

Goals as Reward-Generating Programs Domain Specific Language

October 6, 2023

1 DSL Grammar Definitions

A game is defined by a name, and is expected to be valid in a particular domain, also referenced by a name. A game is defined by four elements, two of them mandatory, and two optional. The mandatory ones are the $\langle constraints \rangle$ section, which defines gameplay preferences, and the $\langle scoring \rangle$ section, which defines how gameplay preferences are counted to arrive at a score for the player in the game. The optional ones are the $\langle setup \rangle$ section, which defines how the environment must be prepared before gameplay can begin, and the $\langle terminal \rangle$ conditions, which specify when and how the game ends.

```
 $\langle game \rangle ::= (\text{define (game } \langle ID \rangle \langle$   
   $(:\text{domain } \langle ID \rangle)$   
   $(:\text{setup } \langle setup \rangle)$   
   $(:\text{constraints } \langle constraints \rangle)$   
   $(:\text{terminal } \langle terminal \rangle)$   
   $(:\text{scoring } \langle scoring \rangle)$   
   $\rangle)$ 
```

$\langle id \rangle ::= /[a-z0-9][a-z0-9]+/ \#$ a letter or digit, followed by one or more letters, digits, or dashes

We will now proceed to introduce and define the syntax for each of these sections, followed by the non-grammar elements of our domain: predicates, functions, and types. Finally, we provide a mapping between some aspects of our gameplay preference specification and linear temporal logic (LTL) operators.

1.1 Setup

The setup section specifies how the environment must be transformed from its deterministic initial conditions to a state gameplay can begin at. Currently, a particular environment room always appears in the same initial conditions, in terms of which objects exist and where they are placed. Participants in our experiment could, but did not have to, specify how the room must be setup so that their game could be played.

The initial $\langle setup \rangle$ element can expand to conjunctions, disjunctions, negations, or quantifications of itself, and then to the $\langle setup\text{-statement} \rangle$ rule. $\langle setup\text{-statement} \rangle$ elements specify two different types of setup conditions: either those that must be conserved through gameplay ('game-conserved'), or those that are optional through gameplay ('game-optional'). These different conditions arise as some setup elements must be maintain through gameplay (for example, a participant specified to place a bin on the bed to throw balls into, it shouldn't move unless specified otherwise), while other setup elements can or must change (if a participant specified to set the balls on the desk to throw them, an agent will have to pick them up (and off the desk) in order to throw them).

Inside the $\langle setup\text{-statement} \rangle$ tags we find $\langle super\text{-predicate} \rangle$ elements, which are logical operations and quantifications over other $\langle super\text{-predicate} \rangle$ elements, function comparisons ($\langle function\text{-comparison} \rangle$, which like predicates also resolve to a truth value), and predicates ($\langle predicate \rangle$). Function comparisons usually consist of a comparison operator and two arguments, which can either be the evaluation of a function or a number. The one exception is the case where the comparison operator is the equality operator ($=$), in which case any number of arguments can be provided. Finally, the $\langle predicate \rangle$ element expands to a predicate acting on one or more objects or variables. For a full list of the predicates we found ourselves using so far, see subsection 2.1 (Predicates).

```
 $\langle setup \rangle ::= (\text{and } \langle setup \rangle \langle setup \rangle^+) \#$  A setup can be expanded to a conjunction, a disjunction, a quantification, or a setup  
statement (see below).  
|  $(\text{or } \langle setup \rangle \langle setup \rangle^+)$   
|  $(\text{not } \langle setup \rangle)$   
|  $(\text{exists } (\langle typed\ list(variable) \rangle) \langle setup \rangle)$   
|  $(\text{forall } (\langle typed\ list(variable) \rangle) \langle setup \rangle)$   
|  $\langle setup\text{-statement} \rangle$ 
```

$\langle \text{setup-statement} \rangle ::= \#$ A setup statement specifies that a predicate is either optional during gameplay or must be preserved during gameplay.

- | $(\text{game-conserved } \langle \text{super-predicate} \rangle)$
- | $(\text{game-optional } \langle \text{super-predicate} \rangle)$

$\langle \text{super-predicate} \rangle ::= \#$ A super-predicate is a conjunction, disjunction, negation, or quantification over another super-predicate. It can also be directly a function comparison or a predicate.

- | $(\text{and } \langle \text{super-predicate} \rangle^+)$
- | $(\text{or } \langle \text{super-predicate} \rangle^+)$
- | $(\text{not } \langle \text{super-predicate} \rangle)$
- | $(\text{exists } (\langle \text{typed list(variable)} \rangle) \langle \text{super-predicate} \rangle)$
- | $(\text{forall } (\langle \text{typed list(variable)} \rangle) \langle \text{super-predicate} \rangle)$
- | $\langle \text{f-comp} \rangle$
- | $\langle \text{predicate} \rangle$

$\langle \text{function-comparison} \rangle ::= \#$ A function comparison: either comparing two function evaluations, or checking that two or more functions evaluate to the same result.

