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). 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 four basic things:

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

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 = "";
Ask::usage = "";
AsOf::usage = "";
Date::usage = "";
DawnOfTime::usage = "";
EndOfTime::usage = "";
EverQ::usage = "";
IfThen::usage = "";
InterviewPopup::usage = "";
InterviewAPI::usage = "";
InterviewWidget::usage = "";
MissingData::usage = "";
Numeric::usage = "";
ProofTree::usage = "";
RuleStub::usage = "";
SwitchThen::usage = "";
TimeLine::usage = "";
TimeLineMap::usage = "";
TimeLinePlot::usage = "";
TrueBefore::usage = "";
TrueBetween::usage = "";
TrueOnOrAfter::usage = "";
Uncertain::usage = "";
```

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

Background: Modeling Legal Rules

Facts

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

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

Rules

Rules derive new facts from exising ones, using **Ask** to mark inputs. Base-level facts are undefined symbols. The following functions are used to demonstrate the syntax and for testing purposes.

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

```
(* age[p_]:= QuantityMagnitude[DateDifference[Ask[doB[p]],Now,"Year"]]; *)
```

```
qualifyingRelationship[a_, b_] := Ask[age[a]] > 18 && Ask[gender[b]] == "Female";
```

Information-Seeking

Identifying Needed Inputs

Ask

The **Ask** function doesn't take any action on its own; it's just a wrapper used to mark inputs, for example `Ask[doB[p]]`. It may later be expanded to include question definitions: `Ask[fact_, q_QuestionObject]`.

ApplyRules

This function resolves goals by applying the rules that we defined to a set of facts:

```
ApplyRules[goals_List, facts_Association] :=
  Quiet[goals //. Normal[KeyMap[Ask[#] &, facts]]];
SetAttributes[ApplyRules, HoldAll];
```

Unit tests:

```
ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <|
  age["Lucy"] → 21, gender["Sam"] → "Female"|>]
```

```
{True}
```

```
{TestID → "dc311d48-8306-4cb3-8c44-a696d7030ed9"}
```

Success ✓ Details »

AddMessages | AddOptions | Rerun

```
ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <|
  age["Lucy"] → 6, gender["Sam"] → "Female"|>]
```

```
{False}
```

```
{TestID → "b210b799-ce54-40b5-ab5c-25c01a727123"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <|age["Lucy"] → 6|>]
```

```
{False}
```

```
{TestID → "52bbb27a-b6d2-4e7f-967b-cf00edd4dcda"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

If there are missing facts, **`ApplyRules`** returns the expression that remains to be evaluated:

```
ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <||>]
```

```
{Ask[age["Lucy"]] > 18 && Ask[gender["Sam"]] == "Female"}
```

```
{TestID → "c5a739c5-dce7-4399-9be7-47887675f1b6"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <|age["Lucy"] → 22|>]
```

```
{Ask[gender["Sam"]] == "Female"}
```

```
{TestID → "77ac1e95-c2a0-4e35-b6d8-14c1da285704"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

MissingData

The **MissingData** function harvests the **Ask** expressions from the expression that results from the application of rules to facts. TODO: Overload the definition of this function to give it a signature like ApplyRules?

```
MissingData[rs_] := Quiet[DeleteDuplicates[Cases[rs, Ask[_], All]] /. Ask[f_] → f];
SetAttributes[MissingData, HoldAll];
```

Unit tests:

```
MissingData[ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <||>]]
```

```
{age["Lucy"], gender["Sam"]}
```

```
{TestID → "6e477025-d158-468e-9b24-561fe2a72b08"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
MissingData[
 ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <|age["Lucy"] → 22|>]]
```

```
{gender["Sam"]}
```

```
{TestID → "be1e44aa-8847-4884-a00e-b27d07bbfeb7"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
MissingData[ApplyRules[{qualifyingRelationship["Lucy", "Sam"]}, <|
   age["Lucy"] → 6, gender["Sam"] → "Female"|>]]
```

```
{}
```

```
{TestID → "05f59c06-5216-4541-88b3-725a38f844b9"}
```

Success Details »

AddMessages AddOptions Rerun

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 recusively driven by the list of missing facts.

```
InterviewPopup[goals_List, facts_Association] :=
  Quiet[Module[{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"]}, <||>]
```

```
{False}
```

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.

Handling Uncertainty

Introduction

States of uncertainty allow the system to keep track of facts the user doesn't know and legal rules that have not been modeled. They are used to create a system of multivalued logic with the following additional states:

RuleStub - Indicates where the rule logic is incomplete (e.g. **RuleStub["42 U.S.C. 1983"]**)
Uncertain - Indicates what is unknown by the user (e.g. a fact she doesn't know the value of, such as **Uncertain[doB[Mary]]**)

The following are used to simplify outputs:

```
RuleStub[{}] = RuleStub[];
Uncertain[{}] = Uncertain[];
```

Tests for States of Uncertainty

The **TimeLine** object is described below.

```
RuleStubQ[x_] := Head[x] === RuleStub;
UncertainQ[x_] := Head[x] === Uncertain;
DSLTypeQ[x_] := TimeLineQ[x] || UncertainQ[x] || RuleStubQ[x];
```

Merging Uncertain States Together

If there are multiple states of the same kind of uncertainty, they need to be merged together.

