Goals as Reward-Producing Programs

Guy Davidson[α*†], **Graham Todd**[β*], **Julian Togelius**[β], **Todd M. Gureckis**[γ], **Brenden M. Lake**[α, γ]
[α]Center for Data Science, [β]Game Innovation Lab, [γ]Department of Psychology
New York University
*: Co-first authorship
†: Corresponding author: guy.davidson@nyu.edu

Abstract

People are remarkably capable of generating their own goals, beginning with child's play and continuing into adulthood. Despite considerable empirical and computational work on goals and goal-oriented behavior, models are still far from capturing the richness of everyday human goals. Here, we bridge this gap by collecting a dataset of human-generated playful goals, modeling them as reward-producing programs, and generating novel human-like goals through program synthesis. Reward-producing programs capture the rich semantics of goals through symbolic operations that compose, add temporal constraints, and allow for program execution on behavioral traces to evaluate progress. To build a generative model of goals, we learn a fitness function over the infinite set of possible goal programs and sample novel goals with a quality-diversity algorithm. Human evaluators found that model-generated goals, when sampled from partitions of program space occupied by human examples, were indistinguishable from human-created games. We also discovered that our model's internal fitness scores predict games that are evaluated as more fun to play and more human-like.

Understanding how humans create, represent, and reason about goals is crucial to understanding human behavior. Goals are pervasive throughout psychology [25, 6, 27], having been studied from perspectives such as motivation [40, 26, 11], personality and social psychology [28, 67], and learning and decision-making [61, 60]. But what is a goal? Elliot & Fryer offer the workable, albeit simplified definition: a representation of a future object to be approached or avoided (see also [27, 60]). Reinforcement learning offers another formulation, operationalizing goals as maximizing cumulative reward over a series of steps [77]. Typical goals in reinforcement learning tasks include reaching a target location, winning in a video or board game [59], or placing an object in a specified position (e.g., Figure 1a), such that success can be characterized by reaching a target state.

In contrast, people routinely create novel, idiosyncratic goals with richness beyond these common modeling settings. Chu et al. [19] report the example of Gareth Wild, who set an unusual goal for himself to park in every spot in a particular grocery store's parking lot (Figure 1b). Children routinely devise fun and compelling goals without external guidance, such as creating a "truck carrier truck" (Figure 1c) or stacking as many blocks as possible in a single tower (Figure 1d). Beyond being fun, these playful goals play a crucial role in learning to structure and solve arbitrary problems [18, 52, 3]. Indeed, it has been argued that autonomously setting and achieving goals is a core component of human intelligence [19, 64].

We propose a framework for modeling human goal generation as synthesizing reward-producing programs (Figure 1, bottom row). There are several advantages to representing goals as symbolic programs, that map an agent's behavior to a reward score indicating the degree of success. First, a structured language facilitates the compositional reuse of motifs across disparate goals. Such reuse makes capturing the wide range of human creativity in goal creation substantially more tractable: In Figure 1e, we illustrate a simple ball-throwing game (in black) and four distinct variants (in red, blue, pink, and brown) composed in part from shared components: balls being thrown (highlighted in yellow), the thrown ball hitting something (orange), and the thrown ball landing somewhere (green). Second, our choice of representation makes goal semantics explicit. The particular grammatical elements of our representation each fulfill particular roles, such as *predicates* (i.e., specific and evaluable relations between objects, colored orange in the programs in Figure 1) and *temporal modals* (i.e., relationships in time between goal components, such as 'until' and 'then' in Figure 1). Finally, goals-as-programs are executable; that is, they can be computationally interpreted to detect when a goal is entirely or partially achieved (Figure 1e, each program would be interpreted and provide a score only when the matching throw trajectory is completed).

