Goals as Reward-Generating Programs Domain Specific Language

March 15, 2022

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 ⟨*constraints*⟩ section, which defines gameplay preferences, and the ⟨*scoring*⟩ section, which defines how gameplay preferences are counted to arrive at a score for the player in the game. The optional ones are the ⟨*setup*⟩ section, which defines how the environment must be prepared before gameplay can begin, and the ⟨*terminal*⟩ conditions, which specify when and how the game ends.

⟨*game*⟩ ::= (define (game ⟨*name*⟩)
(:domain ⟨*name*⟩)
(:setup ⟨*setup*⟩)
(:constraints ⟨*constraints*⟩)
(:terminal ⟨*terminal*⟩)
(:scoring ⟨*scoring*⟩)
)

⟨*name*⟩ ::= /[A-z]+(_[A-z0-9]+)*/ # a letter, optionally followed by letters, numbers, and underscores

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 ⟨*setup*⟩ element can expand to conjunctions, disjunctions, negations, or quantifications of itself, and then to the ⟨*setup-statement*⟩ rule. ⟨*setup-statement*⟩ 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 ⟨*setup-statement*⟩ tags we find ⟨*setup-predicate*⟩ elements, which can again resolve into logical conditions and quantifications of other ⟨*setup-predicate*⟩ elements, but also to function comparisons (⟨*f-comp*⟩) and predicates (⟨*predicate*⟩). 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 ⟨*predicate*⟩ element expands to a predicate acting on one or more objects or variables. We assume the list of predicate existing in a domain will be provided to any models as part of their inputs rather than hard-coded into the grammar. For a full list of the predicates we found ourselves using so far, see subsection 2.1 (Predicates).

⟨*setup*⟩ ::= (and ⟨*setup*⟩ ⟨*setup*⟩$^+$) # A setup can be expanded to a conjunction, a disjunction, a quantification, or a setup statement (see below).
| (or ⟨*setup*⟩ ⟨*setup*⟩$^+$)
| (not ⟨*setup*⟩)
| (exists (⟨*typed list(variable)*⟩) ⟨*setup*⟩)
| (forall (⟨*typed list(variable)*⟩) ⟨*setup*⟩)
| ⟨*setup-statement*⟩

⟨*setup-statement*⟩ ::= # A setup statement specifies that a predicate is either optional during gameplay or must be preserved during gameplay.
| (game-conserved ⟨*super-predicate*⟩)
| (game-optional ⟨*super-predicate*⟩)

⟨*super-predicate*⟩ ::= # 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.
| (and ⟨*super-predicate*⟩$^+$)
| (or ⟨*super-predicate*⟩$^+$)
| (not ⟨*super-predicate*⟩
| (exists (⟨*typed list(variable)*⟩) ⟨*super-predicate*⟩)
| (forall (⟨*typed list(variable)*⟩) ⟨*super-predicate*⟩)
| ⟨*f-comp*⟩
| ⟨*predicate*⟩

⟨*f-comp*⟩ ::= # A function comparison: either comparing two function evaluations, or checking that two ore more functions evaluate to the same result.
| (⟨*comp-op*⟩ ⟨*function-eval-or-number*⟩ ⟨*function-eval-or-number*⟩)
| (= ⟨*function-eval-or-number*⟩$^+$)

⟨*comp-op*⟩ ::= ⟨ | ⟨= | = | ⟩ | ⟩= # Any of the comparison operators.

⟨*function-eval-or-number*⟩ ::= ⟨*function-eval*⟩ | ⟨*number*⟩

⟨*function-eval*⟩ ::= (⟨*name*⟩ ⟨*function-term*⟩$^+$) # An evaluation of a function on any number of arguments.

⟨*function-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩ | ⟨*number*⟩ | ⟨*predicate*⟩

⟨*variable-list*⟩ ::= (⟨*variable-type-def*⟩$^+$) # One or more variables definitions, enclosed by parentheses.

⟨*variable-type-def*⟩ ::= ⟨*variable*⟩$^+$ - ⟨*type-def*⟩ # Each variable is defined by a variable (see next) and a type (see after).

⟨*variable*⟩ ::= /\?[a-z][a-z0-9]*/ # a question mark followed by a letter, optionally followed by additional letters or numbers.