```
mergeStubs[list_List] :=
 RuleStub[DeleteDuplicates[Flatten[list /. RuleStub[x___] -> x]]];

mergeUncertains[list_List] :=
 Uncertain[DeleteDuplicates[Flatten[list /. Uncertain[x___] -> x]]];
```

Unit tests:

```
mergeStubs[{RuleStub["x"], RuleStub["y"]}]
```

```
RuleStub[{"x", "y"}]
```

```
{TestID → "e5f0f6a6-c14c-443f-b2d2-055bfa6be035"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
mergeStubs[{RuleStub[], RuleStub["y"]}]
```

```
RuleStub[{"y"}]
```

```
{TestID → "1b0b033f-12a5-468d-9aac-0015befdd2cc"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
mergeStubs[{RuleStub[], RuleStub[]}]
```

```
RuleStub[]
```

```
{TestID → "cd8114d9-54ad-47b8-9f42-f61e0b8a9b6a"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
mergeUncertains[{Uncertain["x"], Uncertain[f[x]]}]
```

```
Uncertain[{"x", f[x]}]
```

```
{TestID → "8fc1fb6d-c04c-47e8-8951-1d38e2061153"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Short-Circuit Operations

Some operations - And, Or, and Times - short-circuit in their evaluation. For example, False and True short-circuits to False; x * 0 short-circuits to 0. The following functions prevent uncertainty from propagating up the function tree when a short-circuit evaluation is possible. The prefix “static” is intended to distinguish these functions from their time series counterparts.

```
staticAnd[False, q_] = False;
staticAnd[p_, False] = False;
staticAnd[True, True] = True;
staticAnd[p_, q_] := Which[
    MemberQ[{p, q}, _Uncertain], mergeUncertains[Select[{p, q}, UncertainQ]],
    MemberQ[{p, q}, _RuleStub], mergeStubs[Select[{p, q}, RuleStubQ]],
    True, p && q];

staticOr[True, q_] = True;
staticOr[p_, True] = True;
staticOr[False, False] = False;
staticOr[p_, q_] := Which[
    MemberQ[{p, q}, _Uncertain], mergeUncertains[Select[{p, q}, UncertainQ]],
    MemberQ[{p, q}, _RuleStub], mergeStubs[Select[{p, q}, RuleStubQ]],
    True, p || q];

staticTimes[0, q_] = 0;
staticTimes[p_, 0] = 0;
staticTimes[p_, q_] := handleUncertainty[p * q, {p, q}];
```

Some unit tests for the above functions:

```
staticAnd[True, False]
```

```
False
```

```
{TestID → "d9bc19b2-fc75-4873-b194-e4f369ab0dd5"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
staticAnd[True, RuleStub[]]
```

```
RuleStub[]
```

```
{TestID → "611b2409-b09b-4a77-9fd3-c046cca36f7b"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
staticOr[True, RuleStub[]]
```

```
True
```

```
{TestID → "5662072e-bacd-4752-8926-b38c10858700"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
staticOr[Uncertain[], RuleStub[]]
```

```
Uncertain[]
```

```
{TestID → "348dfab3-64bb-47c6-8280-fd0210bb7905"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
staticTimes[0, RuleStub[]]
```

```
0
```

```
{TestID → "f9f084c2-ec40-4f54-a33e-5fd2443640fe"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
staticTimes[Uncertain[], RuleStub[]]
```

```
RuleStub[]
```

```
{TestID → "052cb6d1-ceb6-47e6-b108-91409a20103a"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

HandleUncertainty

These are wrapper rules that return the highest-precedence type of uncertainty or, if all items in the input list are certain, the specified expression.

```
handleUncertainty[expr_, And, {a_, b_}] := staticAnd[a, b];
handleUncertainty[expr_, Or, {a_, b_}] := staticOr[a, b];
handleUncertainty[expr_, Times, {a_, b_}] := staticTimes[a, b];
handleUncertainty[expr_, f_, {a_, b_}] := handleUncertainty[expr, {a, b}];
handleUncertainty[expr_ : Except[_Ask], list_ ?listMayPassQ] := Which[
    MemberQ[list, _RuleStub], mergeStubs[Select[list, RuleStubQ]],
    MemberQ[list, _Uncertain], mergeUncertains[Select[list, UncertainQ]],
    True, expr]
SetAttributes[handleUncertainty, HoldFirst];
```

Helper function: A list should be trapped (not evaluated further) if it contains **Ask** and not (**RuleStub** or **Uncertain**). In other words, if it has **RuleStub** or **Uncertain**, it can be evaluated; if it has **Ask** it cannot; otherwise it can.

```
listMayPassQ[list_List] := Which[
    MemberQ[list, _RuleStub] || MemberQ[list, _Uncertain], True,
    MemberQ[list, _Ask], False,
    True, True];
```

Unit tests:

```
handleUncertainty[False && True, And, {False, True}]
```

```
False
```

```
{TestID → "afe1647e-c163-4633-b33f-6078bb357395"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
handleUncertainty[RuleStub[] && True, And, {RuleStub[], True}]
```

```
RuleStub[]
```

```
{TestID → "fde32e7f-5907-4b4b-ba05-cbe6ea45a26e"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

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.

```
DawnOfTime = {1, 1, 1};
EndOfTime = {9999, 12, 31};
```

TimeLine

TimeLine

Here’s an example of a Boolean **TimeLine**:

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

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

```
TimeLineUnbox[t_TimeLine] := t /. TimeLine[x_] -> x;
```

Convert a scalar to a **TimeLine** (internal):

```
ToTimeLine[v_TimeLine] := v;
ToTimeLine[v_] := TimeLine[{{DawnOfTime, v}}];
```

Extract the dates from one or more time series:

```
TimeLineDates[ts_TimeLine] := Map[Extract[#, 1] &, TimeLineUnbox[ts]];
TimeLineDates[ts1_TimeLine, ts2_TimeLine] :=
  Union[TimeLineDates[ts1], TimeLineDates[ts2]];
```

Extract the values from a time series:

```
TimeLineValues[ts_TimeLine] := Map[Extract[#, 2] &, TimeLineUnbox[ts]];
```

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 timeSeries. (Avg. execution time = 0.00023s.)

```
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**:

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

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

```
TimeLineFirstValue[ts_] := handleUncertainty[TimeLineUnbox[ts][[1]][[2]], {ts}];
```

Some tests of the above:

```
TimeLineValues[TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}]]
```

```
{True, False}
```

```
{TestID → "28d8a0bd-4c01-44cb-ae2c-06995b1b6924"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

