# Generating Human-Like Goals by Synthesizing Reward-Producing Programs

**Guy Davidson**$^{\alpha*}$, **Graham Todd**$^{\beta*}$, **Todd Gureckis**$^{\gamma}$, **Julian Togelius**$^{\beta}$, **Brenden Lake**$^{\alpha,\gamma}$
$^{\alpha}$Center for Data Science, $^{\beta}$Game Innovation Lab, $^{\gamma}$Department of Psychology
New York University
guy.davidson@nyu.edu, gdrtodd@nyu.edu

## Abstract

Humans show a remarkable capacity to generate novel goals, for learning [26] and play [8] alike, and modeling this human capacity would be a valuable step toward more generally-capable artificial agents [9]. We describe a computational model for generating novel human-like goals represented in a domain-specific language (DSL). We learn a 'human-likeness' fitness function over expressions in this DSL from a small (<100 game) human dataset collected in an online experiment. We then use a Quality-Diversity (QD) approach to generate a variety of human-like games with different characteristics and high fitness. We demonstrate that our method can generate synthetic games that are syntactically coherent under the DSL, semantically sensible with respect to environmental objects and their affordances, but distinct from human games in the training set. We discuss key components of our model and its current shortcomings, in the hope that this work helps inspire progress toward self-directed agents with human-like goals.

While both learning and playing, humans readily generate novel goals that make use of environmental affordances and their own abilities, without any top-down guidance or external motivation [21, 1]. For instance, a child exploring a novel collection of toys might choose to stack some blocks in a tower before trying to knock it over with a ball, all without parental encouragement or supervision. Among the suggested benefits of play is a contribution to cognitive development [8]—through play, we test and improve our capacity for problem-solving and reasoning in a variety of contexts. Recent work in reinforcement learning has attempted to extend notions of play, particularly self-created goals, to artificial *autotelic* agents [9]. While these agents, like humans, make use of goals in order to self-direct their learning and exploration, such approaches typically define goals as particular regions of state space [14, 5, 29] or, more recently, as linguistic descriptions generated by and evaluated with large language models [15, 12, 10, 32]. Thus, the "goals" these agents generate tend to be either unstructured or simplistic, and fail to recreate both the process and outcomes of human goal generation. In this work, we propose a novel approach for generating a specific class of human-like goals. We are motivated both by achieving a greater understanding of the human cognitive mechanism of goal generation and by empowering artificial agents to develop and hone skills through iterative task-setting. Our contribution is a model of an underexplored cognitive capacity that could facilitate the development of autotelic agents with richer and more creative self-generated goals.

We build on previous work [11] studying game creation as the capacity to generate playful cognitive goals. We represent these cognitive goals as structured programs in a domain-specific language (DSL). This representation has its root in Language of Thought (LoT) approaches, and such program-like encodings have been used in a variety of cognitive science domains [27]. We focus our attention on games that are both *structured* (defining explicit conditions for scoring and ending the game) as well as *temporally-rich* (i.e. fundamentally built from sequences of states in time, as opposed to the instantaneously-evaluable conditions found in most board or video games). Prior approaches, such as GLTL [22] or reward machines [16] allow representing temporally-extended, non-Markovian goals. We borrow from Icarte et al. [16] the notion of these goals operating as programs over sequences of states, emitting reward when appropriate, but using a domain-specific language capable of representing human games. Our work also falls broadly in the realm of automatic game design, which has mostly focused on board games [25, 4] and video games [30, 28, 17]. While reliably producing human-quality and non-trivial games remains an open challenge, evolutionary approaches have yielded at least some success when combined with expert knowledge [3]. Our approach learns a feature-based fitness function that distinguishes between "human" and "non-human" games, which acts as the objective function for a quality-diversity algorithm that produces a wide range of plausible game samples.

## 1 Representing Games as Programs

To represent human-like games, we use the DSL defined by Davidson et al. [11]. The DSL is specifically designed to represent structured, scorable games played by an embodied agent in an egocentric, 3D environment (the `AI2Thor` simulator [18]). Typical games involve manipulating objects in the environment (balls, blocks, etc.) and positioning the player-controlled agent. Each game program contains at least two sections: a set of **preferences** which specify how a game is played and **scoring** rules which describe how a player's score or reward is determined from their actions in the game. A game preference is either a set of temporal conditions on gameplay expressions (i.e. a sequence of states in which particular statements about the environment are true, which maps to linear temporal logic [23]) or a static condition which must hold at the end of a game. For instance, a game preference might encode the act of throwing a ball into a bin, or an item being placed in a drawer at the end of play. In addition to preferences and scoring, a game may optionally specify **setup** conditions that must be satisfied before play can begin and **terminal** conditions that indicate the end of the game. The full specification of our DSL and predicates are available in Appendix E.

## 2 Modeling the Human-Likeness of Games: a Learned Objective Function

While our DSL can represent a large variety of possible games, validity under the language's probabilistic context-free grammar (PCFG) is no guarantee of quality: syntactically valid programs might still be uninterpretable (e.g. refer to a nonexistent variable), contradictory (e.g. require a predicate to be simultaneously satisfied and unsatisfied) or violate physical intuitions (e.g. require a structure to be made by stacking balls rather than blocks). While these challenges might be sidestepped by directly sampling from a generative model trained on a sufficiently large and varied dataset of high-quality games, producing such a dataset comes with substantial costs. We instead draw inspiration from the capacity of children to invent novel and tractable games from a young age and without exposure to internet-scale amounts of data.

At a high level, our approach yields an explicit measure of game quality by augmenting the modest set of 98 human games translated into the DSL by Davidson et al. [11] with a substantially larger set of games corrupted by introducing random changes sampled from the PCFG. We then learn a feature-based objective function that discriminates between *positive* real games and their *negative* corruptions. Our goal is to learn a fitness function $f : \mathcal{G} \to \mathbb{R}$ mapping example games $g \in \mathcal{G}$, the space of game programs, to real-valued fitness scores, where higher is more "human-like." The following sections describe our feature extraction function $\phi : \mathcal{G} \to \mathbb{R}^F$ (where $F$ is the dimensionality of our feature space), and the form of $f$ and the loss function we use to learn it.

### 2.1 Feature Representation of Games

Our feature extraction procedure $\phi$ represents each game as a $F = 50$-dimensional vector of feature values. Features measure a variety of structural and semantic properties of games, from the size and depth of the syntax tree, to where all used variables are defined (and vice-versa). Particularly notable features include the likelihood of a program under a simple $n$-gram language model trained over the goal program syntax trees and an approximate "feasibility" measure that makes use of a dataset of 382 human *play traces* to determine the proportion of a game's predicate-argument combinations which we have seen satisfied by human players. A full description of our feature set, including the features found to be most predictive of positives and negatives, is available in Appendix B.