⟨*type-def*⟩ ::= ⟨*name*⟩ | ⟨*either-types*⟩ # A veriable type can either be a single name, or a list of type names, as specified by the next rule:

⟨*either-types*⟩ ::= (either ⟨*name*⟩$^+$)

⟨*predicate*⟩ ::= (⟨*name*⟩ ⟨*predicate-term*⟩*)

⟨*predicate-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩

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, ⟨*constraints*⟩, is inherited from the PDDL element with the same name.

The ⟨*constraints*⟩ elements expands into one or more preference definitions, which are defined using the ⟨*pref-def*⟩ element. A ⟨*pref-def*⟩ either expands to a single preference (⟨*preference*⟩), or to a ⟨*pref-forall*⟩ element, which specifies variants of the same preference for different objects, which can be treated differently in the scoring section. A ⟨*preference*⟩ is defined by a name and a ⟨*preference-quantifier*⟩, which expands to an optional quantification (exists, forall, or neither), inside of which we find the ⟨*preference-body*⟩.

A ⟨*preference-body*⟩ expands into one of two options: The first is a set of conditions that should be true at the end of gameplay, using the ⟨*at-end*⟩ operator. Inside an ⟨*at-end*⟩ we find a ⟨*pref-predicate*⟩, which like the ⟨*setup-predicate*⟩ term, can expand to logical operations over predicates, quantifications over predicates, a function comparison, or a predicate.

The second option is specified using the ⟨*then*⟩ syntax, which defines a series of temporal conditions that should hold over a sequence of states. Under a ⟨*then*⟩ operator, we find two or more sequence functions (⟨*seq-func*⟩), 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 ⟨*then*⟩ operators expects to find a sequence of contiguous states that satisfy the different sequence functions. The operators under a ⟨*then*⟩ operator map onto linear temporal logic (LTL) operators, see **??** (**??**) for the mapping and examples.

The ⟨*once*⟩ operator specifies a predicate that must hold for a single world state. If a ⟨*once*⟩ operators appears as the first operator of a ⟨*then*⟩ definition, and a sequence of states $S_a, S_{a+1}, \cdots, S_b$ satisfy the ⟨*then*⟩ 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 ⟨*then*⟩ operator ends with a ⟨*once*⟩ term, there could be other states after the final state (S_{b+1}, S_{b+2}, etc.) that satisfy the predicate in the ⟨*once*⟩ operator, but only one is required. The ⟨*once-measure*⟩ operator is a slight variation of the ⟨*once*⟩ 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 ⟨*hold*⟩ 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 ⟨*hold*⟩ operator appears at the beginning or an end of a ⟨*then*⟩ 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 $S_a, S_{a+1}, \cdots, S_b$ that satisfy this ⟨*then*⟩ operator, the following conditions must hold true: (1) pred_a is true at state S_a, (2) pred_b is true in all states $S_{a+1}, S_{a+2}, \cdots, S_{b-2}, S_{b-1}$, 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 ⟨*hold-while*⟩, which offers a variation of the ⟨*hold*⟩ operator. A ⟨*hold-while*⟩ receives at least two predicates. The first acts the same as predicate in a ⟨*hold*⟩ 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 $S_{a+1}, S_{a+2}, \cdots, S_{b-2}, S_{b-1}$, 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.

⟨*constraints*⟩ ::= ⟨*pref-def*⟩ | (and ⟨*pref-def*⟩$^+$) # One or more preferences.

⟨*pref-def*⟩ ::= ⟨*pref-forall*⟩ | ⟨*preference*⟩ # A preference definitions expands to either a forall quantification (see below) or to a preference.

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

⟨*preference*⟩ ::= (preference ⟨*name*⟩ ⟨*preference-quantifier*⟩) # A preference is defined by a name and a quantifer that includes the preference body.

⟨*preference-quantifier*⟩ ::= # A preference can quantify exsistentially or universally over one or more variables, or none.
- (exists (⟨*variable-list*⟩) ⟨*preference-body*⟩
- (forall (⟨*variable-list*⟩) ⟨*preference-body*⟩)
- ⟨*preference-body*⟩)

⟨*preference-body*⟩ ::= ⟨*then*⟩ | ⟨*at-end*⟩

⟨*at-end*⟩ ::= (at-end ⟨*super-predicate*⟩) # Specifies a prediicate that should hold in the terminal state.

⟨*then*⟩ ::= (then ⟨*seq-func*⟩ ⟨*seq-func*⟩$^+$) # Specifies a series of conditions that should hold over a sequence of states – see below for the specific operators (⟨*seq-func*⟩s), and Section 2 for translation of these definitions to linear temporal logicl (LTL).