- | $(\langle \text{comp-op} \rangle \langle \text{function-eval-or-number} \rangle \langle \text{function-eval-or-number} \rangle)$
- | $(= \langle \text{function-eval-or-number} \rangle^+)$

$\langle \text{comp-op} \rangle ::= \langle \mid \langle = \mid = \mid \rangle \mid \rangle = \#$ Any of the comparison operators.

$\langle \text{function-eval-or-number} \rangle ::= \langle \text{function-eval} \rangle \mid \langle \text{comparison-arg-number} \rangle$

$\langle \text{comparison-arg-number} \rangle ::= \langle \text{number} \rangle$

$\langle \text{number} \rangle ::= /-?\d*\.? \d+ / \#$ A number, either an integer or a float.

$\langle \text{function-eval} \rangle ::= \#$ See valid expansions in a separate section below

$\langle \text{variable-list} \rangle ::= (\langle \text{variable-def} \rangle^+) \#$ One or more variables definitions, enclosed by parentheses.

$\langle \text{variable-def} \rangle ::= \langle \text{variable-type-def} \rangle \mid \langle \text{color-variable-type-def} \rangle \mid \langle \text{orientation-variable-type-def} \rangle \mid \langle \text{side-variable-type-def} \rangle \#$
Colors, sides, and orientations are special types as they are not interchangeable with objects.

$\langle \text{variable-type-def} \rangle ::= \langle \text{variable} \rangle^+ - \langle \text{type-def} \rangle \#$ Each variable is defined by a variable (see next) and a type (see after).

$\langle \text{color-variable-type-def} \rangle ::= \langle \text{color-variable} \rangle^+ - \langle \text{color-type-def} \rangle \#$ A color variable is defined by a variable (see below) and a color type.

$\langle \text{orientation-variable-type-def} \rangle ::= \langle \text{orientation-variable} \rangle^+ - \langle \text{orientation-type-def} \rangle \#$ An orientation variable is defined by a variable (see below) and an orientation type.

$\langle \text{side-variable-type-def} \rangle ::= \langle \text{side-variable} \rangle^+ - \langle \text{side-type-def} \rangle \#$ A side variable is defined by a variable (see below) and a side type.

$\langle \text{variable} \rangle ::= /\?[a-w][a-z0-9]*/ \#$ a question mark followed by a lowercase a-w, optionally followed by additional letters or numbers.

$\langle \text{color-variable} \rangle ::= /\?[x0-9]*/ \#$ a question mark followed by an x and an optional number.

$\langle \text{orientation-variable} \rangle ::= /\?[y0-9]*/ \#$ a question mark followed by an y and an optional number.

$\langle \text{side-variable} \rangle ::= /\?[z0-9]*/ \#$ a question mark followed by an z and an optional number.

$\langle \text{type-def} \rangle ::= \langle \text{object-type} \rangle \mid \langle \text{either-types} \rangle \#$ A variable type can either be a single name, or a list of type names, as specified below

$\langle color\text{-}type\text{-}def \rangle ::= \langle color\text{-}type \rangle \mid \langle either\text{-}color\text{-}types \rangle$ # A color variable type can either be a single color name, or a list of color names, as specified below

$\langle orientation\text{-}type\text{-}def \rangle ::= \langle orientation\text{-}type \rangle \mid \langle either\text{-}orientation\text{-}types \rangle$ # An orientation variable type can either be a single orientation name, or a list of orientation names, as specified below

$\langle side\text{-}type\text{-}def \rangle ::= \langle side\text{-}type \rangle \mid \langle either\text{-}side\text{-}types \rangle$ # A side variable type can either be a single side name, or a list of side names, as specified below

$\langle either\text{-}types \rangle ::= (\text{either } \langle object\text{-}type \rangle^+)$

$\langle either\text{-}color\text{-}types \rangle ::= (\text{either } \langle color \rangle^+)$

$\langle either\text{-}orientation\text{-}types \rangle ::= (\text{either } \langle orientation \rangle^+)$

$\langle either\text{-}side\text{-}types \rangle ::= (\text{either } \langle side \rangle^+)$

$\langle object\text{-}type \rangle ::= \langle name \rangle$

$\langle name \rangle ::= /[A-Za-z][A-Za-z0-9_]+/$ # a letter, followed by one or more letters, digits, or underscores

$\langle color\text{-}type \rangle ::= \text{'color'}$

$\langle color \rangle ::= \text{'blue'} \mid \text{'brown'} \mid \text{'gray'} \mid \text{'green'} \mid \text{'orange'} \mid \text{'pink'} \mid \text{'purple'} \mid \text{'red'} \mid \text{'tan'} \mid \text{'white'} \mid \text{'yellow'}$

$\langle orientation\text{-}type \rangle ::= \text{'orientation'}$

$\langle orientation \rangle ::= \text{'diagonal'} \mid \text{'sideways'} \mid \text{'upright'} \mid \text{'upside_down'}$

$\langle side\text{-}type \rangle ::= \text{'side'}$

$\langle side \rangle ::= \text{'back'} \mid \text{'front'} \mid \text{'left'} \mid \text{'right'}$

$\langle predicate \rangle ::=$ # See valid expansions in a separate section below

$\langle predicate\text{-}or\text{-}function\text{-}term \rangle ::= \langle object\text{-}name \rangle \mid \langle variable \rangle$ # A predicate or function term can either be an object name (from a small list allowed to be directly referred to) or a variable.