### 2.2 Contrastive Learning of a Fitness Function

We learn a fitness function optimized to assign higher fitness scores to *positive* human-generated games than to a set of *negatives*. Our approach takes inspiration from contrastive learning of energy-based models [7], though we attempt to maximize the fitness score assigned to positive examples, rather than minimize their energy. To learn an effective fitness function, the set of negatives must be qualitatively worse than our set of human games without being trivially distinguishable from them. Early experiments indicated that naively sampling from the PCFG with expansion probabilities fit to our collected data produced negatives that met the first criteria but not the second. Instead, we generate negatives by uniformly sampling a single node (and all of its children) from a game's syntax tree before re-generating the node and its children using the PCFG. Given the small dataset, large grammar (see Appendix E), and the lack of context (in the PCFG), resampling a single node (and its descendants) can be sufficient to alter these programs. While these alterations can sometimes be benign, in most cases the regrown expression is semantically 'out of place,' depending in part on the height of the sampled node in the syntax tree — larger regrowths, sampled closer to the root, tend to differ more. To account for the variable difficulty, we generate a large set of negatives, 1024 for each of the 98 positives, for a total of around 100,000 examples.

Our fitness function $f_\theta(g) = \theta^T \phi(g)$ is a simple linear transformation from our feature space to a real-valued output, parameterized by a feature vector $\theta \in \mathbb{R}^F$. We train the fitness function using a softmax loss, not unlike the MEE loss used to train energy-based models [19] or the InfoNCE loss [31]. Formally, for a positive example $g^+$ and a set of negative examples $\{g_k^-\}, k \in \{1, 2, \cdots, K\}$, we assign the following loss:

$$\mathcal{L}(g^+, \{g_k^-\}_1^K; \theta) = \frac{\exp(f_\theta(g^+))}{\exp(f_\theta(g^+)) + \sum_{k=1}^K \exp(f_\theta(g_k^-))} \tag{1}$$

This loss encourages the model to assign higher fitness scores to the real games than the negative examples. Simultaneously, this loss provides a diminishing incentive to push negative fitness scores down as the distance between the positives and negatives increases, intuitively assigning higher loss to negative examples with fitness closer to the positive example's fitness. See Appendix C for full details of our training and cross-validation setups.

## 3 Searching Through the Space of Games

We use MAP-Elites [24], a popular "quality-diversity" algorithm, in order to explore a wider range of human-like games than would be returned through direct optimization of our fitness function. MAP-Elites is a population-based, evolutionary algorithm that works by defining a set of *behavioral characteristics*: discrete-valued functions over genotypes (in our case, games) that form the axes of a multi-dimensional *archive* of cells. At each step, a game $g$ is selected uniformly from among the individuals in the archive and mutated to form a new game $g'$. The mutated $g'$ is evaluated both under the fitness function $f$ and each of the $n$ behavioral characteristics $b_i : \mathcal{G} \to \{0, \ldots, k_i\}$ in order to determine which cell $c = [b_1(g), \ldots, b_n(g)]$ it occupies. If the cell is unoccupied, then $g'$ enters the archive. Otherwise, it enters the archive only if its fitness is greater than the current occupant of the cell (and replaces the previous occupant). In this way, the algorithm maintains an "elite" for each possible combination of values under the behavioral characteristics.

Following prior work on using MAP-Elites for procedural content generation [6], we define a set binary behavioral characteristics that each indicate the presence of particular archetypal game components (i.e. a ball, or the use of the `adjacent` predicate) – the algorithm attempts to find a high-quality game for each possible combination of values under the behavioral characteristics. We use 10 such features for a total archive size of 1024. To mutate a game, we randomly select an operator from among the following: **regrowing** a random node and its children in its syntax tree, **inserting & deleting** the child of a node with multiple potential children, **crossing over** with the syntax tree of another randomly-selected game, **resampling the**

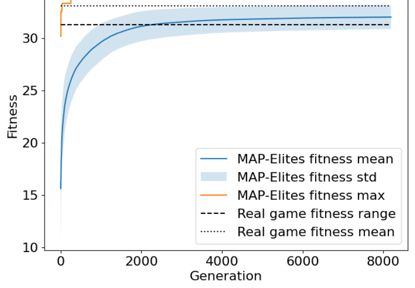

**Figure 1:** MAP-Elites quantitative results.

**variables, initial conditions, or final conditions** used by
a preference, and **resampling the optional game sections** (i.e. setup and terminal conditions). We seed the initial archive by naively sampling from the PCFG – that is, the algorithm does *not* start with corruptions of existing games as were used to train the fitness function. Further details of the algorithm are available in Appendix D.

## 4 Results

**Table 1: MAP-Elites generates simple human-like goals.** Each pair of games in a row has the same set of MAP-Elites behavioral characteristics (section 3), with the fitness scores in parentheses.

| **Real Game #42 (34.482)** | **Real Game #14 (33.154)** | **Real Game #93 (33.391)** |
|---|---|---|
| To play this game, pick up a dodgeball and throw it into a hexagonal bin. The game ends when you decide to stop, and your score is equal to the number of successful throws you made into the bin. | To play this game, stack blocks in the following order from bottom to top: a bridge block, a flat block, a tall cylindrical block, a cube block, and a pyramid block. At the end of the game, your score is 10 times the number of different blocks used in the stack. | To set up the game, place a doggie bed in the room, ensuring it's less than half the room's distance from the center. The objective of the game is to place different objects inside a building along with the doggie bed. The doggie bed should be on the floor, while the other objects should be off the floor and not touching any wall. The game ends when you've placed as many different objects as possible in this way. Your score is determined by the number of different objects you've successfully placed inside the building with the doggie bed. |
| **MAP-Elites Sample (33.986)** | **MAP-Elites Sample (32.955)** | **MAP-Elites Sample (32.123)** |
| To play this game, pick up a beachball and throw it so that it lands inside a hexagonal bin. The game ends after you've successfully thrown the beachball into the bin 22 times or after you've successfully thrown 5 different objects into the bin. Your score is the number of successful throws you made into the bin. | To play this game, place both a block and a pyramid block on a green bridge block. The game ends after this has been done at least 12 times, and your score is the number of times you've successfully placed both blocks on the green bridge block. | To set up the game, place a hexagonal bin less than 0.4 units away from the east wall. The game involves placing a blue cube block on a shelf adjacent to the west wall and positioning an object of the same color as orange closer to the rug than to the door. The game ends when you've placed 23 blue cube blocks on the shelf or positioned 20 different orange-colored objects closer to the rug. Your score is the number of times you've placed a blue cube block on the shelf. |

Figure 1 provides a quantitative summary of the results of our MAP-Elites search process. Broadly, the search is successful: the best fitness in the archive approaches that of human games within only a few "generations," the archive is filled relatively quickly, and overall sample quality continues to improve over time. The key question, however, is whether these quantitative results correlate with actual improvements in the "human-likeness" of generated games. Table 1 provides a comparison between archive exemplars and human games with the same behavioral characteristics. For clarity, we back-translate both human and generated games into neutral natural language and provide the full goal programs in Table 3. Qualitatively, we find that archive exemplars tend to resemble simple versions of the matching human games. The generated games are usually reasonable in their specified predicates and temporal modals, but tend to struggle with coherence between gameplay components (see third example), or with the number of times they task participants with repeating elements (all three examples). To some extent, this may be related to grounding: our current model is not constrained by the number of different types of objects in the environment or the difficulty of reaching proposed goals. To explore novel productions from our model, Table 2 highlights generations from cells in the MAP-Elites that have no match in our dataset (and see Table 4 for the programs in the DSL). These novel samples point towards the model's success at generating games with no immediate point of reference among the human dataset, as well as games that make use of a variety of actions, objects, and properties. However, overall coherence remains a sticking point.