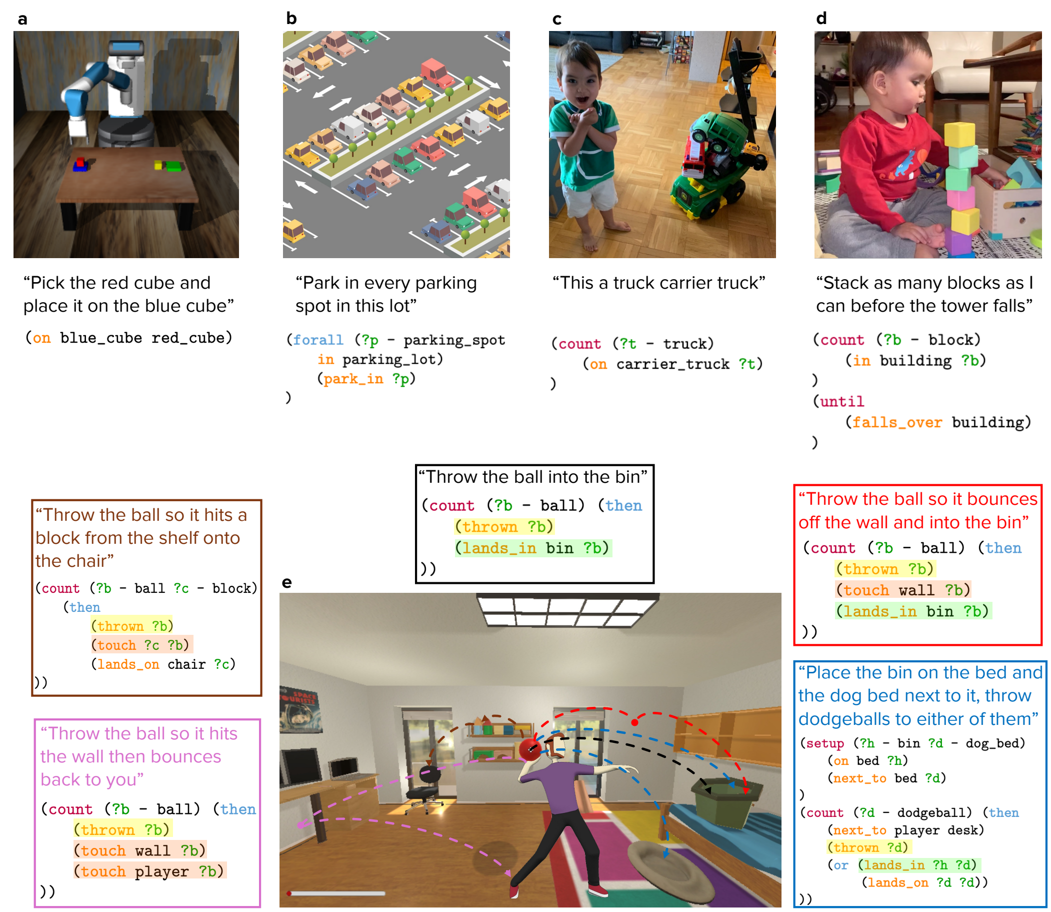


Figure 1: Goals as Reward-Producing Programs. Panels **a-d** show different goals, presented in natural language and mapped to pseudo-code in a program-like representation. Panel **e** shows a set of varied yet related goals in our experiment environment, of which the blue and pink were created by participants in our experiment. Each goal is represented by a throw trajectory (dashed line in the illustration) matching a description of the goal (whose text is the same color as the line). We highlight shared compositional components between programs in yellow, orange, and green. Our program representations are reward-producing, that is, run on sequences of agent interactions with an environment (state-action pairs) and emit a score with respect to the specified goal. Our pseudo-code and domain-specific language both use a LISP-like syntax, where function calls have the function name as the first token inside the parentheses. Participants in our experiment created some of these goals; see Figure SI-1 for representations of the blue and pink programs in our domain-specific language.

In this article, we demonstrate that programs can capture real human-created goals in a specific domain and build a model capable of generating new programs representing human-like goals. We devised a rich experimental environment for goal generation and asked human participants to generate playful goals (games), which we translated into programs in a domain-specific language. We also developed a Goal Program Generator (GPG) model to generate new goals in this representation, learning a fitness metric over programs to capture human likeness and sampling diverse goal programs to maximize fitness. We found that the model succeeds in generating novel games distinct from examples in the training dataset. Human raters evaluated several characteristics of model-generated games, including how human-like they were. Model games from sections of program space closer to participant-created games were judged indistinguishably from the real games, but model samples further away were not rated as highly on average. Analyses revealed that our learned fitness function predicts several human judgment questions, including how human-like games are rated. These results demonstrate that our goal representations and model capture important aspects of how people creatively construct new goals, generating plausible, diverse goals and predicting understandability and fun ratings.

Behavioral results

Although goals play a crucial role in psychological theory, there are few, if any, empirical paradigms for eliciting wide-ranging goals from study participants. We created an experimental setting that aims to capture the rich, playful, and creative nature of how children (and adults) create everyday goals. We used AI2-THOR [46] (an embodied, 3D environment simulation) to set up a room resembling a child's bedroom, filled with toys and other common objects (Figure 2a, and see Figure ED-1 for a larger version). In our task, we asked participants to propose a single-player game to be played in the room. This design allowed participants to imagine and propose a wide range of playful goals, with the aim of game generation helping to make the resulting goals more concrete. We collected a dataset of 98 games, described by participants in natural language. In addition, we recorded full state-action traces of each participant's interactions with the environment, which we leveraged in later experiments (see Dataset collection methods for additional details).

We then manually translated each game from natural language to programs in a domain-specific language (DSL), inspired by language of thought models in computational cognitive science [30, 33, 68, 74, 89]. The DSL is used to model the semantics of games in our dataset, independent of the exact natural language phrasing, and was initially derived from the Planning Domain Definition Language (PDDL [32]), which offers a basic representation for specifying goals (i.e. end states of plans) and preferences (i.e. other costs to optimize while planning). Each program in the DSL contains two mandatory sections: gameplay **preferences** describing how a game is played, and **scoring** rules specifying how to determine a player's score based on the satisfaction of the game's preferences. Game programs may also contain optional **setup** instructions and **terminal** conditions (see Supplementary information I for the full DSL).

Our choice to represent games as programs allows us to quantitatively analyze their structure and fundamental components. We found that people recruit an intuitive physical common sense when creating games (Figure 2b, and see Game dataset analyses methods for details). For instance, if an object is thrown, it's likely a ball, and if an object is stacked, it's likely a block — and while a few participants specified games involving throwing blocks, none attempted to stack balls. Similarly, participants did not specify throwing cumbersome objects (such as the laptop or chair), and a participant who specified throwing a large 'beach ball' clarified that it should land *on* the bin (as the ball does not fit within the bin). We also observed evidence of both compositionality (common structure reuse) and creativity (preponderance of unique structures), summarized in Figure 2c (see Game dataset analyses methods for details). Counting occurrences of grammatical structures while abstracting over the identities of individual objects (i.e. treating `(once (agent_holds block))` and `(once (agent_holds ball))` the same), we find the five most common structures cover almost half of the total observations, offering strong evidence of reuse through compositionality. At the other end of the distribution, we also observe a long tail emblematic of creativity, as one-half of the unique structures we count appear exactly once. Despite not being explicitly prompted to generate novel or creative games, many participants proposed entirely unique gameplay ideas, encouraging us that our experimental paradigm elicits rich and creative goal creation.