$\langle predicate\text{-}or\text{-}function\text{-}color\text{-}term \rangle ::= \langle color \rangle \mid \langle color\text{-}variable \rangle$

$\langle predicate\text{-}or\text{-}function\text{-}orientation\text{-}term \rangle ::= \langle orientation \rangle \mid \langle orientation\text{-}variable \rangle$

$\langle predicate\text{-}or\text{-}function\text{-}side\text{-}term \rangle ::= \langle side \rangle \mid \langle side\text{-}variable \rangle$

$\langle predicate\text{-}or\text{-}function\text{-}type\text{-}term \rangle ::= \langle object\text{-}type \rangle \mid \langle variable \rangle$

$\langle object_name \rangle ::= \text{'agent'} \mid \text{'bed'} \mid \text{'desk'} \mid \text{'door'} \mid \text{'floor'} \mid \text{'main_light_switch'} \mid \text{'mirror'} \mid \text{'room_center'} \mid \text{'rug'} \mid \text{'side_table'}$
 $\mid \text{'bottom_drawer'} \mid \text{'bottom_shelf'} \mid \text{'east_sliding_door'} \mid \text{'east_wall'} \mid \text{'north_wall'} \mid \text{'south_wall'} \mid \text{'top_drawer'} \mid$
 $\text{'top_shelf'} \mid \text{'west_sliding_door'} \mid \text{'west_wall'}$

1.2 Gameplay Preferences

The gameplay preferences specify the core of a game's semantics, capturing how a game should be played by specifying temporal constraints over predicates. The name for the overall element, $\langle constraints \rangle$, is inherited from the PDDL element with the same name.

The $\langle constraints \rangle$ elements expands into one or more preference definitions, which are defined using the $\langle pref\text{-}def \rangle$ element. A $\langle pref\text{-}def \rangle$ either expands to a single preference ($\langle preference \rangle$), or to a $\langle pref\text{-}forall \rangle$ element, which specifies variants of the same preference for different objects, which can be treated differently in the scoring section. A $\langle preference \rangle$ is defined by a

name and a $\langle \textit{preference-quantifier} \rangle$, which expands to an optional quantification (exists, forall, or neither), inside of which we find the $\langle \textit{preference-body} \rangle$.

A $\langle \textit{preference-body} \rangle$ expands into one of two options: The first is a set of conditions that should be true at the end of gameplay, using the $\langle \textit{at-end} \rangle$ operator. Inside an $\langle \textit{at-end} \rangle$ we find a $\langle \textit{super-predicate} \rangle$, which like in the setup section, expands to logical operations or quantifications over other $\langle \textit{super-predicate} \rangle$ elements, function comparisons, or predicates.

The second option is specified using the $\langle \textit{then} \rangle$ syntax, which defines a series of temporal conditions that should hold over a sequence of states. Under a $\langle \textit{then} \rangle$ operator, we find two or more sequence functions ($\langle \textit{seq-func} \rangle$), which define the specific conditions that must hold and how many states we expect them to hold for. We assume that there are no unaccounted states between the states accounted for by the different operators – in other words, the $\langle \textit{then} \rangle$ operators expects to find a sequence of contiguous states that satisfy the different sequence functions. The operators under a $\langle \textit{then} \rangle$ operator map onto linear temporal logic (LTL) operators, see ?? (??) for the mapping and examples.

The $\langle \textit{once} \rangle$ operator specifies a predicate that must hold for a single world state. If a $\langle \textit{once} \rangle$ operators appears as the first operator of a $\langle \textit{then} \rangle$ definition, and a sequence of states S_a, S_{a+1}, \dots, S_b satisfy the $\langle \textit{then} \rangle$ operator, it could be the case that the predicate is satisfied before this sequence of states (e.g. by S_{a-1}, S_{a-2} , and so forth). However, only the final such state, S_a , is required for the preference to be satisfied. The same could be true at the end of the sequence: if a $\langle \textit{then} \rangle$ operator ends with a $\langle \textit{once} \rangle$ term, there could be other states after the final state (S_{b+1}, S_{b+2} , etc.) that satisfy the predicate in the $\langle \textit{once} \rangle$ operator, but only one is required. The $\langle \textit{once-measure} \rangle$ operator is a slight variation of the $\langle \textit{once} \rangle$ operator, which in addition to a predicate, takes in a function evaluation, and measures the value of the function evaluated at the state that satisfies the preference. This function value can then be used in the scoring definition, see subsection 1.4 (Scoring).