**Table 2: MAP-Elites generates interesting, novel goals.** Each of the three games below has high fitness and fills a cell in the MAP-Elites archive with no corresponding human game in our dataset. Each sample's fitness score is in parentheses.

| MAP-Elites Sample (34.039) | MAP-Elites Sample (33.764) | MAP-Elites Sample (33.749) |
|---|---|---|
| To set up the game, place a hexagonal bin near the rug, ensuring that it is less than 0.3 units away. The objective of the game is to change the orientation of the bin from diagonal without touching or holding it. The game ends after the bin's orientation has been changed in this way at least 14 times, and your score is the number of times you've successfully changed the bin's orientation. | To play this game, place pillows on the bed and throw either dodgeballs or beachballs. The game ends after you've thrown a ball at least 5 times. Your score at the end of the game is the number of pillows you've placed on the bed. | To play this game, arrange objects so that two of them are the same color, one of these colored objects is adjacent to another object, and one of the colored objects is inside another object. Additionally, throw either a dodgeball or a golfball. The game ends after you've satisfied either of these conditions: arranged objects in the specified way at least twice, or thrown different balls at least three times. Your score is the number of times you've successfully arranged objects in the specified way. |

## 5 Discussion

We describe a model capable of generating human-like goals synthesized as reward-generating programs. Our quality diversity approach, powered by a constrastively-learned fitness function, is capable of producing coherent programmatic expressions in a relatively sophisticated DSL despite learning from an initial dataset consisting of fewer than 100 examples. We accomplish this by leveraging the structure of our domain (in order to produce a wide range of plausible *negatives*), expert knowledge (in the form of game desiderata and features that encode them), and secondary sources of data (human play-traces which give a sense of which predicates are feasibly satisfiable). Davidson et al. [11] noted 'creativity, compositionally, and common-sense' as key aspects of human goal generation. The relative shortcomings of our model highlight two additional important factors: complexity (not too little, not too much) and coherence (how different aspects of a game relate to each other). With that in mind, the relative success at recovering and modulating simple games indicates a potentially viable way forward in terms of capturing the process by which humans invent novel games.

## 6 Future Work

Our preliminary results point directly toward several avenues for continued research. First, we intend to perform a human evaluation of generated games, in terms of attributes such as "fun," creativity, and discriminability from human games. This could provide a clearer picture of the quantitative and qualitative differences between participant-generated games and model samples, and offer richer training signals for future models. We hope that such an analysis could also shed light on the criteria by which humans evaluate games, including the mechanisms used to perform such evaluations without actually playing the game in question. In addition, our human-inspired game generation system could be leveraged as part of an automatic task curriculum for an artificial agent. This approach falls broadly in line with research in autotelic agents and co-learning [9]. Finally, large language models (LLMs) provide an interesting alternative model class for both generating and evaluating games. While LLMs might detract from the cognitive plausibility of a model, they could be readily leveraged as evaluators in the same way as our $n$-gram feature, or perhaps fulfill a larger role. LLMs could also be incorporated into the sample generation process, such as through evolutionary techniques [20] or Bayesian inference [13].

## A Result Figures

**Table 3: Comparison between real games and corresponding MAP-Elites generations**

### Real Game #42 (fitness 34.4821)

To play this game, pick up a dodgeball and throw it into a hexagonal bin. The game ends when you decide to stop, and your score is equal to the number of successful throws you made into the bin.

```
(define (game 5ff4a242-51) (:domain few-objects-room-v1)
(:constraints
  (and
    (preference throwToBin
      (exists (?d - dodgeball ?h - hexagonal_bin)
        (then
          (once (agent_holds ?d) )
          (hold (and (not (agent_holds ?d) ) (in_motion ?d) ) )
          (once (and (not (in_motion ?d) ) (in ?h ?d) ) )
)))))
(:scoring
  (count throwToBin)
))
```

### Real Game #14 (fitness 33.1543)

To play this game, stack blocks in the following order from bottom to top: a bridge block, a flat block, a tall cylindrical block, a cube block, and a pyramid block. At the end of the game, your score is 10 times the number of different blocks used in the stack.

```
(define (game 613e4bf9-17) (:domain medium-objects-room-v1)
(:constraints
  (and
    (preference castleBuilt
      (exists (?b - bridge_block ?f - flat_block ?t -
              tall_cylindrical_block ?c - cube_block ?p -
              pyramid_block)
        (at-end
          (and
            (on ?b ?f)
            (on ?f ?t)
            (on ?t ?c)
            (on ?c ?p)
))))))
(:scoring
  (* 10 (count-once-per-objects castleBuilt) )
))
```

### Real Game #93 (fitness 33.3907)

To set up the game, place a doggie bed in the room, ensuring it's less than half the room's distance from the center. The objective of the game is to place different objects inside a building along with the doggie bed. The doggie bed should be on the floor, while the other objects should be off the floor and not touching any wall. The game ends when you've placed as many different objects as possible in this way. Your score is determined by the number of different objects you've successfully placed inside the building with the doggie bed.

```
(define (game 61087e4f-114) (:domain medium-objects-room-v1)
(:setup
    (exists (?d - doggie_bed)
      (game-conserved (< (distance room_center ?d) 0.5)
)))
(:constraints
  (and
    (preference objectInBuilding
      (exists (?o - game_object ?d - doggie_bed ?b - building)
        (at-end
          (and
            (not (same_object ?o ?d))
            (in ?b ?d)
            (in ?b ?o)
            (on floor ?d)
            (not (on floor ?o))
            (not (exists (?w - wall) (touch ?w ?o)))
))))))
(:scoring
  (count-once-per-objects objectInBuilding)
))
```

### MAP-Elites Sample (fitness 33.9861)

To play this game, pick up a beachball and throw it so that it lands inside a hexagonal bin. The game ends after you've successfully thrown the beachball into the bin 22 times or after you've successfully thrown 5 different objects into the bin. Your score is the number of successful throws you made into the bin.

```
(define (game evo-8158-92-1) (:domain few-objects-room-v1)
(:constraints
  (and
    (preference preference0
      (exists (?v0 - beachball ?v1 - hexagonal_bin)
        (then
          (once (agent_holds ?v0) )
          (hold (and (in_motion ?v0) (not (agent_holds ?v0) ) ) )
          (once (in ?v1 ?v0) )
)))))
(:terminal
  (or
    (>= (count preference0) 22 )
    (>= (count-once-per-objects preference0) 5 )
))
(:scoring
  (count preference0)
))
```

