LTL/f Final Code Book

Use the instructions below to assess incorrect responses.

There are 3 sets of instructions, for three question types:

- English to LTL
- LTL to English
- Trace Matching

The focus of this rubric is a set of tags, like **ImplicitG**. Each tag comes with an explanation of when to apply it.

When tagging, add comments to explain your reasoning in borderline cases.

English to LTL

Step 1:

Is the answer correct up to missing parentheses? (e.g., the student may have made a typo-level mistake omitting some parentheses) If yes, apply **Precedence** and **STOP.**

> *Rationale: We don't want to guess about removing and/or rearranging parentheses, because that opens a huge space of possibilities. We don't want to keep coding after a Precedence (hence STOP) because it's hard to get agreement afterward --- coders can interpret the ambiguity in different ways. We don't want to force the coder to use intuition to judge between "misconception" and a typo, hence this unified code. (e.g., "always {Red => (after (not Red) and after Red)}" could be a typo or a BadStateIndex misconception.)*
>
> *Wrong-parens example: eventually{Red} and eventually{not Red} and always{ { Red and after{not Red}} => after{always{not Red}} }*

Step 2:

Is this answer correct for one of our "reasonable variants"? If so, tag the problem-appropriate **Reasonable Variant (RV)** label and continue to code *relative to the variant* instead of to the "correct answer". Go back to Step 1 (Precedence) for the variant.

- **Example EL.0** : Applies to responses that interpret the phrase "there cannot be a next state with Green" as a constraint on all future states (if any).

- (2023-05-10) we have no reasonable variants for EmSys'22

  Rationale: In past studies, we often found that student answers appeared to be based on ambiguities in our English prompts. We identified a small number of "reasonable" mis-reads, which correspond to the RV labels. Many of those answers still involved mistakes relative to the variant. We want to quantify those mistakes too.

Step 3:

Apply semantic tags. Use the rubric below.

- **Ok** : Correct answer.

- **NA** : No answer (e.g. "I don't know")

- **?** : If a formula is very complex, very confusing, and/or if there are several potential interpretations about what went wrong, apply the ? tag.

  Rationale: Want to identify and exclude cases where something is so off that the coder was unsure if it was problem A or problem B etc. We can examine it later for a discussion
  (shows the extent of the confusion you can get from LTL ... imagine doing synthesis based off this formula, how far off is that result from the original English spec?!)

- **BadProp** : Mis-used/swapped a logical operator or atom (may have misunderstood the definition of "and" or our English description of a light being on or off).
  - Expected `a0 => a1` but learner wrote `a0 and a1`
  - Expected `a0 => a1` but learner wrote `a0 => a0`

- **BadStateIndex** : Wrote a correct term or subterm that applies at the wrong time/state index. This code should not be applied in cases where fan-out quantification (F, G, U) is omitted, included erroneously, or swapped, but only when the time index at which a subterm is evaluated is incorrect (either singly, or multiply in the scope of a temporal quantifier).
  - Expected `a0 U (a1 and F(a2))` but learner wrote `(a0 U a1) and F(a2)`
  - Expected `a0 and X(a1)` but learner wrote `a0 and a1`
  - Expected `X(a0)` but learner wrote `XX(a0)`
  - Expected `G(a0 => X(a1))` but learner wrote `G(a0 => a1)`.
  - Expected `F(X(y))` but learner wrote `F(y)`

*Rationale: excluding quantifier errors here, to avoid overlap with **ImplicitF, ImplicitG, and BadStateQuantification**.*

- **BadStateQuantification** : Mis-used/swapped a temporal quantifier (G, F, U).
    - Expected `F(a0)` but learner wrote `G(a0)` (wrong quantifier)
    - Expected `a0 U a1` but learner wrote `G(a0) U a1` (extra quantifier)

*Note***:** if the answer appears to use "after" (X) as a binary operator (e.g., expected `a0 => X(a1)` but learner wrote `a0 X a1`), then add a comment.

Rationale: this code is about mis-use of a fan-out quantifier, NOT about mis-use of X, since that is about a concrete index when doing the recursive descent, not exists/forall. Hence, we pulled out the binary-after misconception we noticed before to avoid tainting the semantics of the code.