A second type of operator that exists is the $\langle \textit{hold} \rangle$ operator. It specifies that a predicate must hold true in every state between the one in which the previous operator is satisfied, and until one in which the next operator is satisfied. If a $\langle \textit{hold} \rangle$ operator appears at the beginning or an end of a $\langle \textit{then} \rangle$ sequence, it can be satisfied by a single state, Otherwise, it must be satisfied until the next operator is satisfied. For example, in the minimal definition below:

```
(then
  (once (pred_a))
  (hold (pred_b))
  (once (pred_c))
)
```

To find a sequence of states S_a, S_{a+1}, \dots, S_b that satisfy this $\langle \textit{then} \rangle$ operator, the following conditions must hold true: (1) pred_a is true at state S_a , (2) pred_b is true in all states $S_{a+1}, S_{a+2}, \dots, S_{b-2}, S_{b-1}$, and (3) pred_c is true in state S_b . There is no minimal number of states that the hold predicate must hold for.

The last operator is $\langle \textit{hold-while} \rangle$, which offers a variation of the $\langle \textit{hold} \rangle$ operator. A $\langle \textit{hold-while} \rangle$ receives at least two predicates. The first acts the same as predicate in a $\langle \textit{hold} \rangle$ operator. The second (and third, and any subsequent ones), must hold true for at least state while the first predicate holds, and must occur in the order specified. In the example above, if we substitute (hold (pred_b)) for (hold-while (pred_b) (pred_d) (pred_e)), we now expect that in addition to ped_b being true in all states $S_{a+1}, S_{a+2}, \dots, S_{b-2}, S_{b-1}$, that there is some state $S_d, d \in [a + 1, b - 1]$ where pred_d holds, and another state, $S_e, e \in [d + 1, b - 1]$ where pred_e holds.

$\langle \textit{constraints} \rangle ::= \langle \textit{pref-def} \rangle \mid (\text{and } \langle \textit{pref-def} \rangle^+) \#$ One or more preferences.

$\langle \textit{pref-def} \rangle ::= \langle \textit{pref-forall} \rangle \mid \langle \textit{preference} \rangle \#$ A preference definitions expands to either a forall quantification (see below) or to a preference.

$\langle \textit{pref-forall} \rangle ::= (\text{forall } \langle \textit{variable-list} \rangle \langle \textit{preference} \rangle) \#$ this syntax is used to specify variants of the same preference for different objects, which differ in their scoring. These are specified using the $\langle \textit{pref-name-and-types} \rangle$ syntax element's optional types, see scoring below.

$\langle \textit{preference} \rangle ::= (\text{preference } \langle \textit{name} \rangle \langle \textit{preference-quantifier} \rangle) \#$ A preference is defined by a name and a quantifier that includes the preference body.

$\langle \textit{preference-quantifier} \rangle ::= \#$ A preference can quantify existentially or universally over one or more variables, or none.

```
| (exists (⟨variable-list⟩) ⟨preference-body⟩)
| (forall (⟨variable-list⟩) ⟨preference-body⟩)
| ⟨preference-body⟩
```

$\langle \textit{preference-body} \rangle ::= \langle \textit{then} \rangle \mid \langle \textit{at-end} \rangle$

$\langle at-end \rangle ::= (\text{at-end } \langle super-predicate \rangle) \#$ Specifies a predicate that should hold in the terminal state.

$\langle then \rangle ::= (\text{then } \langle seq-func \rangle \langle seq-func \rangle^+) \#$ Specifies a series of conditions that should hold over a sequence of states – see below for the specific operators ($\langle seq-func \rangle$ s), and Section 2 for translation of these definitions to linear temporal logic (LTL).

$\langle seq-func \rangle ::= \langle once \rangle \mid \langle once-measure \rangle \mid \langle hold \rangle \mid \langle hold-while \rangle \#$ Four of these temporal sequence functions currently exist:

$\langle once \rangle ::= (\text{once } \langle super-predicate \rangle) \#$ The predicate specified must hold for a single world state.

$\langle once-measure \rangle ::= (\text{once } \langle super-predicate \rangle \langle function-eval \rangle) \#$ The predicate specified must hold for a single world state, and record the value of the function evaluation, to be used in scoring.

$\langle hold \rangle ::= (\text{hold } \langle super-predicate \rangle) \#$ The predicate specified must hold for every state between the previous temporal operator and the next one.

$\langle hold-while \rangle ::= (\text{hold-while } \langle super-predicate \rangle \langle super-predicate \rangle^+) \#$ The first predicate specified must hold for every state between the previous temporal operator and the next one. While it does, at least one state must satisfy each of the predicates specified in the second argument onward

For the full specification of the $\langle super-predicate \rangle$ element, see subsection 1.1 (Setup) above.

1.3 Terminal Conditions

Specifying explicit terminal conditions is optional, and while some of our participants chose to do so, many did not. Conditions explicitly specified in this section terminate the game. If none are specified, a game is assumed to terminate whenever the player chooses to end the game.