### MAP-Elites Sample (fitness 32.9549)

To play this game, place both a block and a pyramid block on a green bridge block. The game ends after this has been done at least 12 times, and your score is the number of times you've successfully placed both blocks on the green bridge block.

```
(define (game evo-8180-44-0) (:domain few-objects-room-v1)
(:constraints
  (and
    (preference preference0
      (exists (?v0 - block ?v1 - bridge_block_green ?v2 -
              pyramid_block)
        (at-end
          (and
            (on ?v1 ?v2)
            (on ?v1 ?v0)
))))))
(:terminal
  (>= (count preference0) 12 )
)
(:scoring
  (count preference0)
))
```

### MAP-Elites Sample (fitness 32.1233)

To set up the game, place a hexagonal bin less than 0.4 units away from the east wall. The game involves placing a blue cube block on a shelf adjacent to the west wall and positioning an object of the same color as orange closer to the rug than to the door. The game ends when you've placed 23 blue cube blocks on the shelf or positioned 20 different orange-colored objects closer to the rug. Your score is the number of times you've placed a blue cube block on the shelf.

```
(define (game evo-8111-143-0) (:domain few-objects-room-v1)
(:setup
    (exists (?v0 - hexagonal_bin)
      (game-conserved (< (distance east_wall ?v0) 0.4)
)))
(:constraints
  (and
    (preference preference0
      (exists (?v1 - cube_block_blue ?v0 - shelf)
        (at-end
          (and
            (adjacent west_wall ?v0)
            (on ?v0 ?v1)
    ))))
    (preference preference1
      (exists (?v2 - game_object)
        (at-end
          (and
            (same_color ?v2 orange)
            (< (distance rug ?v2) (distance door ?v2))
))))))
(:terminal
  (or
    (>= (count preference0) 23 )
    (>= (count-once-per-objects preference1) 20 )
))
(:scoring
  (count preference0)
))
```

**Table 4: Novel MAP-Elites samples** (from archive cells without human-created games).

| MAP-Elites Sample (fitness 34.0390) | MAP-Elites Sample (fitness 33.7636) | MAP-Elites Sample (fitness 33.7494) |
|---|---|---|
| To set up the game, place a hexagonal bin near the rug, ensuring that it is less than 0.3 units away. The objective of the game is to change the orientation of the bin from diagonal without touching or holding it. The game ends after the bin's orientation has been changed in this way at least 14 times, and your score is the number of times you've successfully changed the bin's orientation. | To play this game, place pillows on the bed and throw either dodgeballs or beachballs. The game ends after you've thrown a ball at least 5 times. Your score at the end of the game is the number of pillows you've placed on the bed. | To play this game, arrange objects so that two of them are the same color, one of these colored objects is adjacent to another object, and one of the colored objects is inside another object. Additionally, throw either a dodgeball or a golfball. The game ends after you've satisfied either of these conditions: arranged objects in the specified way at least twice, or thrown different balls at least three times. Your score is the number of times you've successfully arranged objects in the specified way. |
| (define (game evo−8170−346−1) (:domain<br>medium−objects−room−v1)<br>(:setup<br>(exists (?v0 − hexagonal_bin)<br>(game−conserved (< (distance rug ?v0) 0.3))<br>))<br>(:constraints<br>(and<br>(preference preference0<br>(exists (?v1 − hexagonal_bin)<br>(then<br>(once (object_orientation ?v1 diagonal) )<br>(hold (and (not (touch agent ?v1) ) (not (agent_holds ?v1)<br>) ) )<br>(once (not (object_orientation ?v1 diagonal) ) )<br>)))))<br>(:terminal<br>(>= (count preference0) 14 )<br>)<br>(:scoring<br>(count preference0)<br>)) | (define (game evo−8179−288−0) (:domain few−objects−room−v1)<br>(:constraints<br>(and<br>(preference preference0<br>(exists (?v0 − pillow ?v1 − bed)<br>(at−end<br>(and<br>(on ?v1 ?v0)<br>))))<br>(preference preference1<br>(exists (?v1 − (either dodgeball beachball))<br>(then<br>(once (agent_holds ?v1) )<br>(hold (and (not (agent_holds ?v1) ) (in_motion ?v1) ) )<br>(once (not (in_motion ?v1) ) )<br>)))))<br>(:terminal<br>(>= (count preference1) 5 )<br>)<br>(:scoring<br>(count preference0)<br>)) | (define (game evo−8174−339−0) (:domain few−objects−room−v1)<br>(:constraints<br>(and<br>(preference preference0<br>(exists (?v0 ?v1 ?v2 ?v3 − game_object)<br>(at−end<br>(and<br>(same_color ?v1 ?v2)<br>(adjacent ?v0 ?v1)<br>(in ?v3 ?v1)<br>))))<br>(preference preference1<br>(exists (?v2 − (either dodgeball golfball))<br>(then<br>(once (agent_holds ?v2) )<br>(hold (and (not (agent_holds ?v2) ) (in_motion ?v2) ) )<br>(once (not (in_motion ?v2) ) )<br>)))))<br>(:terminal<br>(or<br>(>= (count preference0) 2 )<br>(>= (count−once−per−objects preference1) 3 )<br>))<br>(:scoring<br>(count preference0)<br>)) |

## References

[1] M. M. Andersen, J. Kiverstein, M. Miller, and A. Roepstorff. Play in predictive minds: A cognitive theory of play. *Psychological Review*, 130:462–479, 6 2022.

[2] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. pages 858–867. Association for Computational Linguistics, 2007.

[3] C. Browne. Yavalath. *Evolutionary Game Design*, pages 75–85, 2011.

[4] C. Browne and F. Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[5] A. Campero, R. Raileanu, H. Küttler, J. B. Tenenbaum, T. Rocktäschel, and E. Grefenstette. Learning with amigo: Adversarially motivated intrinsic goals. 6 2021.

[6] M. Charity, M. C. Green, A. Khalifa, and J. Togelius. Mech-elites: Illuminating the mechanic space of gvg-ai. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pages 1–10, 2020.

[7] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*, I:539–546, 2005.

[8] J. Chu and L. E. Schulz. Play, curiosity, and cognition. *Annual Review of Developmental Psychology*, 2:317–343, 12 2020.

[9] C. Colas, T. Karch, O. Sigaud, and P.-Y. Oudeyer. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. 12 2020.

[10] C. Colas, L. Teodorescu, P.-Y. Oudeyer, X. Yuan, and M.-A. Côté. Augmenting autotelic agents with large language models. 5 2023.

[11] G. Davidson, T. M. Gureckis, and B. M. Lake. Creativity, compositionality, and common sense in human goal generation. In *Proceedings of the 44th Annual Meeting of the Cognitive Science Society, CogSci 2022*, jul 2022.

[12] Y. Du, O. Watkins, Z. Wang, C. Colas, T. Darrell, P. Abbeel, A. Gupta, and J. Andreas. Guiding pretraining in reinforcement learning with large language models. 7 2023.

