic_] := Which[
  (*knownQ[logic], logic,*)
  (* Can answer be inferred from the current facts? *)
  !uncertainQ[question], Return[question],
  (* If not, ask the crossover question. *)
  True, Return[logic] (* As a last resort,
  dive down into the crossover topic. *)
];
SetAttributes[Crossover, HoldAll];
```

## Counterfactual Rules

Sometimes in legal materials one must consider what conclusion might have been drawn had the facts been slightly different. Modeling this sort of counterfactual reasoning in rules can lead to a lot of rule duplication. The following function allows you to assess a goal with hypothetical facts, all within the context of some other assessment. TODO: This needs to be thought through more carefully.

```
Counterfactual[goal_, hypotheticalFacts_] :=
  First[ApplyRules[{goal}, hypotheticalFacts]];
SetAttributes[Counterfactual, HoldAll];
```

## Node Metadata

TODO: Associate natural language expressions with each fact or function.

```
node["US`Fed`Tax`Income`StandardDeduction`standardDeduction", p_] := <|
  "In" → {"What is ", p, "'s standard (income tax) deduction?"},
  "Out" →
    {p, "'s standard deduction is ", Hold[toUSD[standardDeduction[p]]], "."},
  "Type" → "USD"
|>
```

Possible idea for non-Booleans. Can be converted to question and declarative form.

```
node["US`Fed`Tax`Income`StandardDeduction`standardDeduction", p_] := <|
  "Name" → {p, "'s standard (income tax) deduction?"},
  "Val" → Hold[toUSD[standardDeduction[p]]],
  "Type" → "USD"
|>
```

```
Dataset[{
  <|"Name" → "standardDeduction", "Text" →
    {"What is ", {1}, "'s standard (income tax) deduction?"}, "Type" → "USD"|>,
  <|"Name" → "standardDeduction2", "Text" → "x", "Type" → "USD"|>
}]
```

## Proof Trees

### Explain

#### Introduction

The `Explain[goal, facts]` function applies a rule to a fact pattern and generates a “proof tree” explaining, in a hierarchical format, the reasons for the chain of inferences. Nodes of the proof tree can be expanded and collapsed for ease of debugging. Facts and conclusions are shown in black, and their values in blue. Hovering over a `TimeLine` value will show a `TimeLinePlot` in a tooltip. Hovering over a conclusion will show the rule definition for that conclusion.

A test rule:

```
eligible[p_] :=
  age[p] > 65 &&
  Fact[state[p]] == "Utah" &&
  Fact[income[p]] < 50 000;
age[p_] := Fact[doB[p]] + 17;
```

Example usage.

```
Explain[
  eligible["Joe"], <|state["Joe"] → "Utah", income["Joe"] → 40 000, doB["Joe"] → 84|>]
```

```
▼ eligible[Joe] = True
```

```
  ▼ age[Joe] = 101
```

```
    doB[Joe] = 84
```

```
    state[Joe] = Utah
```

```
    income[Joe] = 40 000
```

With temporal inputs and inferences, `TimeLinePlots` are shown as tooltips.

```
Explain[eligible["Joe"], <|state["Joe"] → "Utah", income["Joe"] →
  TimeLine[{{DawnOfTime, 0}, {{2015, 4, 4}, 80 000}}], doB["Joe"] → 84|>]
```

```
▼ eligible[Joe] = TimeLine[{{{2000, 1, 1}, True}, {{2015, 4, 4}, False}}]
```

```
  ▼ age[Joe] = 101
```

```
    doB[Joe] = 84
```

```
    state[Joe] = Utah
```

```
    income[Joe] = TimeLine[{{{2000, 1, 1}, 0}, {{2015, 4, 4}, 80 000}}]
```



## Implementation

This function generates an explanation of a legal conclusion. Functions like `ApplyRules` and `MissingData` use replacement rules as a way of causing functions (rules) to be applied to input facts. The `Explain` function works differently: It temporarily defines facts as Wolfram Language functions (e.g. `Fact[age["Jim"]] = 24`), generates a `Trace`, cleans up and formats the results of the `Trace`, and then clears the temporary functions from memory.

```
In[730]:= Explain[goal_, facts_] := Block[{proofTree},
  AssertFacts[facts];
  proofTree = ProofTree[fullPruneTrace[goal]];
  Clear[Fact];
  Return[proofTree];
];
SetAttributes[Explain, HoldAll];
```

Assert functions into memory. These are cleared from memory immediately after the proof tree is generated.

```
In[732]:= AssertFacts[facts_Association] := KeyValueMap[(Fact[#1] = #2) &, facts];
```

A proof tree is a hierarchical report of the reasons for a particular determination.

```
In[733]:= ProofTree[{expr_, steps___}] :=
  OpenerView[{formatProofTreeLine[expr], Column[Map[ProofTree, {steps}]]},
    Method → "Active"]
ProofTree[x_] := formatProofTreeLine[x];
```

Prune unnecessary elements from the `Trace`, transforming the result into something that the `ProofTree` function can accept.

```
In[735]:= fullPruneTrace[f_] :=
  DeleteDuplicates[Flatten[pruneTrace[Trace[f, TraceOff → TimeLineMap]], 1]] //.
  {HoldForm[Fact[x_]]} → HoldForm[Fact[x]];
SetAttributes[fullPruneTrace, HoldAll];

pruneTrace[tr_List] := Map[pruneTrace, Catenate[{
  {tr[[1]]},
  Select[tr, Head[#] == List &]
}]];
pruneTrace[tr_] := tr;
```

Formats the expression as a row in the proof tree. Inputs to this function are in `HoldForm`. Expressions that start with `Fact` have that wrapper removed in the display.

```

In[739]:= formatProofTreeLine[f_] := Block[{factQ, val, styledVal},
  factQ = StringTake[ToString[f], {1, 4}] == "Fact";
  val = ReleaseHold[f];
  styledVal = Style[val, Blue];

  (* fact = value *)
  Row[{
    If[factQ, StringTake[ToString[f], {6, -2}],
      Tooltip[HoldForm[f], getDefinitionFromHoldForm[f]]],
    " = ",
    If[TimeLineQ[val], Tooltip[styledVal, TimeLinePlot[val]], styledVal]]
  ];
  SetAttributes[formatProofTreeLine, HoldAll];

```

Given an input like `HoldForm[eligible["Joe"]]`, returns the definition of the `eligible` function as a string. Used to display rules as tooltips in the proof tree.

```

In[741]:= getDefinitionFromHoldForm[expr_] := Block[{str = ToString[expr]},
  ToString[Definition[
    Evaluate[StringTake[str, {1, StringPosition[str, "["][[1, 1]] - 1}]]]]
]

```

## Natural Language Explanation

TODO

## EndPackage

```

End[];
EndPackage[];

```