AsOf

Get the value of a `TimeLine` at a particular point in time:

```
AsOf[date_, ts_TimeLine] := handleUncertainty[QuickAsOf[date, ts], {date, ts}];
AsOf[date_, ts_stub] := handleUncertainty[ts, {date, ts}];
AsOf[date_, scalar_] := scalar;
```

An internal method for doing the above operation quickly:

```
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 < AbsoluteTime[t[[i, 1]]], Return[t[[i - 1]][[2]]] ]];
      (* Else, return the value of the last interval *)
     Return[t[[len]][[2]]];
 ];
QuickAsOf[date_, scalar_] := scalar;
```

Unit tests:

```
AsOf[{2005, 3, 3}, TimeLine[{{{2000, 1, 1}, 483}, {{2010, 1, 1}, 222}}]]
```

```
483
```

```
{TestID → "3b32030d-df76-4980-8139-44e4bbde0453"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AsOf[{2005, 3, 3}, TimeLine[{{{2000, 1, 1}, 483}, {{2010, 1, 1}, RuleStub[]}}]]
```

```
483
```

```
{TestID → "c758c171-2742-4aa3-bde6-39efebb187e8"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AsOf[{2005, 3, 3},
 TimeLine[{{{2000, 1, 1}, Uncertain[]}, {{2010, 1, 1}, RuleStub[]}}]]
```

```
Uncertain[]
```

```
{TestID → "ca007a51-1243-4895-a78f-61b4b4ff0b36"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AsOf[RuleStub[], TimeLine[{{{2000, 1, 1}, 483}, {{2010, 1, 1}, 222}}]]
```

```
RuleStub[]
```

```
{TestID → "28a190b5-ba35-4551-9346-6b16e8db25e3"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AsOf[Uncertain[], TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2010, 1, 1}, 222}}]]
```

```
Uncertain[]
```

```
{TestID → "7d95e0c5-7859-40a8-bfa8-a8bd41c632f9"}
```

Success ✓ Details »

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.

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

TimeLineMap[f_, scalar_ : Except[_Ask]] := handleUncertainty[f[scalar], {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[handleUncertainty[f[#1, #2], f, {#1, #2}] &, ts1, ts2];

(* If the time series doesn't have any stub intervals,
return the scalar stub. *)
TimeLineMap[f_, ts_TimeLine, s_stub] :=
    If[TimeLineContainsStubInterval[ts],
        TimeLineTrim[
         Map[{#[[1]], handleUncertainty[f[#[[2]], s], f, {#[[2]], s}]} &,
     TimeLineUnbox[ts]]
         ],
     s];
TimeLineMap[f_, s_stub, ts_TimeLine] :=
    If[TimeLineContainsStubInterval[ts],
        TimeLineTrim[
         Map[{#[[1]], handleUncertainty[f[s, #[[2]]], f, {s, #[[2]]}]} &,
     TimeLineUnbox[ts]]
         ],
        s];

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

(* TODO: Handle stub arguments *)
TimeLineMap[f_, scalar1_ : Except[_Ask], scalar2_ : Except[_ask]] :=
  handleUncertainty[f[scalar1, scalar2], f, {scalar1, scalar2}];

(* Indicates whether any interval in the time series is a stub *)
TimeLineContainsStubInterval[ts_TimeLine] :=
 MemberQ[TimeLineUnbox[ts], {_, _stub}]
```

Utility function

```
toList[val_List] := val;
toList[val_] := {val};
```

TimeLineMapZip

Combines two time series like a zipper, which is more efficient that using **Union[TimeLineDates[]]** and then applying **f** to each interval. This function is the same as the core **TimeLineMap**, but doesn't handle uncertainty when applying the function **f** at each interval. This is needed for when the default value in an **IfThen** statement is **Uncertain**.

```
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[{DateObject[t1[[1, 1]]], DateObject[t2[[1, 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 &&
            AbsoluteTime[nextDate2] > AbsoluteTime[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 `if` function doesn't return a time series that starts at the `DawnOfTime`:

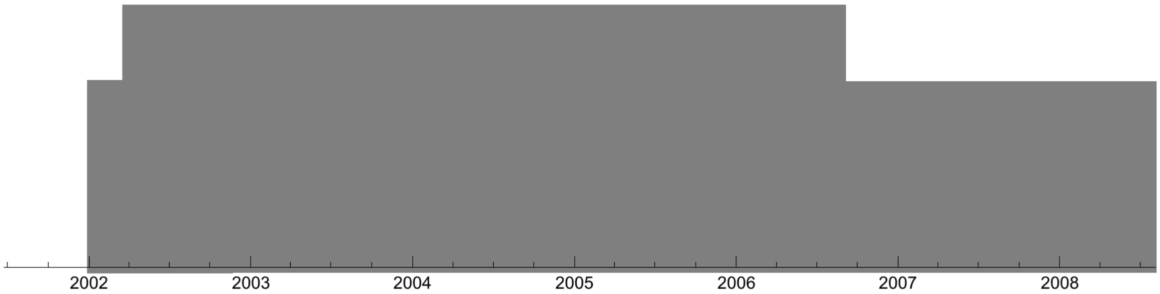
```
TimeLineMapZip[f_, ts1_TimeLine, scalar_] :=
 TimeLineMapZip[f, ts1, TimeLine[{{TimeLineUnbox[ts1][[1, 1]], scalar}}]]
```