- **CycleG** : Believes that G consumes a sequence of states in the trace, and thus only needs to be true for some states rather than all.
    - Expected `G((a0 => X(!a0)) & (!a0 => X(a0)))` but learner wrote `G(a0 => X(!a0))`
        [[alt explanation: "a0" is a variable that could be filled in as True or False, maybe differently in different states, and then the right formula is enough!]]

- **ImplicitF** : Assumed that a formula (or subformula) must happen at some point. This might be as simple as a missing F quantifier, but might also be the apparent belief that some subterm is forced to happen eventually.
    - Expected `F(a0)` but learner wrote `a0`
    - Expected `F(a0) and G(a0 => a1)` but learner wrote `G(a0 => a1)`

- **ImplicitG** : Assumed that a formula (or subformula) applies to all future states, rather than just the current state index.
    - Expected `G(a0)` but learner wrote `a0`
    - Expected `G(a0 => G(a1)) but learner wrote `G(a0 => a1)`

- **ImplicitPrefix**: Correctly specified the suffix of a formula, but left the prefix underconstrained. Overall, the formula accepts more traces than it should.
    - Expected `!a0 U a0&a1` but learner wrote `F(a0&a1)`
    - Expected `F(a0) and G(a0 => XG(!a0))` but learner wrote `F(a0 and XG(!a0))`

- **OtherImplicit** : Omitted some constraints from a formula, perhaps assuming that traces will have some baseline behavior if unconstrained (this might be "the light is off unless I turn it on", or "the light is on unless I turn it off" or "things stay as they are", etc.) **Do not** apply this tag when **ImplicitF** or **ImplicitG** or **ImplicitPrefix** would work.
    - Expected `a0 U G(!a0)` but learner wrote `a0 and F(G(!a0))`
        Rationale: [EDIT: 2023-11-17, new tag ImplicitPrefix seems to cover this thought!]
        We noticed that there were some answers that underconstrained the problem but did not

fall into ImplicitG/ImplicitF (such as saying "eventually a0" and never actually forcing a0 to remain off until that point). Some, but not all, of these could possibly be captured by an ImplicitU code, but that seemed confusing since the second subterm would be implicit and unstated.

- **SeqX** : Misunderstood X(a) as a "spreading" operator that starts at the current time and constrains the next state too. Similarly XX(a) constrains 3 states.
    - Expected `!F(a0 & X(a0) & XX(a0))` but learner wrote `!F(XX(a0))`

- **ExclusiveU** : Assumed that an Until is satisfied only when both the right subterm and the negation of the left subterm hold.
    - Expected `x U ((not x) and y)` but subject wrote `x U y`

- **TraceSplitU** : Believes that the left and right subterms of an Until are evaluated on two *disjoint* traces --- a prefix of the full trace and its infinite suffix. Put another way, believes that Until restricts the scope of its subformulas.
    - Expected `F(Blue & X(F(Blue)))` but subject wrote `F(Blue) U F(Blue)`
    - Expected `Blue U !Blue` but subject wrote `(G(Blue)) U !Blue`

2023-11-17: Splitting a trace makes no sense in LTL because all traces must be infinite (still, learners make the mistake). But in LTLf this is much more reasonable!

- **WeakU** : Confused U with weak-U.
    - Expected `a0 U a1`

Rationale: This code is meant to capture confusion about the fact that `until` forces the eventual truth of its 2nd argument. It is possible there is some overlap with Miscomm tags here: if students can interpret our problem sentences as admitting vacuous truth...)

- **Last** : (LTLf-specific) Applies to responses that make one of two mistakes: (1) claiming that LTLf cannot express a last state, (2) claiming that LTL can express a last state.

- **LenX** : (LTLf) Applies to responses that require too many or too few states
    - Expected `!X(a0)` but learner wrote `X(!a0)`
    - Expected `X(Xw(a0))` but learner wrote `Xw(X(a0))`

LTL to English

Use the English to LTL tags with one addition:

- **Scope** : Mis-read parenthesis.
    - Read `F(a0) && F(a1)` as `F(a0 && a1)`

Notes:

- **RV** does not apply
- **Precedence** does not apply, but **Scope** does

Trace Satisfaction

Use the same tags as for LTL to English.

Notes:

- If there are several possible reasons for a wrong answer and the response comes with no explanation, apply the **?** tag.

    Example: trace = {} {R} {} {} {}* ; formula = X(R) ; answer = No match
    could be **BadStateIndex** or **SeqX**