The terminal conditions expand from the $\langle terminal \rangle$ element, which can expand to logical conditions on nested $\langle terminal \rangle$ elements, or to a terminal comparison. The terminal comparison ($\langle terminal-comp \rangle$) expands to one of three different types of comparisons: $\langle terminal_time_comp \rangle$, a comparison between the total time spent in the game ($\langle total-time \rangle$) and a time number token, $\langle terminal-score-comp \rangle$, a comparison between the total score ($\langle total-score \rangle$) and a score number token, or $\langle terminal-pref-count-comp \rangle$, a comparison between a scoring expression ($\langle scoring-expr \rangle$, see below) and a preference count number token. In most cases, the scoring expression is a preference counting operation.

$\langle terminal \rangle ::= \#$ The terminal condition is specified by a conjunction, disjunction, negation, or comparison (see below).
 $\mid (\text{and } \langle terminal \rangle^+)$
 $\mid (\text{or } \langle terminal \rangle^+)$
 $\mid (\text{not } \langle terminal \rangle)$
 $\mid \langle terminal-comp \rangle$

$\langle terminal-comp \rangle ::= \#$ We support three types of terminal comparisons:
 $\mid \langle terminal-time-comp \rangle$
 $\mid \langle terminal-score-comp \rangle$
 $\mid \langle terminal-pref-count-comp \rangle$

$\langle terminal-time-comp \rangle ::= (\langle comp-op \rangle (\text{total-time}) \langle time-number \rangle) \#$ The total time of the game must satisfy the comparison.

$\langle terminal-score-comp \rangle ::= (\langle comp-op \rangle (\text{total-score}) \langle score-number \rangle) \#$ The total score of the game must satisfy the comparison.

$\langle terminal-pref-count-comp \rangle ::= (\langle comp-op \rangle \langle scoring-expr \rangle \langle preference-count-number \rangle) \#$ The number of times the preference specified by the name and types must satisfy the comparison.

$\langle time-number \rangle ::= \langle number \rangle \#$ Separate type so the we can learn a separate distribution over times than, say, scores.

$\langle score-number \rangle ::= \langle number \rangle$

$\langle preference-count-number \rangle ::= \langle number \rangle$

$\langle comp-op \rangle ::= \langle | \langle = | = | \rangle | \rangle =$

For the full specification of the $\langle scoring-expr \rangle$ element, see subsection 1.4 (Scoring) below.

1.4 Scoring

Scoring rules specify how to count preferences (count once, once for each unique objects that fulfill the preference, each time a preference is satisfied, etc.), and the arithmetic to combine preference counts to a final score in the game.

A $\langle scoring-expr \rangle$ can be defined by arithmetic operations on other scoring expressions, references to the total time or total score (for instance, to provide a bonus if a certain score is reached), comparisons between scoring expressions ($\langle scoring-comp \rangle$), or by preference evaluation rules. Various preference evaluation modes can expand the $\langle preference-eval \rangle$ rule, see the full list and descriptions below.

$\langle scoring \rangle ::= \langle scoring-expr \rangle \#$ The scoring conditions maximize a scoring expression.

$\langle scoring-expr \rangle ::= \#$ A scoring expression can be an arithmetic operation over other scoring expressions, a reference to the total time or score, a comparison, or a preference scoring evaluation.

- | $\langle scoring-external-maximize \rangle$
- | $\langle scoring-external-minimize \rangle$
- | $(\langle multi-op \rangle \langle scoring-expr \rangle^+) \#$ Either addition or multiplication.
- | $(\langle binary-op \rangle \langle scoring-expr \rangle \langle scoring-expr \rangle) \#$ Either division or subtraction.
- | $(- \langle scoring-expr \rangle)$
- | $(total-time)$
- | $(total-score)$
- | $\langle scoring-comp \rangle$
- | $\langle preference-eval \rangle$
- | $\langle scoring-number-value \rangle$

$\langle scoring-external-maximize \rangle ::= (external-forall-maximize \langle scoring-expr \rangle) \#$ For any preferences under this expression inside a (forall ...), score only for the single externally-quantified object that maximizes this scoring expression.

$\langle scoring-external-minimize \rangle ::= (external-forall-minimize \langle scoring-expr \rangle) \#$ For any preferences under this expression inside a (forall ...), score only for the single externally-quantified object that minimizes this scoring expression.

$\langle scoring-comp \rangle ::= \#$ A scoring comparison: either comparing two expressions, or checking that two ore more expressions are equal.

- | $(\langle comp-op \rangle \langle scoring-expr \rangle \langle scoring-expr \rangle)$
- | $(= \langle scoring-expr \rangle^+)$

$\langle preference-eval \rangle ::= \#$ A preference evaluation applies one of the scoring operators (see below) to a particular preference referenced by name (with optional types).

- | $\langle count \rangle$
- | $\langle count-overlapping \rangle$
- | $\langle count-once \rangle$
- | $\langle count-once-per-objects \rangle$
- | $\langle count-measure \rangle$
- | $\langle count-unique-positions \rangle$
- | $\langle count-same-positions \rangle$
- | $\langle count-once-per-external-objects \rangle$

$\langle count \rangle ::= (count \langle pref-name-and-types \rangle) \#$ Count how many times the preference is satisfied by non-overlapping sequences of states.