⟨*seq-func*⟩ ::= ⟨*once*⟩ | ⟨*once-measure*⟩ | ⟨*hold*⟩ | ⟨*hold-while*⟩ # Four of thse temporal sequence functions currently exist:

⟨*once*⟩ ::= (once ⟨*super-predicate*⟩) # The predicate specified must hold for a single world state.

⟨*once-measure*⟩ ::= (once ⟨*super-predicate*⟩ ⟨*function-eval*⟩) # The predicate specified must hold for a single world state, and record the value of the function evaluation, to be used in scoring.

⟨*hold*⟩ ::= (hold ⟨*super-predicate*⟩) # The predicate specified must hold for every state between the previous temporal operator and the next one.

⟨*hold-while*⟩ ::= (hold-while ⟨*super-predicate*⟩ ⟨*super-predicate*⟩$^+$) # 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 ⟨*super-predicate*⟩ 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 ⟨*terminal*⟩ element, which can expand to logical conditions on nested ⟨*terminal*⟩ elements, or to a terminal comparison. The terminal comparison (⟨*terminal-comp*⟩) compares two scoring expressions (⟨*scoring-expr*⟩; see subsection 1.4 (Scoring)), where in most cases, the scoring expressions are either a preference counting operation or a number literal.

⟨*terminal*⟩ ::= # The terminal condition is specified by a conjunction, disjunction, negation, or comparson (see below).
| (and ⟨*terminal*⟩$^+$)
| (or ⟨*terminal*⟩+)
| (not ⟨*terminal*⟩)
| ⟨*terminal-comp*⟩

⟨*terminal-comp*⟩ ::= (⟨*comp-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩) # A comparison operator is used to compare two scoring expressions (see next section).

⟨*comp-op*⟩ ::= ⟨ | ⟨= | = | ⟩ | ⟩=

For the full specification of the ⟨*scoring-expr*⟩ 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.

The ⟨*scoring*⟩ tag is defined by the maximization or minimization of a particular scoring expression, defined by the ⟨*scoring-expr*⟩ rule. A ⟨*scoring-expr*⟩ 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 (⟨*scoring-comp*⟩), or by preference evaluation rules. Various preference evaluation modes can expand the ⟨*preference-eval*⟩ rule, see the full list and descriptions below.

⟨*scoring*⟩ ::= (maximize ⟨*scoring-expr*⟩)
| (minimize ⟨*scoring-expr*⟩) # The scoring conditions maximize or minimize a scoring expression.

⟨*scoring-expr*⟩ ::= # 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.
| (⟨*multi-op*⟩ ⟨*scoring-expr*⟩$^+$) # Either addition or multiplication.
| (⟨*binary-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩) # Either division or subtraction.
| (- ⟨*scoring-expr*⟩)
| (total-time)
| (total-score)

| ⟨*scoring-comp*⟩
| ⟨*preference-eval*⟩

⟨*scoring-comp*⟩ ::= # A scoring comparison: either comparing two expressions, or checking that two ore more expressions are equal.
| (⟨*comp-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩)
| (= ⟨*scoring-expr*⟩$^{+}$)

⟨*preference-eval*⟩ ::= # A preference evaluation applies one of the scoring operators (see below) to a particular preference referenced by name (with optional types).
| ⟨*count-nonoverlapping*⟩
| ⟨*count-once*⟩
| ⟨*count-once-per-objects*⟩
| ⟨*count-nonoverlapping-measure*⟩
| ⟨*count-unique-positions*⟩
| ⟨*count-same-positions*⟩
| ⟨*count-maximal-nonoverlapping*⟩
| ⟨*count-maximal-overlapping*⟩
| ⟨*count-maximal-once-per-objects*⟩
| ⟨*count-maximal-once*⟩
| ⟨*count-once-per-external-objects*⟩

⟨*count-nonoverlapping*⟩ ::= (count-nonoverlapping ⟨*pref-name-and-types*⟩) # Count how many times the preference is satisfied by non-overlapping sequences of states.

⟨*count-once*⟩ ::= (count-once ⟨*pref-name-and-types*⟩) # Count whether or not this preference was satisfied at all.

⟨*count-once-per-objects*⟩ ::= (count-once-per-objects ⟨*pref-name-and-types*⟩) # Count once for each unique combination of objects quantified in the preference that satisfy it.