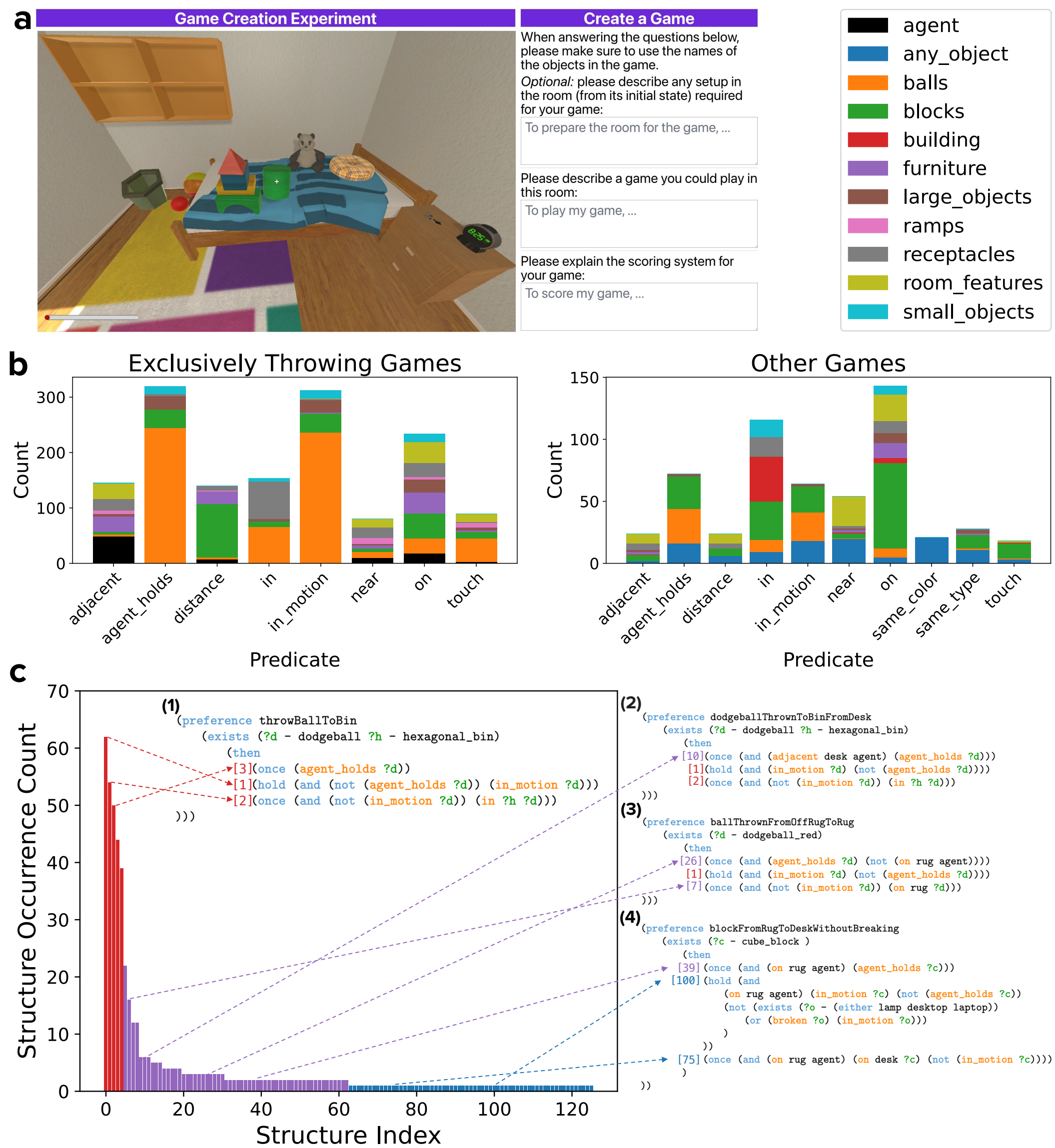


Figure 2: Participants in our behavioral experiment create diverse games reflecting common sense and compositionality. (a): Our online game creation experiment (see full interface in Figure ED-1. **(b):** Participants showcase intuitive common sense. *Left:* In games involving exclusively throwing, participants use balls (orange) far more often than any other object type. *Right:* In other games, participants refer to blocks or "any object" more often, most often checking where objects are placed (using the `in` and `on` predicates). We most often observe balls being thrown and blocks being stacked, and while a few participants specified block-throwing games, no participant created a game involving ball-stacking. Participants also rarely specified throwing large or cumbersome objects (such as the chair or laptop), and only used buildings to specify stacking objectives (as opposed to moving or throwing them). **(c):** We analyze the occurrence of various abstract structures in our programs (see Game dataset analyses methods for details). **Red:** The five most common structures cover almost half (47.5%) of total occurrences, showing extensive compositional reuse. The three most common structures combine into simple ball-to-bin throwing preference (**(1)**, structure indices in square brackets). **Purple:** Other structures are reused fewer times, covering most remaining occurrences (another 40.5%). These rarer structures allow for creating more complex throwing elements, constraining where the player throws the ball from **(2,3)** or to **(3)**. **Blue:** Exactly half of the structures (63 / 126) appear only once — this long tail of expressions offers evidence of creativity. The last throwing preference **(4)**, specifying throwing a block from the rug onto the desk without moving off the rug or breaking any of the objects on the desk, uses two unique structures.

Modelling Results

We next develop a computational model to synthesize human-like goals. Guided by insights from our behavioral analyses, we design our model to explicitly leverage cognitive capacities (i.e. common sense and compositionality) that people seem to recruit in creating goals. Our Goal Program Generator model (GPG, illustrated in Figure 3) operates over a high-dimensional program space and learns how to generate goals maximizing a fitness measure. Upon entering a new environment, people can create goals without extensive data-driven demonstrations; therefore we aim for a model that can similarly generate goals without a large number of examples.

Central to GPG is a fitness function $f(g) = \theta \cdot \phi(g)$ that maps $f : \mathcal{G} \to \mathbb{R}$ from a game $g \in \mathcal{G}$ to a real-valued score that aims to encode its human-likeness. We transform each game into an 89-dimensional vector of features that capture properties relating to structure (e.g., the size and depth of its syntax tree) or logic (e.g., whether any expressions are redundant). Other features proxy cognitive capacities, such as physical common sense (estimating predicate feasibility from play data) or compositionality (n-gram model over syntax elements). We leverage our programmatic representation of goals in order to automate this feature extraction process (see Fitness function methods for details).

