Chronology Template B/chrt

Purpose

To record information now which will be needed later, when a condition phrased in the perfect tense is tested.

§**1. Scheme I.** If source text contains a condition like "if the well has been dry, ...", then we need to keep a chronological record by testing at every turn whether or not the well is dry: we log whether this is true now, whether it has ever been true, for how many consecutive turns it has been true (to a maximum of 127), and how many times it has become true having just been false (again, to a maximum of 127 times). All this information is packed into a single word, arranged as a 15-bit bitmap: the least significant bit is the state of the condition now (true or false), the next 7 bits are the number of false-to-true "trips" observed over time, and the top 7 bits are the number of consecutive turns on which the condition has been true ("consecutives"). The 16th and most significant bit is unused, so that the state is always positive even in a 16-bit virtual machine – a convenience since we then don't need to worry about the effect of signed division and remainder on the bitmap.

There is no need to store a flag for "has this condition ever been true", because this is equivalent the number of trips being greater than zero. This might look wrong for a condition which is true at start of play – say, if the well was always dry – because then its state has never changed from false to true. But in fact when the VM starts up the state word is initially always 0: it's only when the startup rulebook fires the update chronological records rule (see below) that we first test whether the well is dry, and that forces a trip from false to true if the well is dry at start of play. Therefore, if T is the number of trips for the condition then $T = 0$ if and only if the condition has been false from the very start of play. If the condition is true now, then $T = 1$ if and only if it has always been so.

§**2. Present and Past.** Each individual condition has its own unique "PT number", counting upwards from 0 in order of compilation by NI, and a "chronological record" is a word array with a state word as described above for each PT number.

However, we keep not one but two chronological records: one for the situation now, called the "present chronological record", and one for the situation as it was just before interesting things most recently happened, called the "past chronological record". If one of those interesting things was that the well ran dry, then in our example the state word for the condition "if the well has been dry" would be different in the two chronological records. We keep two records in order to be able to detect conditions in four different tenses:

(1) Present tense ("if the well is dry"): none of this machinery is used, because we can just test directly instead, so a present tense condition has no PT-number.
(2) Past tense ("if the well was dry"): we look at the flag bit in the past chronological record.
(3) Perfect tense ("if the well has been dry"): we look at whether or not $T > 0$ in the present chronological record.
(4) Past perfect tense ("if the well had been dry"): we look at whether or not $T > 0$ in the past chronological record.

It's a somewhat ambiguous matter of context in English when the reference time is for past tenses. If somebody comes out into the sunshine and says, "But it was raining," when does he mean? We would reasonably guess earlier that day, and probably only an hour or two ago, but that's a contextual judgement made on the basis of our own experience of how rapidly weather changes. The context for Inform source text is always that of actions, so our convention is that the reference time for past tenses is the point just before the current action began. Such key moments – when things are just about to happen – are called "chronology points". There is a CP just before each action begins; there is a CP in the startup process; and there is a

CP at the end of each turn, for good measure. CPs happen when somebody calls `ChronologyPoint()`. The action machinery does this directly, while the startup CP and the CP at end of turn happen in the course of the update chronological records rule (below).

§3. Chronology Point.

This is when the time of reference for past tenses is now: so it is where the past becomes the present. Soon, exciting things will happen, and the present will go on developing, while the past will remain as it was; until things calm down again and we come to another chronology point.

```
[ ChronologyPoint pt;
    for (pt=0:pt<NO_PAST_TENSE_CONDS:pt++)
        past_chronological_record-->pt = present_chronological_record-->pt;
];
```

§4. Update Chronological Records Rule.

It might seem odd that a routine to, supposedly, update something would only call another routine called `TestSinglePastState`: but in this setting, any test updates the state, because it changes the number of times something has been found true, and so on.

```
[ UPDATE_CHRONOLOGICAL_RECORDS_R pt;
    for (pt=0: pt<NO_PAST_TENSE_CONDS: pt++) TestSinglePastState(false, pt, true, -1);
    ChronologyPoint();
    rfalse;
];
```

§5. Test Single Past State.