Visualizing TimeLines

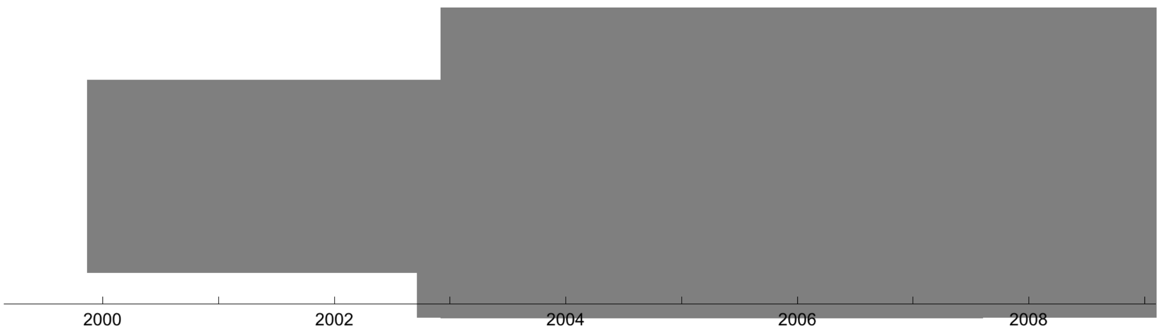
The following functions visually display TimeLines. Note that `TimeLinePlot`, defined here, is different from `TimelinePlot`, defined in the core Wolfram Language.

```
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, {3, "Year"}], 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_Uncertain] :=
  Style[Labeled[Interval[{DateObject[start], DateObject[end]}], val], Gray];
intervalStyling[start_, end_, val_RuleStub] :=
  Style[Labeled[Interval[{DateObject[start], DateObject[end]}], val], Purple];
intervalStyling[start_, end_, val_] :=
  Labeled[Interval[{DateObject[start], DateObject[end]}], val];
```

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




```
TimeLinePlot[
 {TimeLine[{{{2002, 3, 3}, True}, {{2002, 5, 5}, False}, {{2005, 3, 1}, Uncertain[]}}],
  TimeLine[{{{2000, 3, 3}, 22 700},
    {{2003, 5, 5}, 34 000}, {{2005, 7, 1}, Uncertain[]}}]}]
```



Misc

```
(* Creates a time series starting at the beginning of a calendar year,
with values that change annually *)
AnnualTimeLine[startYear_, list_List] := TimeLine[
 Insert[Transpose[{Map[{#, 1, 1} &,
   Range[startYear, startYear + Length[list] - 1]], list}],
    {DawnOfTime, RuleStub["Law"]}, {1}]
  ]

(* Determines whether the first interval in a time series represents "dawn." *)
(* DELETE? *)
timeLineStartsAtDawn[ts_] := handleUncertainty[
  DateList[TimeLineUnbox[ts][[1]][[1]]][[1 ;; 3]] ==
   DateList[DawnOfTime][[1 ;; 3]], {ts}];
```

Boolean TimeLines

Boolean Operators

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

```
and[p_, q_] := TimeLineMap[And, p, q];
Unprotect[And];
And[False, a_] = False;
And[a_, False] = False;
And[a_, b_TimeLine] := and[a, b];
And[a_, b_RuleStub] := and[a, b];
And[a_, b_Uncertain] := 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];
Or[a_, b_RuleStub] := or[a, b];
Or[a_, b_Uncertain] := or[a, b];
Protect[Or];

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

Unit tests:

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}] &&
 TimeLine[{{{2000, 1, 1}, False}, {{2005, 1, 1}, True}}]
```

```
TimeLine[{{{1, 1, 1}, False}, {{2005, 1, 1}, True}, {{2010, 1, 1}, False}}]
```

```
TimeLine[{{{2000, 1, 1}, False}, {{2005, 1, 1}, True}, {{2010, 1, 1}, False}}]
```

```
{TestID → "7a3f786d-2daf-46e9-993f-9d173d4a2669"}
```

Failure ✖ Details »

AddMessages AddOptions ReplaceOutput Rerun

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}] ||
 TimeLine[{{{2000, 1, 1}, False}, {{2005, 1, 1}, True}}]
```

```
True
```

```
{TestID → "b18616e8-3b6d-46c1-80ce-70ed71f138b9"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}] && True
```

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

```
{TestID → "87280230-2d15-41df-a910-3ebfbf90d99f"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}] && False
```

```
False
```

```
{TestID → "2e68921e-0ba6-44de-b623-a87e3a36860a"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}] && RuleStub[]
```

```
TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2010, 1, 1}, False}}]
```

```
{TestID → "bdec8849-286e-4249-a984-bfabc9585366"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
RuleStub[] && TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}]
```

```
TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2010, 1, 1}, False}}]
```

```
{TestID → "efa50fbf-913e-4eb9-8977-4bef8316dada"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}] || True
```

```
True
```

```
{TestID → "fdeaa2b8-a224-4eb2-a17e-31bd053aa02a"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
! TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}]
```

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

```
{TestID → "51d5dc88-83c5-4d44-a825-cf659d889065"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}] || RuleStub[]
```

```
TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, RuleStub[]}}]
```

```
{TestID → "d27d7749-0359-4788-9986-43ef471f5e34"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
! TimeLine[{{DawnOfTime, True}}]
```

```
False
```

```
{TestID → "9537c8bd-ae5a-46f3-8a5f-074113e487d5"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
! RuleStub[]
```

```
RuleStub[]
```

```
{TestID → "1eca99b7-87bf-48ea-89aa-03c62d123f5f"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
! Uncertain[]
```

```
Uncertain[]
```

```
{TestID → "07c312a7-e08a-47c1-8917-ba3b53d34708"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

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 *)
TrueOnOrAfter[date_] := handleUncertainty[
    If[date === DawnOfTime, True,
     TimeLine[{{DawnOfTime, False}, {date, True}}]],
    {date}];

(* Creates a Boolean time series that's true before a given date,
and false thereafter *)
TrueBefore[date_] := handleUncertainty[
    If[date === DawnOfTime, False,
     TimeLine[{{DawnOfTime, True}, {date, False}}]],
    {date}];

(* Creates a Boolean time series that's true between two dates,
and otherwise false *)
TrueBetween[date1_, date2_] := and[TrueOnOrAfter[date1], TrueBefore[date2]];
```

Unit tests:

```
TrueOnOrAfter[{2017, 2, 2}]
```

```
TimeLine[{{{1, 1, 1}, False}, {{2017, 2, 2}, True}}]
```

```
{TestID → "119ab0da-6849-407f-8a9f-1faf41e8a7fd"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TrueOnOrAfter[DawnOfTime]
```

```
True
```

```
{TestID → "753bdcad-8583-4d87-b1cf-6a15540f54df"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TrueBefore[{2017, 2, 2}]
```

```
TimeLine[{{{1, 1, 1}, True}, {{2017, 2, 2}, False}}]
```

```
{TestID → "3506f3a4-0653-40ca-a40d-8b6efd81d584"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TrueBefore[DawnOfTime]
```

```
False
```

```
{TestID → "747d92f5-f251-4fef-87c1-b04381e8ac5f"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TrueBetween[{2017, 2, 2}, {2020, 7, 7}]
```

```
TimeLine[{{{1, 1, 1}, False}, {{2017, 2, 2}, True}, {{2020, 7, 7}, False}}]
```

```
{TestID → "1228a6b0-8f47-463b-9830-da58906f97d9"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TrueBetween[DawnOfTime, {2020, 7, 7}]
```

```
TimeLine[{{{1, 1, 1}, True}, {{2020, 7, 7}, False}}]
```

```
{TestID → "903a3d0d-eef5-473e-9861-2cae75c1cd7d"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Ever & Always

Does a time series ever have a particular value? Does it always? (TODO: Deal with scalars and temporal uncertainty.)

```
EverQ[ts_, val_] := MemberQ[TimeLineValues[ts], val];
EverQ[ts_] := EverQ[ts, True];

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

Unit tests:

```
EverQ[True]
```

```
True
```

```
{TestID → "643b5c13-ea3a-4506-a410-5b2506000332"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
EverQ[False]
```

```
False
```

```
{TestID → "b60b438f-bc44-4732-8167-c1fc7e21f793"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
EverQ[TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}]]
```

```
True
```

```
{TestID → "e9159525-699c-4dca-a17d-b6585bc06ebe"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
EverQ[TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}], False]
```

```
True
```

```
{TestID → "7756c348-37f6-4c27-8d08-40da59cad018"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AlwaysQ[True]
```

```
True
```

```
{TestID → "6a01a9cf-6093-45a7-bfad-b4895acb6001"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AlwaysQ[False]
```

```
False
```

```
{TestID → "bd72bfe9-f68f-403c-9ea3-d0a9bd76dac2"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AlwaysQ[TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}]]
```

```
False
```

```
{TestID → "0158097b-cf4d-40e5-8033-fd8735ac81ef"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
AlwaysQ[7, 7]
```

```
True
```

```
{TestID → "47559451-5ee6-4c28-b2ac-46fd7af56d2c"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

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.

```
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];
```

Tests of InternalIf:

```
InternalIf[TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}], 1, 0]
```

```
TimeLine[{{{2000, 1, 1}, 1}, {{2010, 1, 1}, 0}}]
```

```
{TestID → "9634baac-aece-4875-92d2-e2e30322b192"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
InternalIf[TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}],
 TimeLine[{{DawnOfTime, True}, {{2020, 1, 1}, False}}], 0]
```

```
TimeLine[{{{1, 1, 1}, True}, {{2010, 1, 1}, 0}}]
```

```
{TestID → "788e3981-08c0-4a99-a951-ceb8534c8e80"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
InternalIf[True, TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}], False]
```

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

```
{TestID → "888941dd-97ee-4470-bd56-e4bccb85c476"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
InternalIf[False, TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}], False]
```

```
False
```

```
{TestID → "8779f9e5-0f85-4920-8047-14275ae4e27a"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
InternalIf[TimeLine[{{{2000, 1, 1}, True}, {{2010, 1, 1}, False}}],
 TimeLine[{{DawnOfTime, True}, {{2020, 1, 1}, False}}],
 TimeLine[{{DawnOfTime, RuleStub[]}, {{2015, 1, 1}, Uncertain[]}}]]
```

```
TimeLine[
 {{{1, 1, 1}, True}, {{2010, 1, 1}, RuleStub[]}, {{2015, 1, 1}, Uncertain[]}}]
```

```
{TestID → "ea95eec7-4afe-419d-abcd-924cfd1311c0"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Unit tests for recursive time series functions:

```
fib[n_] := InternalIf[n ≤ 2, 1, fib[n - 1] + fib[n - 2]];
```

```
fib[TimeLine[{{{2000, 1, 1}, 8}, {{2010, 1, 1}, 9}}]]
```

```
TimeLine[{{{2000, 1, 1}, 21}, {{2010, 1, 1}, 34}}]
```

```
{TestID → "34b3b9a6-2a47-403c-9e4a-f7e65fe409b3"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
fib[TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2010, 1, 1}, 9}}]]
```

```
TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2010, 1, 1}, 34}}]
```

```
{TestID → "d88db481-c88e-4925-9e13-796e5c729c42"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
fib[Uncertain[]]
```

```
Uncertain[]
```

```
{TestID → "ab4fd64b-98c7-4126-99a5-c5f4d09b2945"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

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.