Our framework is most committed to goals-as-programs that can be scored via a learned fitness function; it is less committed, at present, to the specific algorithms for parameter learning and program search [58]. In this implementation, parameter learning of feature weights θ proceeds in a *contrastive* fashion [16, 48] by optimizing for the difference in scores between our set of human-generated games and a substantially larger set of corrupted (i.e. lower quality) games obtained through random tree-regrowth [33] on our dataset (see Figure ED-2a and details in Fitness function methods). This learned fitness function then guides an evolutionary search procedure in order to generate novel games. Broadly inspired by work in genetic programming, we use a *quality-diversity* algorithm [69, 14] called MAP-Elites [62] to generate a set of samples that widely cover the space of programs in addition to optimizing the fitness function (see Figure ED-2b). The details of our implementation, including the particular criteria used for maintaining sample diversity, are available in MAP-Elites methods.

Generated games

GPG produces a variety of outputs that range from variants of simple games in our reference dataset to games in entirely new regions of program space. In Figure 4, we present examples of `matched` games generated by our model that "match" a game in our `real` human dataset (provided for reference) by occupying the same niche defined by the MAP-Elites algorithm. In the first pair (Figure 4, left), the model proposes an original block-stacking objective: where the human participant created a tower, the model asks to stack three blocks all on the same taller block. The second and third pairs (Figure 4, middle and right) demonstrate the model's ability to propose throwing games. In both cases, the model proposes interesting detailed objectives, some unseen in our training set (e.g., throwing balls onto the top shelf or desk), that match the niche of the participant games by having the same high-level configuration. However, minor elements in generated games tend to be less readily explainable (e.g. the scoring condition in the left-most generated game, which arbitrarily multiplies the number of satisfactions by 0.4). Our model also produces `unmatched` games that occupy niches without corresponding human games (Figure 5). These include whimsical combinations of throwing and block-stacking (Figure 5, left), a game that combines ball throwing and small object placement (Figure 5, middle), and a game that offers a collection of varied block-stacking objectives (all-on-one, a T-shape, and a tower; Figure 5, right). Though these programs represent creative goals, with preferences that are each individually sensible, at times they struggle to combine into a coherent whole (e.g., the golf ball throwing and block placement elements in Figure 5, left, which do not readily describe an easily explainable game).

More quantitatively, Figure ED-3 shows that GPG quickly produces games with fitness scores in the range of human samples and does so across many of the niches defined by our

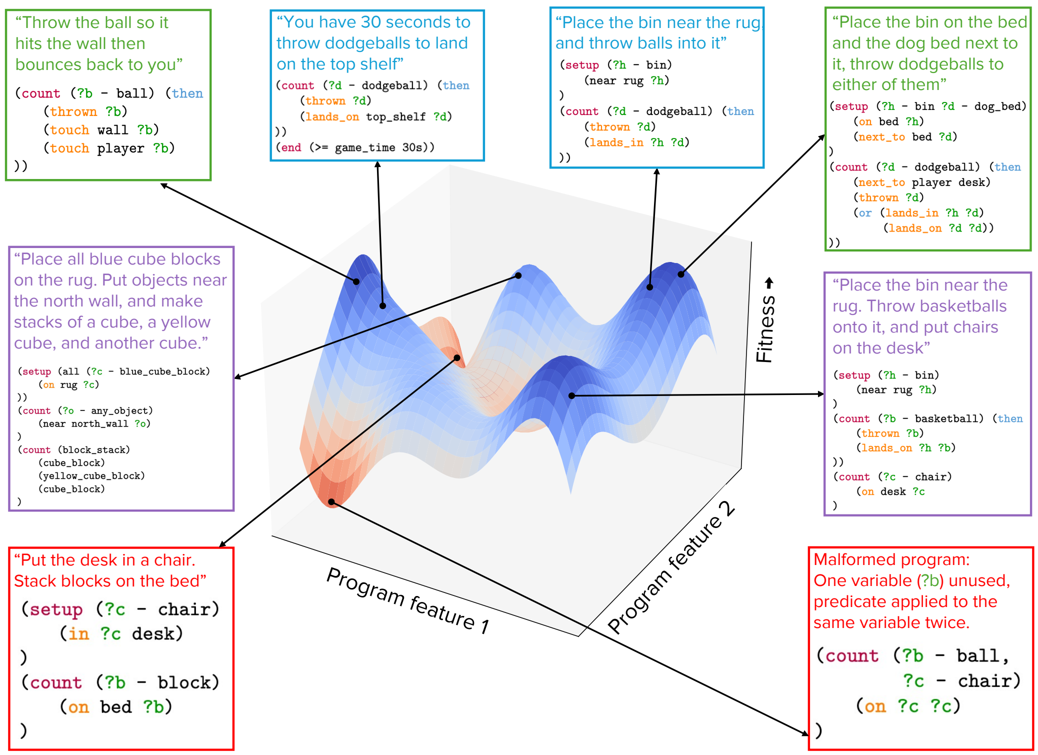


Figure 3: Goal Program Generator model overview. Our model operates on programs in some high-dimensional space (visualized in two dimensions). We learn a fitness metric (Z-axis) capturing desirable aspects of programs, using a dataset of human-created goals (highlighted in green). Our model then generates diverse new samples maximizing the fitness measure, some “matching” participant-created goal programs on diversity criteria (in blue) and other “unmatched” novel goals (in purple). These programs stand in contrast to potential failure modes, such as generating programs that are malformed or semantically incoherent (in red). All (non-red) goals in this figure were created by participants in our experiment or by our model; see Figure SI-1 for their full representations in our domain-specific language.

behavioral characteristics. Considering only games that pass a plausibility characteristic we include in our MAP-Elites search (see MAP-Elites methods for details), 1889 games (94.45%) exceed the fitness score of the least fit real game, and exactly 1000 games exceed the fitness of the median human game. To the extent that our fitness function captures human likeness, our model produces human-like games; we next use human evaluators to extrinsically test our model.