[13] K. Ellis. Human-like few-shot learning via bayesian reasoning over natural language. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[14] C. Florensa, D. Held, X. Geng, and P. Abbeel. Automatic goal generation for reinforcement learning agents. In *International conference on machine learning*, pages 1515–1528. PMLR, 2018.

[15] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning*, pages 1769–1782. PMLR, 2023.

[16] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research 73 (2022)*, 73:173–208, 10 2022.

[17] A. Khalifa, M. C. Green, D. Perez-Liebana, and J. Togelius. General video game rule generation. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 170–177. IEEE, 2017.

[18] E. Kolve, R. Mottaghi, W. Han, E. VanderBilt, L. Weihs, A. Herrasti, M. Deitke, K. Ehsani, D. Gordon, Y. Zhu, et al. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*, 2017.

[19] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. J. Huang. *A Tutorial on Energy-Based Learning*. MIT Press, 2006.

[20] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pages 331–366. Springer, 2023.

[21] A. S. Lillard. *The Development of Play*, volume 3, pages 425–468. Wiley-Blackwell, 2015.

[22] M. L. Littman, U. Topcu, J. Fu, C. Isbell, M. Wen, and J. MacGlashan. Environment-independent task specifications via gltl. *arXiv*, 4 2017.

[23] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer New York, 1992.

[24] J.-B. Mouret and J. Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[25] B. Pell. Metagame in symmetric chess-like games. 1992.

[26] A. Ram and D. Leake. *Goal-Driven Learning*. MIT Press, 1995.

[27] J. S. Rule, J. B. Tenenbaum, and S. T. Piantadosi. The Child as Hacker. *Trends in Cognitive Sciences*, 24(11):900–915, nov 2020.

[28] A. M. Smith, M. J. Nelson, and M. Mateas. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 91–98. IEEE, 2010.

[29] O. E. L. Team, A. Stooke, A. Mahajan, C. Barros, C. Deck, J. Bauer, J. Sygnowski, M. Trebacz, M. Jaderberg, M. Mathieu, et al. Open-ended learning leads to generally capable agents. *arXiv preprint arXiv:2107.12808*, 2021.

[30] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 111–118. IEEE, 2008.

[31] van den Oord Aaron, Y. Li, and O. Vinyals. Representation learning with contrastive predictive coding. 7 2018.

[32] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

## B Full Feature Set

To simplify training fitness models, we ensure that all feature values are on the unit interval, using the following feature types:

- A binary value (marked with `[b]`)
- A proportion between zero and one (`[p]`)
- A real value discretized to two or more levels and treated as an indicator variable (`[d]`, with the levels listed at the end of the description)
- A float value normalized to the unit interval over the full dataset of positive and negative games (`[f]`)

For our n-gram features, we extract n-gram tokens from an in-order traversal of the syntax tree. We use 5-gram models with stupid backoff [2] with a discount factor of 0.4, and report the mean log score as the feature value, both jointly over the entire game program and separately over the different sections (setup, preferences, terminal conditions, and scoring).

For our predicate play trace features, we use a simplified version of the predicate satisfaction computation aspect of our reward machine (DSL program interpreter). We record, for every human play trace we have, and each predicate listed below, for every object assignment that satisfies it in that trace, all indices of states at which the predicate is satisfied. Recording specific states allows to us compute conjunctions, disjunctions, and negations in addition to individual predicate satisfactions. We limit ourselves to a subset of our predicates, which covers over 95% of predicate references in our dataset: `above`, `adjacent`, `agent_crouches`, `agent_holds`, `broken`, `game_start`, `game_over`,`in`, `in_motion`, `object_orientation`, `on`, `open`, `toggled_on`, and `touch`. Any predicate that is not implemented is assumed to be feasible to have been satisfied.

Our full feature set is:

- `ast_ngram_full_n_5_score [f]`: What is the mean 5-gram model score under an n-gram model trained on the real games?
- `ast_ngram_setup_n_5_score [f]`: What is the mean 5-gram model score of the setup section under an n-gram model trained on the real game setup sections?
- `ast_ngram_constraints_n_5_score [f]`: What is the mean 5-gram model score of the gameplay preferences section under an n-gram model trained on the real game preferences sections?
- `ast_ngram_terminal_n_5_score [f]`: What is the mean 5-gram model score of the terminal conditions section under an n-gram model trained on the real game terminal sections?
- `ast_ngram_scoring_n_5_score [f]`: What is the mean 5-gram model score of the scoring section under an n-gram model trained on the real game scoring sections?
- `predicate_found_in_data_prop [p]`: What proportion of predicates are satisfied at least once in our human play trace data?
- `predicate_found_in_data_small_logicals_prop [p]`: What proportion of logical expressions over predicates (with four or fewer children, limited for computational reasons) are satisfied at least once in our human play trace data?
- `section_doesnt_exist_setup [b]`: Does a game not have an (optional) setup section? (to allow counteracting feature values for the setup for games that do not have a setup component)
- `section_doesnt_exist_terminal [b]`: Does a game not have an (optional) terminal conditions section? (to allow counteracting feature values for the terminal conditions for games that do not have a terminal conditions component)
- `variables_used_all [b]`: Are all variables defined used at least once?
- `variables_used_prop [p]`: What proportion of variables defined are used at least once?
- `preferences_used_all [b]`: Are all preferences defined referenced at least once in terminal or scoring expressions?
- `preferences_used_prop [p]`: What proportion of preferences defined are referenced at least once in terminal or scoring expressions?
- `num_preferences_defined [d]`: How many preferences are defined? (1-7)
- `setup_objects_used [p]`: What proportion of objects referenced in the setup are also referenced in the gameplay preferences?