⟨*count-nonoverlapping-measure*⟩ ::= (count-nonoverlapping-measure ⟨*pref-name-and-types*⟩) # Can only be used in preferences including a ⟨*once-measure*⟩ modal, maps each preference satistifaction to the value of the function evaluation in the ⟨*once-measure*⟩.

⟨*count-unique-positions*⟩ ::= (count-unique-positions ⟨*pref-name-and-types*⟩) # Count how many times the preference was satisfied with quantified objects that remain stationary within each preference satisfcation, and have different positions between different satisfactions.

⟨*count-same-positions*⟩ ::= (count-same-positions ⟨*pref-name-and-types*⟩) # Count how many times the preference was satisfied with quantified objects that remain stationary within each preference satisfcation, and have (approximately) the same position between different satisfactions.

⟨*note*⟩ : # All of the count-maximal-... operators refer to counting only for preferences inside a (forall ...), and count only for the object quantified externally that has the most preference satisfactions to it. If there exist multiple preferences in a single (forall ...) block, score for the single object that satisfies the most over all such preferences.

⟨*count-maximal-nonoverlapping*⟩ ::= (count-maximal-nonoverlapping ⟨*pref-name-and-types*⟩) # For the single externally quantified object with the most satisfcations, count non-overlapping satisfactions of this preference.

⟨*count-maximal-overlapping*⟩ ::= (count-maximal-overlapping ⟨*pref-name-and-types*⟩) # For the single externally quantified object with the most satisfcations, count how many satisfactions of this preference with different objects overlap in their states.

⟨*count-maximal-once-per-objects*⟩ ::= (count-maximal-once-per-objects ⟨*pref-name-and-types*⟩) # For the single externally quantified object with the most satisfcations, count this preference for each set of quantified objects that satisfies it.

⟨*count-maximal-once*⟩ ::= (count-maximal-once ⟨*pref-name-and-types*⟩) # For the externally quantified object with the most satisfcations (across all preferences in the same (forall ...) block), count this preference at most once.

⟨*count-once-per-external-objects*⟩ ::= (count-once-per-external-objects ⟨*pref-name-and-types*⟩) # 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.

⟨*pref-name-and-types*⟩ ::= ⟨*name*⟩ ⟨*pref-object-type*⟩* # The optional ⟨*pref-object-type*⟩s are used to specify a particular instance of the preference for a given object, see the ⟨*pref-forall*⟩ syntax above.

⟨*pref-object-type*⟩ ::= : ⟨*name*⟩ # 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.

2 Non-Grammar Definitions

2.1 Predicates

The predicates are not defined as part of the grammar, but rather, we envision them is being specific to a domain and being specified to any model as part of the model inputs. Predicates can act on any number of arguments, and return a Boolean value.

The following enumerates all predicates currently found in our game dataset:

```
(= <arg1> <arg2>) [7 references] ; Are these two objects the same object?
(above <arg1> <arg2>) [5 references] ; Is the first object above the second object?
(adjacent <arg1> <arg2>) [76 references] ; Are the two objects adjacent? [will probably be
    implemented as distance below some threshold]
(adjacent_side <3 or 4 arguments>) [14 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 <2 or 3 arguments>) [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>) [311 references] ; Is the object in motion?
(is_setup_object <arg1>) [10 references] ; Is this the object of the same type referenced in
    the setup?
(object_orientation <arg1> <arg2>) [15 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>) [165 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_type <arg1> <arg2>) [3 references] ; Are these two objects of the same type?
(toggled_on <arg1>) [4 references] ; Is this object toggled on?
(touch <arg1> <arg2>) [49 references] ; Are these two objects touching?
(type <arg1> <arg2>) [9 references] ; Is the first argument, an object, an instance of the type
     specified by the second argument?
```

2.2 Functions

Functions operate similarly to predicates, but rather than returning a Boolean value, they return a numeric value or a type. Similarly to predicates, functions are not a part of the grammar, and may vary by problem domain.