Participant Game #14 (36.491)

Gameplay: stack a flat block on a bridge block, then place a tall cylindrical block on the flat block, followed by a cube block on the tall cylindrical block, and finally a pyramid block on the cube block.

Scoring: you get 10 points for each different set of blocks you have successfully stacked in this order by the end of the game

```
(define (game game-14) (: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))
))
```

Participant Game #31 (37.338)

Gameplay: throw a ball so that it touches a wall and then either catch it or touch it

Scoring: you get 1 point for each time you successfully throw the ball, it touches a wall, and you are either holding it again or touching it after its flight

```
(define (game game-31) (:domain many-objects-room-v1)
(:constraints
  (and
    (preference ballThrownToWallToAgent
      (exists (?b - ball ?w - wall)
        (then
          (once (agent_holds ?b))
          (hold-while (and (not (agent_holds ?b)) (in_motion
              ?b)) (touch ?w ?b))
          (once (or (agent_holds ?b) (touch agent ?b)))
)))))
(:scoring
  (count ballThrownToWallToAgent)
))
```

Participant Game #40 (36.152)

Setup: Place a green golf ball near the door and ensure it remains there for the entire game. During the game, place at least one dodgeball near the green golf ball.

Gameplay: While standing next to the green golf ball and the door, throw dodgeballs with the goal of getting them to stop inside a hexagonal bin.

Terminal: The game ends when you have thrown the same dodgeball and it has stopped moving more than once, or when you have thrown and stopped at least three different objects.

Scoring: You earn 10 points for each different object that you successfully throw into the hexagonal bin.

```
(define (game game-40) (:domain many-objects-room-v1)
(:setup
  (exists (?g - golfball_green)
    (and
      (game-conserved
        (near door ?g)
     )
      (forall (?d - dodgeball)
        (game-optional
          (near ?d ?g)
)))))
(:constraints
  (and
    (forall (?d - dodgeball)
      (and
        (preference dodgeballThrownToBin
          (exists (?h - hexagonal_bin ?g - golfball_green)
            (then
              (once (and (adjacent ?g agent) (adjacent door
                  agent) (agent_holds ?d)))
              (hold (and (in_motion ?d) (not (agent_holds ?d))
                  ))
              (once (and (not (in_motion ?d)) (in ?h ?d)))
            )
          )
        )
        (preference throwAttemptFromDoor
          (exists (?g - golfball_green)
            (then
              (once (and (adjacent ?g agent) (adjacent door
                  agent) (agent_holds ?d)))
              (hold (and (in_motion ?d) (not (agent_holds ?d))
                  ))
              (once (not (in_motion ?d)))
)))))))
(:terminal
  (or
    (> (external-forall-maximize (count throwAttemptFromDoor)
        ) 1)
    (>= (count-once-per-objects throwAttemptFromDoor) 3)
))
(:scoring
  (* 10 (count-once-per-objects dodgeballThrownToBin))
))
```

Matched Model Game (36.994)

Gameplay: stack three blocks on top of a tall rectangular block, with two of the stacked blocks being the same type as the tall rectangular block.

Scoring: your score is 1.4 times the number of such stacks you have at the end of the game.

```
(define (game gpg-8176-22-0) (:domain many-objects-room-v1)
(:constraints
  (and
    (preference preference0
      (exists (?v0 - block ?v1 - block ?v2 -
          tall_rectangular_block ?v3 - block)
        (at-end
          (and
            (on ?v2 ?v0)
            (on ?v2 ?v1)
            (on ?v2 ?v3)
            (same_type ?v0 ?v2)
            (same_type ?v1 ?v2)
))))))
(:scoring
  (+ (* 0.4 (count preference0))
    (count preference0)
)))
```

Matched Model Game (37.324)

Gameplay: throw dodgeballs so that they land and come to rest on the top shelf

Terminal: the game ends after 30 seconds

Scoring: you get 1 point for each dodgeball that is resting on the top shelf at the end of the game

```
(define (game gpg-8176-22-0) (:domain many-objects-room-v1)
(:constraints
  (and
    (preference preference0
      (exists (?v0 - block ?v1 - block ?v2 -
          tall_rectangular_block ?v3 - block)
        (at-end
          (and
            (on ?v2 ?v0)
            (on ?v2 ?v1)
            (on ?v2 ?v3)
            (same_type ?v0 ?v2)
            (same_type ?v1 ?v2)
))))))
(:scoring
  (+ (* 0.4 (count preference0))
    (count preference0)
)))
```

Matched Model Game (37.020)

Setup: Place a hexagonal bin near the rug and ensure it remains there for the entire game.

Gameplay: Throw dodgeballs aiming to land them on the desk or in side the hexagonal bin.

Scoring: You earn points for each dodgeball that comes to rest either on the desk or inside the hexagonal bin. Your final score is the sum of these points.

```
(define (game gpg-8178-6-0) (:domain medium-objects-room-v1)
(:setup
  (exists (?v0 - hexagonal_bin)
    (game-conserved
      (near rug ?v0)
)))
(:constraints
  (and
    (preference preference0
      (exists (?v0 - dodgeball)
        (then
          (once (agent_holds ?v0))
          (hold (and (not (agent_holds ?v0)) (in_motion ?v0)))
          (once (and (not (in_motion ?v0)) (on desk ?v0)))
        )
      )
    )
    (preference preference1
      (exists (?v0 - hexagonal_bin ?v1 - dodgeball)
        (then
          (once (agent_holds ?v1))
          (hold (and (not (agent_holds ?v1)) (in_motion ?v1)))
          (once (and (not (in_motion ?v1)) (in ?v0 ?v1)))
)))))
(:scoring
  (+ (count preference1) (count preference0))
))
```

Figure 4: Goal Program Generator model produces simple, coherent, human-like games. Each pair of games in a column has the same set of MAP-Elites behavioral characteristics. Parentheses: the fitness score assigned by the model to each game. Natural language descriptions are generated through automated back-translation from programs (see Supplementary information E for details). To ascertain that the model-generated programs are distinct from training set examples, we also provide in Figure SI-2 the most similar real exemplar using an edit distance, and see Supplementary information F for details.