- `any_setup_objects_used [b]`: Are any objects referenced in the setup also referenced in the gameplay preferences?
- `repeated_variables_found [b]`: Are there any cases where the same variable is used twice in the same predicate?
- `repeated_variable_type_in_either [b]`: Are there any cases where the same variable types is used twice in an `either` quantification?
- `redundant_expression_found [b]`: Are there any cases where a logical expression over predicates is redundant (can be trivially simplified)?
- `redundant_scoring_terminal_expression_found [b]`: Are there any cases where a scoring terminal expression is redundant (can be trivially simplified)?
- `unnecessary_expression_found [b]`: Are there any cases where a logical expression over predicates is unnecessary (contradicts itself, or is trivially true)?
- `adjacent_same_modal_found [b]`: Are there any cases where the same modal is used twice in a row?
- `identical_consecutive_seq_func_predicates_found [b]`: Are there any cases where the same exact predicates (and their arguments) are applied in consecutive modals (making them redundant)?
- `disjoint_preferences_found [b]`: Are there any preferences which quantify over disjoint sets of objects?
- `disjoint_preferences_prop [p]`: What proportion of preferences quantify over disjoint sets of objects?
- `disjoint_preferences_scoring_terminal_types [p]`: Do the preferences referenced in the scoring and terminal section quantify over disjoint sets of object types?
- `disjoint_preferences_scoring_terminal_predicates [p]`: Do the preferences referenced in the scoring and terminal section use disjoint sets of predicates?
- `disjoint_preferences_same_predicates_only [b]`: Do any preferences make use solely of the `same_color`, `same_object`, and `same_type` predicates?
- `disjoint_seq_funcs_found [b]`: Are there any cases where modals in a preference refer to disjoint sets of variables or objects?
- `disjoint_modal_predicates_found [b]`: Are there any cases where modals in a preference refer to disjoint sets of predicates?
- `disjoint_modal_predicates_prop [p]`: What proportion of modals in a preference refer to disjoint sets of predicates?
- `predicate_without_variables_or_agent [b]`: Are there any predicates that do not reference any variables or the agent?
- `two_number_operation_found [b]`: Are there any cases where an arithmetic operation is applied to two numbers? (e.g. `(+ 5 5)` instead of simplifying it)
- `section_without_pref_or_total_count_terminal [b]`: Does the terminal section in this game fail to reference any preferences, or the `(total-time)` or `(total-score)` tokens?
- `section_without_pref_or_total_count_scoring [b]`: Does the scoring section in this game fail to reference any preferences, or the `(total-time)` or `(total-score)` tokens?
- `pref_forall_used_correct [b]`: For the `forall` over preferences syntax, if it is used, is it used correctly in this game?
- `pref_forall_used_incorrect [b]`: For the `forall` over preferences syntax, if it is used, is it used incorrectly in this game? (to allow learning differential values between correct and incorrect usage)
- `pref_forall_external_forall_used_correct [b]`: If the `external-forall-maximize` or `external-forall-minimize` syntax is used, is it used correctly in this game?
- `pref_forall_external_forall_used_incorrect [b]`: If the `external-forall-maximize` or `external-forall-minimize` syntax is used, is it used incorrectly in this game?
- `pref_forall_external_forall_used_correct [b]`: If the `count-once-per-external-objects` count operator is used, is it used correctly in this game?
- `pref_forall_external_forall_used_incorrect [b]`: If the `count-once-per-external-objects` count operator is used, is it used incorrectly in this game?

- `pref_forall_pref_forall_correct_arity_correct [b]`: If optional object names and types are provided to a count operation, are they provided with an arity consistent with the `forall` variable quantifications?
- `pref_forall_pref_forall_correct_arity_incorrect [b]`: If optional object names and types are provided to a count operation, are they provided with an arity inconsistent with the `forall` variable quantifications?
- `pref_forall_pref_forall_correct_types_correct [b]`: If optional object names and types are provided to a count operation, are they provided with types consistent with the `forall` variable quantifications?
- `pref_forall_pref_forall_correct_types_incorrect [b]`: If optional object names and types are provided to a count operation, are they provided with types inconsistent with the `forall` variable quantifications?

### B.1 Features Most Predictive of Real or Regrown Games

The following features (in order) had the largest weight, indicating they were most predictive of positive (real, human-generated) examples in our dataset:

1. `ast_ngram_full_n_5_score`
2. `ast_ngram_constraints_n_5_score`
3. `predicate_found_in_data_prop`
4. `section_doesnt_exist_setup`
5. `ast_ngram_setup_n_5_score`
6. `variables_used_all`
7. `preferences_used_all`
8. `ast_ngram_scoring_n_5_score`
9. `setup_objects_used`
10. `section_doesnt_exist_terminal`

The following features (in order) had the smallest weights, indicating they were most predictive of negative (regrown) examples in our dataset:

1. `pref_forall_pref_forall_correct_types_incorrect`
2. `pref_forall_used_incorrect`
3. `disjoint_seq_funcs_found`
4. `repeated_variables_found`
5. `redundant_expression_found`
6. `pref_forall_pref_forall_correct_arity_incorrect`
7. `two_number_operation_found`
8. `predicate_without_variables_or_agent`
9. `disjoint_modal_predicates_found`
10. `pref_forall_used_correct`

## C Objective function algorithm descriptions

Algorithm 1 below outlines how we train our fitness model. The number $N$ of of positive examples is fixed (98 in our full dataset), and fewer during cross-validation. We generate $M = 1024$ negatives for each of the positive examples, and the number of features $F$ is fixed as well. We perform cross-validation to select hyperparameter values $B \in \{1, 2, 4\}$, and $K \in \{256, 512, 1025\}$, selecting the set that minimizes the cross-validated loss. We optimize the model with SGD, with a learning rate $\eta \in \{1e-3, 4e-3\}$ also selected via cross-validation. We use weight decay with $\lambda = 0.003$ to regularize the model. We train the model for up to 25000 epochs, or until the model plateaus for $P = 500$ epochs. After cross-validation, we train our final objective function on the entire dataset. The final model we report uses $B = 1$ positive games per batch, $K = 1024$ negatives samples from our entire dataset for that positive, a learning rate $\eta = 4e-3$, and $F = 50$ features.

**Algorithm 1** Fitness model training loop

```
Input: Real games $\mathcal{D}^+ \in \mathbb{R}^{N \times \times 1 F}$, regrown games $\mathcal{D}^- \in \mathbb{R}^{N \times M \times F}$
Input: Fitness model $f_\theta : \mathbb{R}^F \to \mathbb{R}$, optimizer
  $N$ positive examples, $M$ negatives generated per positive, $B$ batch size, $F$ features, $K$ negatives
  sampled per positive in each epoch, $P$ plateau epochs
Output: Converged fitness model $W_\theta$
  best model ← None
  best loss ← $\infty$
  last improvement epoch ← $-1$
  for epoch $i$ do
      ▷ Assign negatives randomly to each positive
      Shuffle the first two dimensions of $\mathcal{D}^-$
      ▷ Reorder the positives in each epoch
      Shuffle the first dimension of $\mathcal{D}^+$
      for each batch do
          $X^+$ ← the next $B$ positives                          ▷ $X^+$: $B \times 1 \times F$
          $X^-$ ← $K$ sampled negatives for each positive         ▷ $X^-$: $B \times K \times F$
          $X$ ← concat($X^+, X^-$)                                ▷ $X$: $B \times (1 + K) \times F$
          $Y$ ← $f_\theta(X)$                                     ▷ $Y$: $B \times (1 + K)$
          $L$ ← softmax loss($Y$)                                 ▷ $L$: scalar
          Take backward step on loss and optimizer step
      end for
      epoch validation losses ← []
      for each batch in validation do
          <the above procedure without the optimizer steps>
          <append each batch's loss to epoch validation losses>
      end for
      epoch loss ← mean(epoch validation losses)
      if epoch loss < best loss then
          best model ← copy of $f_\theta(X)$
          best loss ← epoch loss
          last improvement epoch ← i
      else if $i$ − last improvement epoch > $P$ then
          break
      end if
  end for
  return best model
```

## D MAP-Elites Algorithm Details

The set of behavioral characteristics used by MAP-Elites are listed below. We select behavioral characteristics which roughly correspond to the common archetypes observed in human games or that group semantically-similar predicates.