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

s

Unit tests:

```
IfThen[True, 8, 9]
```

```
8
```

```
{TestID → "e9143387-9148-4c98-8b57-6041db845bb3"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
IfThen[False, 8, 9]
```

```
9
```

```
{TestID → "639556a9-66b4-4979-bf33-a5040e687600"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
IfThen[False, 8, False, 9, 10]
```

```
10
```

```
{TestID → "d1343dd1-4d6e-4343-b79c-cc661b55a5a0"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
IfThen[TrueOnOrAfter[{1996, 4, 4}], 42, 43]
```

```
TimeLine[{{{1, 1, 1}, 43}, {{1996, 4, 4}, 42}}]
```

```
{TestID → "35cba894-0da3-4b69-9d54-ca3f4a9eba2a"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
IfThen[TrueOnOrAfter[{1996, 4, 4}], 42, TrueOnOrAfter[{1993, 4, 4}], 43, 44]
```

```
TimeLine[{{{1, 1, 1}, 44}, {{1993, 4, 4}, 43}, {{1996, 4, 4}, 42}}]
```

```
{TestID → "bcb5ed5d-09bc-4784-a1cf-471ac0f92b71"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Recursion tests:

```
fib2[n_] := IfThen[n ≤ 2, 1, fib[n - 1] + fib[n - 2]];
```

```
fib2[TimeLine[{{{2000, 1, 1}, 10}, {{2010, 1, 1}, 9}}]]
```

```
TimeLine[{{{2000, 1, 1}, 55}, {{2010, 1, 1}, 34}}]
```

```
{TestID → "890f03bc-6f28-46b3-8046-a0b3d3a57b21"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

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.

```
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[irrelevanTest_, def_] := def;
SetAttributes[SwitchThen, HoldRest];

(* Tests for the equality of two lists *)
listEqual[{a1_, a2_}, {b1_, b2_}] :=
  handleUncertainty[a1 == b1 && a2 == b2, {a1, a2, b1, b2}];
listEqual[a_List, b_List] := handleUncertainty[a == b, Join[a, b]];
```

Unit tests:

```
SwitchThen["a", "a", 1, "b", 2, "default"] == Switch["a", "a", 1, "b", 2] == 1
```

```
True
```

```
{TestID → "fafd595b-b1cd-4d93-832d-879de2cdb7d4"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
SwitchThen["b", "a", 1, "b", 2, "default"] == Switch["b", "a", 1, "b", 2] == 2
```

```
True
```

```
{TestID → "df15d38c-cf8d-43ca-be8c-63dc101f59b4"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
SwitchThen["c", "a", 1, "b", 2, "default"]
```

```
"default"
```

```
{TestID → "d556b804-af21-4e02-8b97-5f6d2a796431"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
SwitchThen[TrueOnOrAfter[{1996, 4, 4}], True, 1, False, 2, "default"]
```

```
TimeLine[{{{1, 1, 1}, 2}, {{1996, 4, 4}, 1}}]
```

```
{TestID → "1d40c9c9-83ff-4d8f-bb37-950e7c267287"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
SwitchThen[TrueOnOrAfter[{1996, 4, 4}], False, 1, True, 2, "default"]
```

```
TimeLine[{{{1, 1, 1}, 1}, {{1996, 4, 4}, 2}}]
```

```
{TestID → "dc171333-14dc-46f4-8a60-79e12679b0a1"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Here the issue is the comparison of different types. Need to overload the === operator?

```
SwitchThen[True, "a", 1, TrueOnOrAfter[{1996, 4, 4}], 2, "default"]
```

```
TimeLine[{{{1, 1, 1}, "default"}, {{1996, 4, 4}, 2}}]
```

```
If["a" == True, 1, "a" == True]
```

```
{TestID → "204aee46-c83b-4ad8-a818-3e23f66a775b"}
```

Failure ✗ Details »

AddMessages AddOptions ReplaceOutput Rerun

Tests where the switch expression is a list:

```
SwitchThen[{1, 2}, {1, 2}, "a", {2, 3}, "b", "default"]
```

```
"a"
```

```
{TestID → "b4a48c6d-3585-4aad-9aa2-d92f7c50648e"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
SwitchThen[{1, 4}, {1, 2}, "a", {2, 3}, "b", "default"]
```

```
"default"
```

```
{TestID → "0563dcdd-2fca-473c-b786-877736ef7d03"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
SwitchThen[{2, 3}, {1, 2}, "a", {2, 3}, "b", "default"]
```

```
"b"
```

```
{TestID → "13cf5ef2-82e7-4dce-96aa-4117f3df02d1"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
SwitchThen[{}, {1, 2}, "a", {2, 3}, "b", "default"]
```

```
"default"
```

```
{TestID → "f4a8ba50-5a0c-42cf-8572-2f9a41ab236f"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Arithmetic

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:

```
plus[ts1_, ts2_] := TimeLineMap[Plus, ts1, ts2]
Unprotect[Plus];
Plus[a_ ? DSLTypeQ, b_ ?DSLTypeQ ] := plus[a, b];
Plus[a_ ? DSLTypeQ, b_ ] := plus[a, b];
Plus[a_ , b_ ?DSLTypeQ ] := plus[a, b];
Protect[Plus];
```

Possibly to add 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];
```

```
TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 0}}] +
 TimeLine[{{{1, 1, 1}, 22}, {{2010, 1, 1}, 33}}]
```

```
TimeLine[{{{1, 1, 1}, 23}, {{2010, 1, 1}, 33}}]
```

```
{TestID → "006099f3-3cc9-45f6-a5f5-4573a003ed1a"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
plus[Uncertain[], TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 0}}] ]
```

```
Uncertain[]
```

```
{TestID → "0f664b1b-65e7-49c7-af2d-010e0fad3daf"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
7 + Uncertain[]
```

```
Uncertain[]
```

```
{TestID → "a8cf3de8-2bbd-4a9b-aca3-66a1094ebdec"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 0}}] + RuleStub[]
```

```
RuleStub[]
```

```
{TestID → "58d1f694-d9a9-454d-8b3e-8603e0054ca7"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
44 + TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 0}}]
```

```
TimeLine[{{{1, 1, 1}, 45}, {{2010, 1, 1}, 44}}]
```

```
{TestID → "a2b7dfbd-3bce-47e9-a7f7-f6413845796b"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
44 + TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 0}}] - 71
```