Unmatched Model Sample (36.066)

Setup: place a hexagonal bin near the north wall and make sure it stays there throughout the game.

Gameplay: throw golfballs aiming to have them stop on and inside the hexagonal bin, and stack blocks so that each has three cube blocks on top, with one cube block being the same type as the block it's on.

Scoring: you score points based on the number of correctly stacked blocks minus four times the number of golfballs that stop on and in side the hexagonal bin.

```
(define (game gpg-8172-48-1) (:domain many-objects-room-v1)
(:setup
  (exists (?v0 - hexagonal_bin)
    (game-conserved
      (near north_wall ?v0)
)))
(:constraints
  (and
    (preference preference0
      (exists (?v0 - hexagonal_bin ?v1 - golfball)
        (then
          (once (agent_holds ?v1))
          (hold (and (not (agent_holds ?v1)) (in_motion ?v1)))
          (once (and (on ?v0 ?v1) (not (in_motion ?v1)) (in
              ?v0 ?v1)))
        )
      )
    )
    (preference preference1
      (exists (?v0 - cube_block ?v2 - cube_block ?v3 - block
          ?v4 - cube_block)
        (at-end
          (and
            (on ?v3 ?v0)
            (on ?v3 ?v2)
            (on ?v3 ?v4)
            (same_type ?v4 ?v3)
))))))
(:scoring
  (+ (* -4 (count preference0))
    (count preference1)
)))
```

Unmatched Model Sample (36.881)

Gameplay: throw dodgeballs and place credit cards or CDs into a hexagonal bin

Scoring: you get 40 points for each dodgeball that ends up in the hexagonal bin multiplied by the number of credit cards or CDs in the bin, plus 1 point for each dodgeball thrown regardless of where it lands .

```
(define (game gpg-8189-236-1) (:domain many-objects-room-v1)
(:constraints
  (and
    (preference preference0
      (exists (?v0 - hexagonal_bin ?v1 - dodgeball)
        (then
          (once (agent_holds ?v1))
          (hold (and (not (agent_holds ?v1)) (in_motion ?v1)))
          (once (and (not (in_motion ?v1)) (in ?v0 ?v1)))
        )
      )
    )
    (preference preference1
      (exists (?v1 - dodgeball)
        (then
          (once (agent_holds ?v1))
          (hold (and (not (agent_holds ?v1)) (in_motion ?v1)))
          (once (not (in_motion ?v1)))
        )
      )
    )
    (preference preference2
      (exists (?v2 - (either credit_card cd) ?v3 -
          hexagonal_bin)
        (at-end
          (in ?v3 ?v2)
)))))
(:scoring
  (+ (* 40 (count preference0) (count preference2))
    (count preference1)
)))
```

Unmatched Model Sample (35.872)

Gameplay: Stack blocks in specific configurations

Scoring: You get 1 point for each stack where one cube block is on top of another cube block with a tall rectangular block on the same cube block. You also get 1 point for each stack where a cube block is on top of a tall rectangular block, which is on top of another cube block, with an additional cube block on top of the tall rectangular block, provided the bottom cube block is the same type as the tall rectangular block. Additionally, you get 1 point for each stack where a tall rectangular block is on top of a cube block, which is on top of another cube block, and the top cube block is the same type as a third cube block. Your final score is the sum of points from these three configurations.

```
(define (game gpg-8127-244-0) (:domain many-objects-room-v1)
(:constraints
  (and
    (preference preference0
      (exists (?v0 - cube_block ?v1 - cube_block ?v2 -
          tall_rectangular_block ?v3 - cube_block)
        (at-end
          (and
            (on ?v3 ?v0)
            (on ?v3 ?v2)
            (on ?v3 ?v1)
          )
        )
      )
    )
    (preference preference1
      (exists (?v0 - cube_block ?v1 - cube_block ?v2 -
          tall_rectangular_block ?v3 - cube_block)
        (at-end
          (and
            (on ?v2 ?v0)
            (on ?v1 ?v2)
            (on ?v2 ?v3)
            (same_type ?v1 ?v2)
          )
        )
      )
    )
    (preference preference2
      (exists (?v0 - cube_block ?v1 - cube_block ?v2 -
          tall_rectangular_block ?v3 - cube_block)
        (at-end
          (and
            (on ?v3 ?v2)
            (on ?v0 ?v3)
            (same_type ?v3 ?v1)
))))))
(:scoring
  (+ (* 1 (count preference1) (count preference0))
    (count preference2)
)))
```

Figure 5: Goal Program Generator model produces 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. Parentheses: the fitness score assigned by the model to each game.

Human evaluations

To systematically and extrinsically evaluate our model, we performed human evaluations using a second set of human participants ($n = 100$; see Figure ED-4 for the evaluation interface and Human evaluation methods for details). Evaluated games belonged to one of three different categories mentioned above: `real` participant-created games from our behavioral experiment, or `matched` or `unmatched` model-generated games (games in Figure 4 and Figure 5 were included; see Human evaluation methods for details). Participants evaluated three games in each category above (without knowing their categories) in a randomized order and provided Likert scale ratings on each game for seven measures, including human likeness, fun, and creativity. Our final dataset includes 892 participant-game evaluations, each consisting of a rating for all seven measures.