`TestSinglePastState` is called with four arguments:

(a) `past_flag` is true if we want to test a condition like “if the well was dry” or “if the well had been dry”, which concern only the state as it was at the last chronology point – in other words, the `past_chronological_record`, which we must not change; whereas `past_flag` is false if we want to test a present tense like “if the well is dry” or “if the well has been dry”, because then we deal with the `present_chronological_record` which must be kept continuously updated.

(b) `pt` is the PT number for the condition.

(c) `turn_end` is true if we are making the test at the end of a turn, as part of the update chronological records rule, and false otherwise. (For these purposes the start of play is a turn end since the UCRR runs then, too.)

(d) `wanted` describes what information the function should return, as detailed in its code below.

```
{-call:Code::Chronology::past_actions_i6_routines}
{-call:Code::Chronology::chronology_extents_i6_escape}
[ TestSinglePastState past_flag pt turn_end wanted
    old new trips consecutives ct_0 ct_1 I7BASPL;
    if (past_flag) {
        new = (past_chronological_record-->pt) & 1;
        trips = ((past_chronological_record-->pt) & $$11111110)/2;
        consecutives = ((past_chronological_record-->pt) & $$111111100000000)/256;
    } else {
        old = (present_chronological_record-->pt) & 1;
        trips = ((present_chronological_record-->pt) & $$11111110)/2;
        consecutives = ((present_chronological_record-->pt) & $$111111100000000)/256;
! Test cases for conditions by PT number: each sets "new" to whether it is true or false now
{-call:Code::Chronology::past_tenses_i6_escape}
        if (new == false) {
            consecutives = 0;
        } else {
            if (old == false) { trips++; if (trips > 127) trips = 127; }
            if (turn_end) { consecutives++; if (consecutives > 127) consecutives = 127; }
        }
        present_chronological_record-->pt = new + 2*trips + 256*consecutives;
    }
    ! print pt, ": old=", old, " new=", new, " trips=", trips, " consec=", consecutives,
    ! " wanted=", wanted, "^";
    switch(wanted) {
        0: if (new) return new;
        1: if (new) return trips;
        2: if (new) return consecutives+1; ! Plus one because we count the current turn
        4: return new;
        5: return trips;
        6: return consecutives;
    }
    return 0;
];
```

§**6. Scheme II.** Actions discussed in the past tense – "if we have taken the ball" or, more subtly, "Instead of waiting for the third turn" (which also refers to the past) – are handled in a related but simpler way. NI counts such references, just as with other past tense conditions above, but this time stores a state in two simple word arrays: `TimesActionHasHappened` and `TurnsActionHasBeenHappening`.

It might reasonably be asked why we don't simply use all of the clever machinery above: why have an entirely different system here? One reason is that actions are events and not continuous states of being. "The well is dry" could be true for any extent of time, but "taking the ball" either happens at a given moment or doesn't: it is not continuously true. We are therefore in the business of counting events, not measuring durations. Another reason is that the point of reference for past tenses is different. It makes no sense to say "if we were looking" because that would mean at a time just before the current action, when actions were probably not happening at all; while "if we had looked" and "if we have looked" would almost always be identical in meaning for the same reason. So we need a much simpler system with just one possible past tense; we don't need to keep two different states; and we use the extra storage to enable us to count the number of times, and the number of turns, to at least 31767 instead of stopping at 127. (We also generate more legible code to test past tense actions.)

§**7. Past Action Routines.** Actions can be quite complicated to test even in the present, and this is much more conveniently done in a routine with its own NI-generated stack frame; so each past tense action has its own testing routine, with a name like `PAPR_41`, which returns true or false depending on whether the action is going on now. The word array `PastActionsI6Routines` then contains a null-terminated list of these routines.

§**8. Track Actions.** The routine `TrackActions` then updates the two arrays, and is the equivalent for past tense actions of `ChronologyPoint`. It's called twice for each in-world action: `TrackActions(false)` just as a new action is about to begin, and then `TrackActions(true)` just after it has finished – note in particular that if action B happens in the middle of action A, for instance because of a try phrase, then the `TrackActions(true)` call for B happens when A is back as the current action. In fact, that's the point: because it tells us that B was not the main action for the turn, but only a phase which has now passed again.