$\langle count-overlapping \rangle ::= (count-overlapping \langle pref-name-and-types \rangle) \#$ Count how many times the preference is satisfied by overlapping sequences of states.

$\langle count-once \rangle ::= (count-once \langle pref-name-and-types \rangle) \#$ Count whether or not this preference was satisfied at all.

$\langle \text{count-once-per-objects} \rangle ::= (\text{count-once-per-objects } \langle \text{pref-name-and-types} \rangle) \#$ Count once for each unique combination of objects quantified in the preference that satisfy it.

$\langle \text{count-measure} \rangle ::= (\text{count-measure } \langle \text{pref-name-and-types} \rangle) \#$ Can only be used in preferences including a $\langle \text{once-measure} \rangle$ modal, maps each preference satisfaction to the value of the function evaluation in the $\langle \text{once-measure} \rangle$.

$\langle \text{count-unique-positions} \rangle ::= (\text{count-unique-positions } \langle \text{pref-name-and-types} \rangle) \#$ Count how many times the preference was satisfied with quantified objects that remain stationary within each preference satisfaction, and have different positions between different satisfactions.

$\langle \text{count-same-positions} \rangle ::= (\text{count-same-positions } \langle \text{pref-name-and-types} \rangle) \#$ Count how many times the preference was satisfied with quantified objects that remain stationary within each preference satisfaction, and have (approximately) the same position between different satisfactions.

$\langle \text{count-once-per-external-objects} \rangle ::= (\text{count-once-per-external-objects } \langle \text{pref-name-and-types} \rangle) \#$ Similarly to count-once-per-objects, but counting only for each unique object or combination of objects quantified in the (forall ...) block including this preference.

$\langle \text{pref-name-and-types} \rangle ::= \langle \text{name} \rangle \langle \text{pref-object-type} \rangle^*$ # The optional $\langle \text{pref-object-type} \rangle$ s are used to specify a particular instance of the preference for a given object, see the $\langle \text{pref-forall} \rangle$ syntax above.

$\langle \text{pref-object-type} \rangle ::= : \langle \text{type-name} \rangle \#$ The optional type name specification for the above syntax. For example, `pref-name:dodgeball` would refer to the preference where the first quantified object is a dodgeball.

$\langle \text{scoring-number-value} \rangle ::= \langle \text{number} \rangle$

2 Non-Grammar Definitions

2.1 Predicates

The following section describes the predicates we define. Predicates operate over a specified number of arguments, which can be variables or object names, and return a boolean value (true/false).

```
(above <arg1> <arg2>) [5 references] ; Is the first object above the second object?
(adjacent <arg1> <arg2>) [78 references] ; Are the two objects adjacent? [will probably be
    implemented as distance below some threshold]
(adjacent_side <3 or 4 arguments>) [15 references] ; Are the two objects adjacent on the sides
    specified? Specifying a side for the second object is optional, allowing to specify <obj1> <
    side1> <obj2> or <obj1> <side1> <obj2> <side2>
(agent_crouches ) [2 references] ; Is the agent crouching?
(agent_holds <arg1>) [327 references] ; Is the agent holding the object?
(between <arg1> <arg2> <arg3>) [7 references] ; Is the second object between the first object
    and the third object?
(broken <arg1>) [2 references] ; Is the object broken?
(equal_x_position <arg1> <arg2>) [2 references] ; Are these two objects (approximately) in the
    same x position? (in our environment, x, z are spatial coordinates, y is the height)
(equal_z_position <arg1> <arg2>) [5 references] ; Are these two objects (approximately) in the
    same z position? (in our environment, x, z are spatial coordinates, y is the height)
(faces <arg1> <arg2>) [6 references] ; Is the front of the first object facing the front of the
    second object?
(game_over ) [4 references] ; Is this the last state of gameplay?
(game_start ) [3 references] ; Is this the first state of gameplay?
(in <arg1> <arg2>) [121 references] ; Is the second argument inside the first argument? [a
    containment check of some sort, for balls in bins, for example]
(in_motion <arg1>) [315 references] ; Is the object in motion?
(is_setup_object <arg1>) [13 references] ; Is this the object of the same type referenced in
    the setup?
(object_orientation <arg1> <arg2>) [14 references] ; Is the first argument, an object, in the
    orientation specified by the second argument? Used to check if an object is upright or
    upside down
```

```

(on <arg1> <arg2>) [168 references] ; Is the second object on the first one?
(open <arg1>) [3 references] ; Is the object open? Only valid for objects that can be opened,
    such as drawers.
(opposite <arg1> <arg2>) [4 references] ; So far used only with walls, or sides of the room, to
    specify two walls opposite each other in conjunction with other predicates involving these
    walls
(rug_color_under <arg1> <arg2>) [11 references] ; Is the color of the rug under the object (
    first argument) the color specified by the second argument?
(same_color <arg1> <arg2>) [23 references] ; If two objects, do they have the same color? If
    one is a color, does the object have that color?
(same_object <arg1> <arg2>) [7 references] ; Are these two variables bound to the same object?
(same_type <arg1> <arg2>) [14 references] ; Are these two objects of the same type? Or if one
    is a direct reference to a type, is this object of that type?
(toggled_on <arg1>) [4 references] ; Is this object toggled on?
(touch <arg1> <arg2>) [48 references] ; Are these two objects touching?