Table 1 summarizes the quantitative responses from our human evaluations. We begin with a simple statistical comparison of the ratings of the games in the different categories using the nonparametric Mann-Whitney U-test [57] (and see Human evaluation methods for additional details). Participants respond similarly to the `real` and `matched` games, with no statistically significant differences in the average response scores across all seven attributes. On the other hand, the `unmatched` games differ on a number of attributes. Compared to both `real` and `matched` games, participants judge them to be less easily understood, less fun to play and watch, and less human-like. One potential explanation for the apparent similarity between `matched` and `real` games is that the former simply replicate the latter in form and function. We examined this question and found that `matched` and `real` games have substantial functional differences (see summary in Figure ED-5, details in Supplementary information G.4, and methodological details in Sample similarity comparison methods).

To further analyze these differences and the extent to which they are mediated by our fitness measure, we performed a mixed-effects regression analysis whose results are summarized

Table 1: Human evaluation result summary

Measure	Mean score by category Real (R)	Matched (M)	Unmatched (U)	Significance of difference R vs M	R vs U	M vs U
Understandable ↑	3.943	3.923	3.331	-	***	***
Fun to play ↑	2.522	2.430	2.068	-	***	***
Fun to watch ↑	2.385	2.313	2.024	-	***	**
Helpful† ↑	2.997	2.987	2.840	-	-	-
Difficult ↓↑	2.582	2.660	2.676	-	-	-
Creative ↑	2.318	2.213	2.143	-	*	-
Human-like ↑	2.813	2.670	2.119	-	***	***

Evaluators don't distinguish between participant-created `real` and `matched` model games, but do distinguish `unmatched` games from both. Participants responded to seven Likert questions on a 5-point scale, one for each attribute in the first column (see Human evaluation methods). We report the nonparametric Mann-Whitney U test [57] for differences in outcomes. See Supplementary Table SI-2 for test statistics and P-values. *: $P < 0.05$, **: $P < 0.01$, ***: $P < 0.001$.
†: The full measure description is "Helpful for interacting with the simulated environment." In most measures, higher scores are better, indicated by the ↑, other than *Difficult* ↓↑, in which 3 means "appropriately difficult", and scores below and above indicate too easy and too hard respectively.

in Table ED-1. We fit independent models using each of the seven attributes we asked our human evaluators to judge as the dependent variables. We include fixed effects for the fitness score and membership in the `real` and `matched` groups (treating the `unmatched` group as a baseline), and random effects for the participants and individual games (see Human evaluation methods and Supplementary Table SI-3 for full details). We find that our fitness function captures many of the evaluated attributes: higher fitness predicts higher ratings of understandability, fun to play, and human likeness ($\beta_{\text{fit}} > 0$); conversely, higher fitness also predicts lower ratings of helpfulness, difficulty, and creativity ($\beta_{\text{fit}} < 0$). Our positive findings are promising: they indicate that our fitness function, learned to maximize human likeness in a symbolic program space, also captures intuitive human notions of understandability and fun. Conversely, we view the negative relations as evidence of some degree of mode-seeking: our fitness measure likely assigns the highest scores to the games most representative of the dataset at large. These modal games are plausibly neither particularly creative nor difficult, which means that participants might find also them less helpful for learning the details of the environment. Finally, differences in attribute ratings persist between groups even accounting for any mediating effects of fitness scores (see Supplementary information G.2 for details)

We also performed ablations of key model components corresponding to the cognitive capacities we found our participants recruited. To ablate *physical common sense*, we remove from our fitness function the two features that estimate the feasibility of a game's preferences by leveraging our database of participant-environment interactions. Analogously, we ablate the intuitive *coherence* we observe in human goals by removing the features that capture the coordination of gameplay elements between different sections. Ablating *compositionality* is more difficult, as our programmatic representation is inherently compositional. We do so by removing the crossover mutation operator used to generate new samples during MAP-Elites, which most explicitly leverages the compositional structure of games. In all cases, model performance degrades substantially, either in sample fitness scores or in goal plausibility as estimated using our database of participant-environment interactions (see Supplementary information H for further details).

Discussion

Goals are a critical aspect of human cognition and, in fact, the starting place for many models of human behavior. However, the representation of goals is often impoverished. In this article, we proposed a new framework for understanding human goals as reward-producing programs, and thus understanding goal generation as program generation. To evaluate this framework, we developed an interactive experiment in which participants created playful goals, operationalized as games to be played in a virtual environment. By analyzing the program-based translation of these games, we highlighted several cognitive capacities recruited by our participants, such as physical common sense and compositionality. These capacities, in turn, informed our modeling efforts. We then built a computational model that learns from a small dataset of games and generates goals that are both novel and human-like according to human evaluators.

This work unites various strands of research in cognitive science, artificial intelligence, and game design. First, we build on a substantial literature studying the psychology of goals [25, 6, 27, 60, 19]. We emphasize open-ended goal creation given that generating new exemplars is a core capacity of human conceptual representations [88] and the utility of games in the study of cognition [2]. Our work also relates to goal-conditioned reinforcement learning [55], and we aim to improve on the goal representations used for such agents that tend to lack the variety and richness of human-created goals [21, ch. 7]. Our goal program interpreter conceptually draws on the notion of reward machines introduced in [42]. Finally, we are inspired by the automatic game design literature, such as synthesizing board game variants [66, 38, 12] or simple video games [83, 75, 91, 45]. Unlike our approach, these efforts often optimize program synthesis for some heuristic notion of fun [12, 83] rather than explicitly modeling human-like game generation.