For each of the action patterns we track, we need to count the number of times such an action has been tried (*not* the number of times it succeeded), and also the number of consecutive turns on which it has been "the" action. The count of the number of tries is unambiguous and easy to keep: it is incremented at the start-of-action call only, and only if the action matches. But the consecutive turn count is more problematic. The user wants to think of actions and turns as synonymous, and to write conditions like "Before going for the third turn", and we want to play along because that's a natural phrasing: but actions and turns are not synonymous at all. The rules are therefore:

(1) At the start-of-action call, provided the action is not a silent one, the consecutive turns count is either incremented or zeroed, depending on whether the action matches. Silent actions are ignored because they are almost always knock-on actions tried in the course of other actions, and are certainly not the "main" action of the turn. But the count can be incremented only once in the course of each turn, so that "Instead of taking something for the third turn: ..." cannot happen on the very first turn because TAKE ALL causes three or more take actions.

(2) At the readjustment call, provided the action is not a silent one, the consecutive turns count is zeroed if the action does not match.

Why do we zero the count in rule (2)? Well, suppose that the player types OPEN BOX, so that the opening action takes place, and that it causes a further (non-silent) action, say examining the lid: and suppose that the action pattern we track the turn count for is "examining something". Then the following calls take place:

(a) `TrackActions(false)` while the action is "opening the box". Turn count zeroed by rule (1).
(b) `TrackActions(false)` while the action is "examining the lid". Turn count incremented by rule (1).
(c) `TrackActions(true)` while the action is "opening the box". Turn count zeroed by rule (2).
(d) `TrackActions(true)` while the action is "opening the box". Turn count zeroed by rule (2).

Thus rule (2) prevents us from counting this turn as the first of a sequence of turns in which examining something was the action, because it wasn't the main action of the turn.

A further complication is that we need to record the qualification, or not, even of silent actions, because testing rules like

    Instead of taking the top hat less than three times...

works by checking that (a) we are currently taking the top hat, and (b) have done so fewer than three times before. (b) uses the turn count described above; but (a) cannot simply look to see if that count is positive, since the action might be happening silently and thus not have contributed to the count; so we also record a flag to hold whether the action seems to be happening in this turn, silent or not.

And a still further complication is that out-of-world actions should never affect counts of turns for in-world actions. Thus,

    Every turn jumping for three turns: say "A demon appears!"

must not have the count to three broken by the player typing SAVE, which causes an out-of-world action but doesn't use a turn. The simplest way to deal with that would be to make `TrackActions` do nothing when `oow` is set, but then we would get rules like this wrong:

    Check requesting the pronoun meanings for the first time: ...

because it *does* make sense for OOW actions to have a count of how many times they've happened, even though they don't occur in simulated time. An OOW action can cause its own `ActionCurrentlyHappeningFlag` to be set, but it can't cause any other action's flag to be cleared, and nor can it zero the turns count for another action.

```
[ TrackActions readjust oow ct_0 ct_1 i;
    for (i=0: PastActionsI6Routines-->i: i++) {
        if ((PastActionsI6Routines-->i).call()) {
            ! Yes, the current action matches action pattern i:
            if (readjust) continue;
            (TimesActionHasHappened-->i)++;
            if (LastTurnActionHappenedOn-->i ~= turns + 5) {
                LastTurnActionHappenedOn-->i = turns + 5;
                ActionCurrentlyHappeningFlag->i = 1;
                if (keep_silent == false)
                    (TurnsActionHasBeenHappening-->i)++;
            }
        } else {
            ! No, the current action doesn't match action pattern i:
            if (oow == false) {
                if (keep_silent == false) { TurnsActionHasBeenHappening-->i = 0; }
                if (LastTurnActionHappenedOn-->i ~= turns + 5)
                    ActionCurrentlyHappeningFlag->i = 0;
            }
        }
    }
];
```

§**9. Storage.** The necessary array allocation.

```
Array TimesActionHasHappened-->(NO_PAST_TENSE_ACTIONS+1);
Array TurnsActionHasBeenHappening-->(NO_PAST_TENSE_ACTIONS+1);
Array LastTurnActionHappenedOn-->(NO_PAST_TENSE_ACTIONS+1);
Array ActionCurrentlyHappeningFlag->(NO_PAST_TENSE_ACTIONS+1);
Array past_chronological_record-->(NO_PAST_TENSE_CONDS+1);
Array present_chronological_record-->(NO_PAST_TENSE_CONDS+1);
```