- The game uses the `agent_holds` or `in_motion` predicate
- The game uses the `in` or `on` predicate
- The game uses the `adjacent` or `touch` predicate
- The game uses an object in the `balls` category
- The game uses an object in the `receptacles` category
- The game uses an object in the `blocks` or `buildings` categories
- The game uses an object in the `furniture` (e.g. the bed or desk) or `room_features` categories (e.g. the door or mirror)
- The game uses an object in the `small_items` (e.g. the cellphone or keys) or `large_items` (e.g. the chair or laptop)
- The game uses an object in the general `game_object` category

- The game contains the optional `setup` section

In addition, we add one more "pseudo behavioral characteristic" that explicitly captures a few general coherence properties of games – specifically features that we expect either *all* plausibly human-generated games to either exhibit or *none* of them to exhibit. While these features are also captured by learned fitness function, we use this behavioral characteristic as a sort of first-stage filter: if a game fails to meet these criteria, then it cannot reasonably be said to be "human-quality," regardless of its fitness evaluation. For all reported games, we ensure that each of the criteria are satisfied. The criteria included in this behavioral characteristic include whether all all variables are defined / used in preferences, whether all preferences are used in either terminal or scoring conditions, and whether the game avoids a set of grammatical but obviously nonsensical or redundant expressions. There are a total of 21 features used in this behavioral characteristic.

We begin the MAP-Elites algorithm by generating 1024 random games from the PCFG. We then sort each of the games in descending order of fitness and add them to the archive until either **(a)** every possible value of each behavioral characteristic is represented by at least one game (note that this is not the same as every possible *combination* of behavioral characteristic values being represented), or **(b)** at least 128 cells of the archive are occupied.

We run MAP-Elites for 8192 "generations," where each generation consists of 750 potential updates in which we randomly select a parent game, sample a mutation operator to apply, and potentially add the resulting mutated game to the archive.

## E Full Domain Specific Language Description

### E.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 ⟨*ID*⟩)
    (:domain ⟨*ID*⟩)
    (:setup ⟨*setup*⟩)
    (:constraints ⟨*constraints*⟩)
    (:terminal ⟨*terminal*⟩)
    (:scoring ⟨*scoring*⟩)
    )

⟨*id*⟩ ::= /[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.

#### E.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 ⟨*super-predicate*⟩ elements, which are logical operations and quantifications over other ⟨*super-predicate*⟩ elements, function comparisons (⟨*function-comparison*⟩, which like predicates also resolve to a truth value), 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. For a full list of the predicates we found ourselves using so far, see subsubsection E.2.1.

⟨*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*⟩

⟨*function-comparison*⟩ ::= # 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*⟩ | ⟨*comparison-arg-number*⟩

⟨*comparison-arg-number*⟩ ::= ⟨*number*⟩

⟨*number*⟩ ::= /-?\d*\.?\d+/ # A number, either an integer or a float.

⟨*function-eval*⟩ ::= # See valid expansions in a separate section below

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

⟨*variable-def*⟩ ::= ⟨*variable-type-def*⟩ | ⟨*color-variable-type-def*⟩ | ⟨*orientation-variable-type-def*⟩ | ⟨*side-variable-type-def*⟩ # Colors, sides, and orientations are special types as they are not interchangable with objects.

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

⟨*color-variable-type-def*⟩ ::= ⟨*color-variable*⟩$^+$ - ⟨*color-type-def*⟩ # A color variable is defined by a variable (see below) and a color type.

⟨*orientation-variable-type-def*⟩ ::= ⟨*orientation-variable*⟩$^+$ - ⟨*orientation-type-def*⟩ # An orientation variable is defined by a variable (see below) and an orientation type.

⟨*side-variable-type-def*⟩ ::= ⟨*side-variable*⟩$^+$ - ⟨*side-type-def*⟩ # A side variable is defined by a variable (see below) and a side type.

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

⟨*color-variable*⟩ ::= /\?x[0-9]*/ # a question mark followed by an x and an optional number.

⟨*orientation-variable*⟩ ::= /\?y[0-9]*/ # a question mark followed by an y and an optional number.

⟨*side-variable*⟩ ::= /\?z[0-9]*/ # a question mark followed by an z and an optional number.

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

⟨*color-type-def*⟩ ::= ⟨*color-type*⟩ | ⟨*either-color-types*⟩ # A color variable type can either be a single color name, or a list of color names, as specified below

⟨*orientation-type-def*⟩ ::= ⟨*orientation-type*⟩ | ⟨*either-orientation-types*⟩ # An orientation variable type can either be a single orientation name, or a list of orientation names, as specified below

⟨*side-type-def*⟩ ::= ⟨*side-type*⟩ | ⟨*either-side-types*⟩ # A side variable type can either be a single side name, or a list of side names, as specified below

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

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

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

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

⟨*object-type*⟩ ::= ⟨*name*⟩

⟨*name*⟩ ::= /[A-Za-z][A-za-z0-9_]+/ # a letter, followed by one or more letters, digits, or underscores

⟨*color-type*⟩ ::= 'color'

⟨*color*⟩ ::= 'blue' | 'brown' | 'gray' | 'green' | 'orange' | 'pink' | 'purple' | 'red' | 'tan' | 'white' | 'yellow'

⟨*orientation-type*⟩ ::= 'orientation'

⟨*orientation*⟩ ::= 'diagonal' | 'sideways' | 'upright' | 'upside_down'

⟨*side-type*⟩ ::= 'side'

⟨*side*⟩ ::= 'back' | 'front' | 'left' | 'right'

⟨*predicate*⟩ ::= # See valid expansions in a separate section below

⟨*predicate-or-function-term*⟩ ::= ⟨*object-name*⟩ | ⟨*variable*⟩ # A predicate or function term can either be an object name (from a small list allowed to be directly referred to) or a variable.

⟨*predicate-or-function-color-term*⟩ ::= ⟨*color*⟩ | ⟨*color-variable*⟩

$\langle$*predicate-or-function-orientation-term*$\rangle$ ::= $\langle$*orientation*$\rangle$ | $\langle$*orientation-variable*$\rangle$

$\langle$*predicate-or-function-side-term*$\rangle$ ::= $\langle$*side*$\rangle$ | $\langle$*side-variable*$\rangle$

$\langle$*predicate-or-function-type-term*$\rangle$ ::= $\langle$*object-type*$\rangle$ | $\langle$*variable*$\rangle$

$\langle$*object-name*$\rangle$ ::= 'agent' | 'bed' | 'desk' | 'door' | 'floor' | 'main_light_switch' | 'mirror' | 'room_center' | 'rug' | 'side_table' | 'bottom_drawer' | 'bottom_shelf' | 'east_sliding_door' | 'east_wall' | 'north_wall' | 'south_wall' | 'top_drawer' | 'top_shelf' | 'west_sliding_door' | 'west_wall'

### E.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 expand into one or more preference definitions, which are defined using the $\langle$*pref-def*$\rangle$ element. A $\langle$*pref-def*$\rangle$ either expands to a single preference ($\langle$*preference*$\rangle$), or to a $\langle$*pref-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 $\langle$*preference-quantifier*$\rangle$, which expands to an optional quantification (exists, forall, or neither), inside of which we find the $\langle$*preference-body*$\rangle$.