```

2.2 Functions

he following section describes the functions we define. Functions operate over a specified number of arguments, which can be variables or object names, and return a number.

```

(building_size <arg1>) [2 references] ; Takes in an argument of type building, and returns how
    many objects comprise the building (as an integer).
(distance <arg1> <arg2>) [114 references] ; Takes in two arguments of type object, and returns
    the distance between the two objects (as a floating point number).
(distance_side <arg1> <arg2> <arg3>) [6 references] ; Similarly to the adjacent_side predicate,
    but applied to distance. Takes in three or four arguments, either <obj1> <side1> <obj2> or
    <obj1> <side1> <obj2> <side2>, and returns the distance between the first object on the side
    specified to the second object (optionally to its specified side).
(x_position <arg1>) [4 references] ; Takes in an argument of type object, and returns the x
    position of the object (as a floating point number).

```

2.3 Types

The types are currently not defined as part of the grammar, other than the small list of *<object-name>* tokens that can be directly referred to, and are marked with an asterisk below. The following enumerates all expansions of the various *<type>* rules:

```

game_object [33 references] ; Parent type of all objects
agent* [90 references] ; The agent
building [20 references] ; Not a real game object, but rather, a way to refer to structures the
    agent builds
----- Blocks -----
block [28 references] ; Parent type of all block types:
bridge_block [11 references]
bridge_block_green [0 references]
bridge_block_pink [0 references]
bridge_block_tan [0 references]
cube_block [38 references]
cube_block_blue [8 references]
cube_block_tan [1 reference]
cube_block_yellow [8 references]
cylindrical_block [11 references]
cylindrical_block_blue [0 references]
cylindrical_block_green [0 references]
cylindrical_block_tan [0 references]
flat_block [5 references]
flat_block_gray [0 references]
flat_block_tan [0 references]
flat_block_yellow [0 references]
pyramid_block [13 references]

```



```

pyramid_block_blue [3 references]
pyramid_block_red [2 references]
pyramid_block_yellow [2 references]
tall_cylindrical_block [7 references]
tall_cylindrical_block_green [0 references]
tall_cylindrical_block_tan [0 references]
tall_cylindrical_block_yellow [0 references]
tall_rectangular_block [0 references]
tall_rectangular_block_blue [0 references]
tall_rectangular_block_green [0 references]
tall_rectangular_block_tan [0 references]
triangle_block [3 references]
triangle_block_blue [0 references]
triangle_block_green [0 references]
triangle_block_tan [0 references]
----- Balls -----
ball [40 references] ; Parent type of all ball types:
beachball [23 references]
basketball [18 references]
dodgeball [108 references]
dodgeball_blue [6 references]
dodgeball_red [4 references]
dodgeball_pink [8 references]
golfball [25 references]
golfball_green [3 references]
golfball_white [0 references]
----- Colors -----
color [6 references] ; Likewise, not a real game object, mostly used to refer to the color of
    the rug under an object
blue [6 references]
brown [5 references]
gray [0 references]
green [8 references]
orange [3 references]
pink [19 references]
purple [4 references]
red [8 references]
tan [2 references]
white [1 reference]
yellow [14 references]
----- Furniture -----
bed* [51 references]
blinds [2 references] ; The blinds on the windows
desk* [40 references]
desktop [6 references]
main_light_switch* [3 references] ; The main light switch on the wall
side_table* [4 references] ; The side table/nightstand next to the bed
shelf_desk [2 references] ; The shelves under the desk
----- Large moveable/interactable objects -----
book [11 references]
chair [18 references]
laptop [7 references]
pillow [14 references]
teddy_bear [14 references]
----- Orientations -----
diagonal [1 reference]
sideways [2 references]
upright [10 references]
upside_down [1 reference]
----- Ramps -----
ramp [0 references] ; Parent type of all ramp types:
curved_wooden_ramp [17 references]