```
TimeLine[{{{1, 1, 1}, -26}, {{2010, 1, 1}, -27}}]
```

```
{TestID → "327c6d2f-77cb-4282-84ac-7d32b72c56fb"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
44 + TimeLine[{{{1, 1, 1}, RuleStub[]}, {{2010, 1, 1}, 0}}]
```

```
TimeLine[{{{1, 1, 1}, RuleStub[]}, {{2010, 1, 1}, 44}}]
```

```
{TestID → "16e47c5c-0ef2-4a38-80db-9e09f9079de8"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Multiplication

Multiplication:

```
times[ts1_, ts2_] := TimeLineMap[Times, ts1, ts2]
Unprotect[Times];
Times[a_ ? DSLTypeQ, b_ ?DSLTypeQ] := times[a, b];
Times[a_ ? DSLTypeQ, b_ ] := times[a, b];
Times[a_ , b_ ?DSLTypeQ ] := times[a, b];
Protect[Times];
```

```
TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 0}}] * RuleStub[]
```

```
TimeLine[{{{1, 1, 1}, RuleStub[]}, {{2010, 1, 1}, 0}}]
```

```
{TestID → "1ab6d469-37e0-4e7f-aee7-d945aa8e871b"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 0}}] * 0
```

```
0
```

```
{TestID → "08b07a20-77b5-45b6-bf07-94e47d1b60d2"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
0 * RuleStub[]
```

```
0
```

```
{TestID → "6096f1d9-a163-43a2-ba63-251d7257aac3"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{1, 1, 1}, 1}, {{2010, 1, 1}, 10}}] *
 TimeLine[{{{1, 1, 1}, 2}, {{2015, 1, 1}, 20}}]
```

```
TimeLine[{{{1, 1, 1}, 2}, {{2010, 1, 1}, 20}, {{2015, 1, 1}, 200}}]
```

```
{TestID → "6c02d327-f957-4e1c-bf82-49bc2d7cd568"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{1, 1, 1}, Uncertain[]}, {{2010, 1, 1}, 10}}] *
 TimeLine[{{{1, 1, 1}, RuleStub[]}, {{2015, 1, 1}, 20}}]
```

```
TimeLine[{{{1, 1, 1}, RuleStub[]}, {{2015, 1, 1}, 200}}]
```

```
{TestID → "dc264673-0777-4ae9-a1ef-f27892a5470d"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
-7 * TimeLine[{{{1, 1, 1}, 14}, {{2015, 1, 1}, 20}}]
```

```
TimeLine[{{{1, 1, 1}, -98}, {{2015, 1, 1}, -140}}]
```

```
{TestID → "eec2b17b-3283-4d16-b452-be19fb31279b"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Division

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

```
Unprotect[Power];
Power[t_ ? DSLTypeQ, -1] := TimeLineMap[1 / # &, t];
Protect[Power];
```

```
TimeLine[{{{1, 1, 1}, 14}, {{2015, 1, 1}, 20}}] / 2
```

```
TimeLine[{{{1, 1, 1}, 7}, {{2015, 1, 1}, 10}}]
```

```
{TestID → "07ed8e09-52b1-43a1-8ba0-991f0703b29b"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
40 / TimeLine[{{{1, 1, 1}, 10}, {{2015, 1, 1}, 20}}]
```

```
TimeLine[{{{1, 1, 1}, 4}, {{2015, 1, 1}, 2}}]
```

```
{TestID → "86cc3530-a545-4b8c-a770-d0c4f4c56793"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{1, 1, 1}, 14}, {{2015, 1, 1}, 20}}] / Uncertain[]
```

```
Uncertain[]
```

```
{TestID → "60915a1d-0076-40ed-93e2-d8d33e3e3d8d"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{1, 1, 1}, Uncertain[]}, {{2015, 1, 1}, 20}}] / 2
```

```
TimeLine[{{{1, 1, 1}, Uncertain[]}, {{2015, 1, 1}, 10}}]
```

```
{TestID → "a1d0f856-f034-4073-bfd5-b36a7955faf4"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
40 / TimeLine[{{{1, 1, 1}, 10}, {{2015, 1, 1}, RuleStub[]}}]
```

```
TimeLine[{{{1, 1, 1}, 4}, {{2015, 1, 1}, RuleStub[]}}]
```

```
{TestID → "a1d0f856-f334-4073-bfd5-b36a7955faf4"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Numeric

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

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

```
Numeric[TimeLine[{{{1, 1, 1}, 3 / 2}, {{2015, 1, 1}, 1 / 3}}]]
```

```
TimeLine[{{{1, 1, 1}, 1.5`}, {{2015, 1, 1}, 0.3333333333333333`}}]
```

```
{TestID → "857313d4-2831-4f07-83fc-64aa72fc6ed3"}
```

Success Details »

AddMessages AddOptions Rerun

```
Numeric[TimeLine[{{{1, 1, 1}, 3 / 2}, {{2015, 1, 1}, 1 / 3}}]]
```

```
TimeLine[{{{1, 1, 1}, 1.5`}, {{2015, 1, 1}, 0.3333333333333333`}}]
```

```
{TestID → "f590b4c2-f756-4907-8bfa-f37293169542"}
```

Success Details »

AddMessages AddOptions Rerun

```
Numeric[7]
```

```
7.
```

```
{TestID → "bdc08b4e-1d92-4062-bd0a-36f5814d86d6"}
```

Success Details »