A $\langle$*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 $\langle$*at-end*$\rangle$ operator. Inside an $\langle$*at-end*$\rangle$ we find a $\langle$*super-predicate*$\rangle$, which like in the setup section, expands to logical operations or quantifications over other $\langle$*super-predicate*$\rangle$ elements, function comparisons, or predicates.

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

The $\langle$*once*$\rangle$ operator specifies a predicate that must hold for a single world state. If a $\langle$*once*$\rangle$ operators appears as the first operator of a $\langle$*then*$\rangle$ definition, and a sequence of states $S_a, S_{a+1}, \cdots, S_b$ satisfy the $\langle$*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 $\langle$*then*$\rangle$ operator ends with a $\langle$*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 $\langle$*once*$\rangle$ operator, but only one is required. The $\langle$*once-measure*$\rangle$ operator is a slight variation of the $\langle$*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 subsubsection E.1.4.

A second type of operator that exists is the $\langle$*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 $\langle$*hold*$\rangle$ operator appears at the beginning or an end of a $\langle$*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 $S_a, S_{a+1}, \cdots, S_b$ that satisfy this $\langle$*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 $S_{a+1}, S_{a+2}, \cdots, S_{b-2}, S_{b-1}$, 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 the predicate in a ⟨*hold*⟩ operator. The second (and third, and any subsequent ones), must hold true for at least one 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 pred_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 subsubsection E.1.1 above.

#### E.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*⟩) expands to one of three different types of copmarisons: ⟨*terminal-time-comp*⟩, a comparison between the total time spent in the game ((total−time)) and a time number token, ⟨*terminal-score-comp*⟩, a comparison between the total score ((total−score)) and a score number token, or ⟨*terminal-pref-count-comp*⟩, a comparison between a scoring expression (⟨*scoring-expr*⟩, see below) and a preference count number token. In most cases, the scoring expression is a preference counting operation.

⟨*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*⟩ ::= # We support three ttypes of terminal comparisons:
 | ⟨*terminal-time-comp*⟩
 | ⟨*terminal-score-comp*⟩
 | ⟨*terminal-pref-count-comp*⟩

⟨*terminal-time-comp*⟩ ::= (⟨*comp-op*⟩ (total-time) ⟨*time-number*⟩) # The total time of the game must satisfy the comparison.

⟨*terminal-score-comp*⟩ ::= (⟨*comp-op*⟩ (total-score) ⟨*score-number*⟩) # The total score of the game must satisfy the comparison.

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

⟨*time-number*⟩ ::= ⟨*number*⟩ # Separate type so the we can learn a separate distribution over times than, say, scores.

⟨*score-number*⟩ ::= ⟨*number*⟩

⟨*preference-count-number*⟩ ::= ⟨*number*⟩

⟨*comp-op*⟩ ::= ⟨ | ⟨= | = | ⟩ | ⟩=

For the full specification of the ⟨*scoring-expr*⟩ element, see subsubsection E.1.4 below.

### E.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 ⟨*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*⟩ ::= ⟨*scoring-expr*⟩ # The scoring conditions maximize 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.
 | ⟨*scoring-external-maximize*⟩
 | ⟨*scoring-external-minimize*⟩
 | (⟨*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-number-value*⟩

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

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

⟨*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*⟩
| ⟨*count-overlapping*⟩
| ⟨*count-once*⟩
| ⟨*count-once-per-objects*⟩
| ⟨*count-measure*⟩
| ⟨*count-unique-positions*⟩
| ⟨*count-same-positions*⟩
| ⟨*count-once-per-external-objects*⟩

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

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

⟨*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*⟩ ::= : ⟨*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.

⟨*scoring-number-value*⟩ ::= ⟨*number*⟩

## E.2 Non-Grammar Definitions

### E.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?*

### E.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).*

### E.2.3 Types

The types are currently not defined as part of the grammar, other than the small list of $\langle$*object-name*$\rangle$ tokens that can be directly referred to, and are marked with an asterisk below. The following enumerates all expansions of the various $\langle$*type*$\rangle$ 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*
−−−−−−−−−− (∗ \textbf{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]
−−−−−−−−−− (∗ \textbf{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]
−−−−−−−−−− (∗ \textbf{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]
−−−−−−−−−− (∗ \textbf{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*
−−−−−−−−−− (∗ \textbf{Large moveable/interactable objects} ∗) −−−−−−−−−−
book [11 references]
chair [18 references]
laptop [7 references]
pillow [14 references]
teddy_bear [14 references]
−−−−−−−−−− (∗ \textbf{Orientations} ∗) −−−−−−−−−−
diagonal [1 reference]
sideways [2 references]
upright [10 references]
upside_down [1 reference]
−−−−−−−−−− (∗ \textbf{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]
−−−−−−−−−− (∗ \textbf{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.*
−−−−−−−−−− (∗ \textbf{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*
−−−−−−−−−− (∗ \textbf{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]
---------- (* \textbf{Sides} *) ----------
back [3 references]
front [9 references]
left [3 references]
right [2 references]
```

## E.3 Modal Definitions in Linear Temporal Logic

### E.3.1 Linear Temporal Logic definitions

We offer a mapping between the temporal sequence functions defined in subsubsection E.1.2 and linear temporal logic (LTL) operators. As we were creating this DSL, we found that the syntax of the ⟨*then*⟩ 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 $\varphi$ and $\psi$ 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, $\psi$ will be true. If we are at timestep $i$, then $S_{i+1} \vdash \psi$
- **Finally**, $F\psi$: at some future timestep, $\psi$ will be true. If we are at timestep $i$, then $\exists j > i : S_j \vdash \psi$
- **Globally**, $G\psi$: from this timestep on, $\psi$ will be true. If we are at timestep $i$, then $\forall j : j \geq i : S_j \vdash \psi$
- **Until**, $\psi U \varphi$: $\psi$ will be true from the current timestep until a timestep at which $\varphi$ 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 $\psi$ and $\varphi$ 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.

### E.3.2 Satisfying a ⟨*then*⟩ operator

Formally, to satisfy a preference using a ⟨*then*⟩ 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 ⟨*then*⟩ operator by translating the constituent sequence-functions (⟨*once*⟩, ⟨*hold*⟩, ⟨*while-hold*⟩)[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 $\psi$) := $\psi X \cdots$

(hold $\psi$) := $\psi U \cdots$

(hold-while $\psi\ \alpha\ \beta \cdots \nu$) := $(\psi M \alpha) X (\psi M \beta) X \cdots X (\psi M \nu) X \psi U \cdots$ where the last $\psi U \cdots$ allows for additional states satisfying $\psi$ 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 $\psi$ := (agent_holds ?b), $\varphi$ := (in_motion ?b) as:

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

Here's another example:

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

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

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

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