```

```

triangular_ramp [10 references]
triangular_ramp_green [1 reference]
triangular_ramp_tan [0 references]
----- Receptacles -----
doggie_bed [26 references]
hexagonal_bin [123 references]
drawer [5 references] ; Either drawer in the side table
bottom_drawer* [0 references] ; The bottom of the two drawers in the nightstand near the bed.
top_drawer* [6 references] ; The top of the two drawers in the nightstand near the bed.
----- Room features -----
door* [9 references] ; The door out of the room
floor* [26 references]
mirror* [0 references]
poster* [0 references]
room_center* [0 references]
rug* [37 references]
shelf [10 references]
bottom_shelf* [1 reference]
top_shelf* [5 references]
sliding_door [2 references] ; The sliding doors on the south wall (big windows)
east_sliding_door* [1 reference] ; The eastern of the two sliding doors (the one closer to the
desk)
west_sliding_door* [0 references] ; The western of the two sliding doors (the one closer to the
bed)
wall [17 references] ; Any of the walls in the room
east_wall* [0 references] ; The wall behind the desk
north_wall* [1 reference] ; The wall with the door to the room
south_wall* [2 references] ; The wall with the sliding doors
west_wall* [3 references] ; The wall the bed is aligned to
----- Small objects -----
alarm_clock [8 references]
cellphone [6 references]
cd [6 references]
credit_card [1 reference]
key_chain [5 references]
lamp [2 references]
mug [3 references]
pen [2 references]
pencil [2 references]
watch [2 references]
----- Sides -----
back [3 references]
front [9 references]
left [3 references]
right [2 references]

```

3 Modal Definitions in Linear Temporal Logic

3.1 Linear Temporal Logic definitions

We offer a mapping between the temporal sequence functions defined in subsection 1.2 (Gameplay Preferences) and linear temporal logic (LTL) operators. As we were creating this DSL, we found that the syntax of the $\langle \textit{then} \rangle$ operator felt more convenient than directly writing down LTL, but we hope the mapping helps reason about how we see our temporal operators functioning. LTL offers the following operators, using φ and ψ as the symbols (in our case, predicates). Assume the following formulas operate sequence of states S_0, S_1, \dots, S_n :

- **Next**, $X\psi$: at the next timestep, ψ will be true. If we are at timestep i , then $S_{i+1} \vdash \psi$
- **Finally**, $F\psi$: at some future timestep, ψ will be true. If we are at timestep i , then $\exists j > i : S_j \vdash \psi$
- **Globally**, $G\psi$: from this timestep on, ψ will be true. If we are at timestep i , then $\forall j : j \geq i : S_j \vdash \psi$

- **Until**, $\psi U \varphi$: ψ will be true from the current timestep until a timestep at which φ is true. If we are at timestep i , then $\exists j > i : \forall k : i \leq k < j : S_k \vdash \psi$, and $S_j \vdash \varphi$.
- **Strong release**, $\psi M \varphi$: the same as until, but demanding that both ψ and φ are true simultaneously: If we are at timestep i , then $\exists j > i : \forall k : i \leq k \leq j : S_k \vdash \psi$, and $S_j \vdash \varphi$.

Aside: there's also a **weak until**, $\psi W \varphi$, which allows for the case where the second is never true, in which case the first must hold for the rest of the sequence. Formally, if we are at timestep i , if $\exists j > i : \forall k : i \leq k < j : S_k \vdash \psi$, and $S_j \vdash \varphi$, and otherwise, $\forall k \geq i : S_k \vdash \psi$. Similarly there's **release**, which is the similar variant of strong release. We're leaving those two as an aside since we don't know we'll need them.

3.2 Satisfying a $\langle \text{then} \rangle$ operator

Formally, to satisfy a preference using a $\langle \text{then} \rangle$ operator, we're looking to find a sub-sequence of S_0, S_1, \dots, S_n that satisfies the formula we translate to. We translate a $\langle \text{then} \rangle$ operator by translating the constituent sequence-functions ($\langle \text{once} \rangle$, $\langle \text{hold} \rangle$, $\langle \text{while-hold} \rangle$)¹ to LTL. Since the translation of each individual sequence function leaves the last operand empty, we append a 'true' (\top) as the final operand, since we don't care what happens in the state after the sequence is complete.

(once ψ) := $\psi X \dots$

(hold ψ) := $\psi U \dots$

(hold-while $\psi \alpha \beta \dots \nu$) := $(\psi M \alpha) X (\psi M \beta) X \dots X (\psi M \nu) X \psi U \dots$ where the last $\psi U \dots$ allows for additional states satisfying ψ until the next modal is satisfied.

For example, a sequence such as the following, which signifies a throw attempt:

```
(then
  (once (agent_holds ?b))
  (hold (and (not (agent_holds ?b)) (in_motion ?b)))
  (once (not (in_motion ?b)))
)
```

Can be translated to LTL using $\psi := (\text{agent_holds } ?b)$, $\varphi := (\text{in_motion } ?b)$ as:

$\psi X (\neg \psi \wedge \varphi) U (\neg \varphi) X \top$

Here's another example:

```
(then
  (once (agent_holds ?b))       $\alpha$ 
  (hold-while
    (and (not (agent_holds ?b)) (in_motion ?b))   $\beta$ 
    (touch ?b ?r)   $\gamma$ 
  )
  (once (and (in ?h ?b) (not (in_motion ?b))))   $\delta$ 
)
```

If we translate each predicate to the letter appearing in blue at the end of the line, this translates to:

$\alpha X (\beta M \gamma) X \beta U \delta X \top$

¹These are the ones we've used so far in the interactive experiment dataset, even if we previously defined other ones, too.