AddMessages AddOptions Rerun

Comparison

Comparison

Comparison operators:

```
greater[ts1_, ts2_] := TimeLineMap[Greater, ts1, ts2]
Unprotect[Greater];
Greater[a_ ? DSLTypeQ, b_ ?DSLTypeQ] := greater[a, b];
Greater[a_ ? DSLTypeQ, b_ ] := greater[a, b];
Greater[a_ , b_ ?DSLTypeQ ] := greater[a, b];
Protect[Greater];

greaterEqual[ts1_, ts2_] := TimeLineMap[GreaterEqual, ts1, ts2]
Unprotect[GreaterEqual];
GreaterEqual[a_ ? DSLTypeQ, b_ ?DSLTypeQ] := greaterEqual[a, b];
GreaterEqual[a_ ? DSLTypeQ, b_ ] := greaterEqual[a, b];
GreaterEqual[a_ , b_ ?DSLTypeQ ] := greaterEqual[a, b];
Protect[GreaterEqual];

less[ts1_, ts2_] := TimeLineMap[Less, ts1, ts2]
Unprotect[Less];
Less[a_ ? DSLTypeQ, b_ ?DSLTypeQ] := less[a, b];
Less[a_ ? DSLTypeQ, b_ ] := less[a, b];
Less[a_ , b_ ?DSLTypeQ ] := less[a, b];
Protect[Less];

lessEqual[ts1_, ts2_] := TimeLineMap[LessEqual, ts1, ts2]
Unprotect[LessEqual];
LessEqual[a_ ? DSLTypeQ, b_ ?DSLTypeQ] := lessEqual[a, b];
LessEqual[a_ ? DSLTypeQ, b_ ] := lessEqual[a, b];
LessEqual[a_ , b_ ?DSLTypeQ ] := lessEqual[a, b];
Protect[LessEqual];

equal[ts1_, ts2_] := TimeLineMap[Equal, ts1, ts2]
Unprotect[Equal];
Equal[a_ ? DSLTypeQ, b_ ?DSLTypeQ] := equal[a, b];
Equal[a_ ? DSLTypeQ, b_ ] := equal[a, b];
Equal[a_ , b_ ?DSLTypeQ ] := equal[a, b];
Protect[Equal];

unequal[ts1_, ts2_] := TimeLineMap[Unequal, ts1, ts2]
Unprotect[Unequal];
Unequal[a_ ? DSLTypeQ, b_ ?DSLTypeQ] := unequal[a, b];
Unequal[a_ ? DSLTypeQ, b_ ] := unequal[a, b];
Unequal[a_ , b_ ?DSLTypeQ ] := unequal[a, b];
Protect[Unequal];
```

Unit tests:

```
TimeLine[{{{2000, 1, 1}, 4}, {{2010, 1, 1}, 3}}] >= RuleStub[]
```

```
RuleStub[]
```

```
{TestID → "650225d6-0699-459a-a3e2-37b73b31ddd2"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, 4}, {{2010, 1, 1}, 3}}] >=
 TimeLine[{{{2000, 1, 1}, 6}, {{2020, 1, 1}, 2}}]
```

```
TimeLine[{{{2000, 1, 1}, False}, {{2020, 1, 1}, True}}]
```

```
{TestID → "22dc15c4-0de5-457e-bfee-c54e01815789"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
8 == TimeLine[{{{2000, 1, 1}, 4}, {{2010, 1, 1}, 3}}]
```

```
False
```

```
{TestID → "0746d01f-9618-47ee-a998-75ce6e3493fd"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
8 == TimeLine[{{{2000, 1, 1}, 4}, {{2010, 1, 1}, 8}}]
```

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

```
{TestID → "4713f2ef-3c57-4256-9454-1666c3ec6d0b"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, 4}, {{2010, 1, 1}, 3}}] <
 TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2020, 1, 1}, 2}}]
```

```
TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2020, 1, 1}, False}}]
```

```
{TestID → "0e6b98fd-d7ed-41f0-912e-bad825f2d31c"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
Uncertain[] != TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2020, 1, 1}, 2}}]
```

```
TimeLine[{{{2000, 1, 1}, RuleStub[]}, {{2020, 1, 1}, Uncertain[]}}]
```

```
{TestID → "a84e70ed-86a2-477b-8954-a45a97129979"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{2000, 1, 1}, 4}, {{2010, 1, 1}, 3}}] <
 TimeLine[{{{2003, 1, 1}, 5}, {{2020, 1, 1}, 2}}]
```

```
TimeLine[{{{2000, 1, 1}, True}, {{2020, 1, 1}, False}}]
```

```
{TestID → "f7508d5a-1786-419b-9709-872366ea35d3"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

```
TimeLine[{{{1, 1, 1}, 4}}] == 9
```

```
False
```

```
{TestID → "6a0eeef5-919e-4288-b606-6c41e80a153b"}
```

Success ✓ Details »

AddMessages AddOptions Rerun

Date Comparisons

```
{2000, 2, 2} > {2000, 2, 1}
```

```
{2000, 2, 2} > {2000, 2, 1}
```

```
Date
```

Date Operations

Date Composition and Decomposition

```
Date[y, m, d]
```

DatePlus, DateDifference

```
DatePlus[date, n]
```

```
DatePlus[date, {n, unit}]
```

List Operations

List

```
Min
Max
Length
Map
MemberQ
Total
ContainsAny
ContainsAll
```

IntervalsSince

Absolute Time

Per Interval

Elapsed Time

Miscellaneous

Proof Trees

Proof Trees

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

```
ProofTree[{expr_, steps___}] :=
 OpenerView[{Tooltip[expr, ReleaseHold[expr]], Column[ProofTree /@ {steps}]}]
ProofTree[x_] := x
```

```
Trace[standardDeduction["Pat"], _["Pat"]] // ProofTree
```

```
▼ standardDeduction[Pat]
    ▼ ineligibleForStandardDeductionQ[Pat]
        ▶ {mfsQ[Pat], {marriedQ[Pat], {maritalStatus[Pat]}}}
        ▼ shortTaxYearQ[Pat]
            ▶ taxYearType[Pat]
        ▶ nonresidentAlienQ[Pat]
        ▶ dualStatusAlienQ[Pat]
    ▶ table20Dash3Applies[Pat]
    ▶ table20Dash2Applies[Pat]
    ▶ standardDeductionMostPeople[Pat]
```

EndPackage

```
End[]
```

```
EndPackage[];
```