The following describes all functions currently found in our game dataset:

```
(building_size ) [2 references] ; Takes in an argument of type building, and returns how many
    objects comprise the building (as an integer).
(color ) [26 references] ; Take in an argument of type object, and returns the color of the
    object (as a color type object).
(distance ) [114 references] ; Takes in two arguments of type object, and returns the distance
    between the two objects (as a floating point number).
(distance_side ) [5 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).
(type ) [2 references] ; Takes in an argument of type object, and returns the type of the
    object (as a string).
(x_position ) [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 also not defined as part of the grammar, and we envision them operating similarly to the predicates and functions, varying by domain and provided to any models as part of its inputs .

The following describes all types currently found in our game dataset:

```
game_object [33 references] ; Parent type of all objects
agent [87 references] ; The agent
building [21 references] ; Not a real game object, but rather, a way to refer to structures the
     agent builds
----------  Blocks  ----------
block [27 references] ; Parent type of all block types:
bridge_block [11 references]
cube_block [40 references]
blue_cube_block [8 references]
tan_cube_block [1 reference]
yellow_cube_block [8 references]
flat_block [5 references]
pyramid_block [14 references]
blue_pyramid_block [3 references]
red_pyramid_block [2 references]
triangle_block [3 references]
yellow_pyramid_block [2 references]
cylindrical_block [12 references]
tall_cylindrical_block [7 references]
----------  Balls  ----------
ball [40 references] ; Parent type of all ball types:
beachball [23 references]
basketball [18 references]
dodgeball [110 references]
blue_dodgeball [6 references]
red_dodgeball [4 references]
pink_dodgeball [18 references]
golfball [28 references]
green_golfball [2 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 [1 reference]
brown [1 reference]
```

```
green [5 references]
pink [14 references]
orange [3 references]
purple [4 references]
red [8 references]
tan [1 reference]
white [1 reference]
yellow [14 references]
----------  Other moveable/interactable objects  ----------
alarm_clock [8 references]
book [11 references]
blinds [2 references] ; The blinds on the windows
chair [17 references]
cellphone [6 references]
cd [6 references]
credit_card [1 reference]
curved_wooden_ramp [17 references]
desktop [6 references]
doggie_bed [26 references]
hexagonal_bin [124 references]
key_chain [5 references]
lamp [2 references]
laptop [7 references]
main_light_switch [3 references] ; The main light switch on the wall
mug [3 references]
triangular_ramp [10 references]
green_triangular_ramp [1 reference]
pen [2 references]
pencil [2 references]
pillow [12 references]
teddy_bear [14 references]
watch [2 references]
----------  Immoveable objects  ----------
bed [48 references]
corner [N/A references] ; Any of the corners of the room
south_west_corner [2 references] ; The corner of the room where the south and west walls meet
door [9 references] ; The door out of the room
desk [40 references]
desk_shelf [2 references] ; The shelves under the desk
drawer [5 references] ; Either drawer in the side table
top_drawer [6 references] ; The top of the two drawers in the nightstand near the bed.
floor [24 references]
rug [37 references]
shelf [10 references]
bottom_shelf [1 reference]
top_shelf [5 references]
side_table [4 references] ; The side table/nightstand next to the bed
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)
wall [17 references] ; Any of the walls in the room
north_wall [1 reference] ; The wall with the door to the room
south_wall [1 reference] ; The wall with the sliding doors
west_wall [2 references] ; The wall the bed is aligned to
----------  Non-object-type predicate arguments  ----------
back [3 references]
front [8 references]
left [2 references]
right [2 references]
sideways [3 references]
upright [10 references]
upside_down [2 references]
```

```
front_left_corner [1 reference] ; The front-left corner of a specific object (as determined by
    its front)
```

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 $\langle then \rangle$ 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 $S_0, S_1, \cdots, S_n$:

- **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 $\langle then \rangle$ operator

Formally, to satisfy a preference using a $\langle then \rangle$ operator, we're looking to find a sub-sequence of $S_0, S_1, \cdots, S_n$ that satisfies the formula we translate to. We translate a $\langle then \rangle$ operator by translating the constituent sequence-functions ($\langle once \rangle$, $\langle hold \rangle$, $\langle while\text{-}hold \rangle$)[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) := $\psi X \cdots$

(hold ψ) := $\psi U \cdots$

(hold-while ψ α β $\cdots \nu$) := $(\psi M \alpha) X (\psi M \beta) X \cdots X (\psi M \nu) X \psi U \cdots$ where the last $\psi U \cdots$ 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 ψ := (agent_holds ?b), φ := (in_motion ?b) as:

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

Here's another example:

```
(then
    (once (agent_holds ?b))   α
    (hold-while
        (and (not (agent_holds ?b)) (in_motion ?b))  β
        (touch ?b ?r)  γ
    )
```

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

```
    (once  (and (in ?h ?b) (not (in_motion ?b))))  δ
)
```

If we translate each predicate to the letter appearing in blue at the end of the line, this translates to:
$\alpha X(\beta M\gamma)X\beta U\delta X\top$