In instantiating our model, we necessarily make a number of specific implementational and algorithmic decisions. Our framework is committed to the representation of goals as *reward-producing programs*: computationally executable mappings from agent behavior to indications of progress towards a goal. We find it crucial that these programs capture the rich, temporally extended nature of goals people create, and that they facilitate the flexible and compositional creation that people seem to engage in [47, 88]. Our model operates over these programs and assumes two primary components: a measure of human likeness and a method to sample novel programs. The model we instantiate operates at Marr's [58] computational level; it tackles the *what* of goal generation without directly inspecting *how* people do so. We are excited for future work to seek algorithmic-level signatures and constraints on how children and adults flexibly generate rich goals and leverage them to build models that operate closer to Marr's algorithmic level. Particularly, understanding whether specific primitives in our DSL map onto cognitive primitives will require further work.

Our model strongly relies on its approach to sample diversity, which arises from the choice of "behavioral characteristics" that define the axes along which the MAP-Elites algorithm maintains diversity. In this work, we select behavioral characteristics based on notable gameplay components observed in our human dataset; future work could explore other techniques for maintaining diversity, including the automated selection of behavioral characteristics [22, 36]. Our current features approximating intuitive physical common sense are indirect, using participant interactions with the environment to estimate feasibility. Future approaches could integrate planning or physical simulation to improve our model's understanding of physics [84, 15]. Finally, our model is inherently coupled to the environment and dataset we collected — particularly given the engineering effort to instantiate various types of knowledge. This approach has some distinct advantages: we can isolate various cognitive capacities, interpret their contribution to our fitness measure (Supplementary information B.1), and ablate their roles (Supplementary information H). Simultaneously, some of the challenges our model faces (such as coherence between program components) might be alleviated by incorporating natural language or by leveraging the capabilities of large language models to write code and adapt to in-context instructions. Language models could also alleviate our current reliance on manual (and potentially arbitrary) translations from participant game descriptions to the proposed mental language of goal programs (see [89] for a discussion on using language to construct meaning through programs, and [79] building programs to act as world models).

We see two particularly promising ways in which our representational framework could be used going forward. First, there is increasing interest in building artificial agents that can flexibly explore and generalize across environments [72, 31]. The *autotelic* perspective argues that empowering agents to propose and pursue self-generated goals is a fruitful way to improve their ability to generalization [21]. However, goals in such systems are often derived from agent or object positions [29, 80], short natural language descriptions [24, 87], or limited temporally-aware mechanisms [54, 51] — all impoverished when compared with the diverse goals humans flexibly create. We are excited for future work to empower artificial agents with richer goals that reflect human-like novelty and difficulty, for two specific reasons. First, we believe access to complex and varied goals would enable agents to learn flexible representations of their environments that support higher behavioral adaptability [19]. Second, we view compositional goal production as facilitating effective exploration of unseen goals [20] (and see [90] for a discussion of generalization and exploration). We also note our current approach estimates goal fitness without considering additional higher-level objectives that might guide goal generation. Prior literature offers curiosity [82, 9], empowerment [34, 1, 23], information gain [73, 53], novelty [78, 9], and learning progress [81, 8] as compelling potential objectives. Future work could instantiate goal generators that consider these objectives as auxiliary terms to the fitness function and compare the behaviors that arise in artificial agents through pursuing them.

If we are to understand goals as programs, our proposed framework may also help advance our understanding of intuitive psychology and goal inference [76, 44, 56]. Previous work proposed that our ability to understand other people's goals, as part of our Theory of Mind, operates through inverse reinforcement learning: inferring an agent's reward from observing their behavior [43]. Many prior approaches eschew goals entirely, using some function approximator (e.g., a neural network) to estimate reward, resulting in an uninterpretable estimator that can struggle to generalize [5]. We envision leveraging our goal programs as a prior distribution for a Bayesian Theory of Mind [7] approach, scaling up previous approaches that relied on a small number of predefined goals [86], to create models that would parse an agent's behavior and provide an interpretable, semantically explicit estimate of their goal [37]. Applying our framework to either of these proposed problems would offer a substantial long-term challenge building on the work we present in this article. Nevertheless, we see an exciting prospect to leverage this approach to improve the understanding of human goals and endow machines with human-like goal concepts and capabilities.

Acknowledgements

Author GD thanks members of the Human and Machine Learning Lab and the Computation & Cognition Lab at NYU for their feedback at various stages of this project. Authors GD and BML are supported by the National Science Foundation under NSF Award 1922658. Author GT is supported by the NSF GRFP. Author TMG's work on this project is supported by NSF BCS 2121102.

Methods

Dataset collection methods

Experimental design: After a consent form and instructions quiz, participants completed a tutorial designed to familiarize them with the controls for our environment. After successfully completing the tutorial, they were placed in one of three variations of the main experiment room, with the same structure but different amounts of available toys and objects. Participants were then free to explore this new room until they had a game ready, and could freely reset it to its initial state in the meantime. Participants were asked to create games with the following restrictions: single-player, require no additional space or objects that they do not see in the room, and include a scoring system. While the latter constraint may seem limiting, we note that any arbitrary goal can be scored by rewarding the achievement of the goal.

Dataset collection: Participants then reported their game in natural language in three text boxes, one of which was optional (see Figure ED-1). The optional first one allowed specifying whether there was any setup or preparation required to get the room from its default initial state to one that would allow playing the game (e.g., placing the bin on the bed). The second text box allowed participants to describe the game's gameplay, and the third offered space to describe the scoring rules. To encourage participants to imagine playing their game, they were also asked to report their perceived difficulty level and how many points they thought they might score if they played it. Participants then had a chance to play their game and revise it should they want to; if participants opted to revise their games, we analyzed the revised ones. We contacted 192 participants via Prolific [65] of whom 114 finished the experiment and another 12 were paid due to technical difficulties. Participants were paid a base rate of $10 and received a $2 bonus if their game satisfied the required constraints. Successful participants took 44.4 minutes on average, with a standard deviation of 23.3 minutes. We then excluded 8 games that did not satisfy the constraints we posed on participants, 6 duplicates (including some due to technical difficulties from participants who restarted the experiment), and 6 other games that were unclear or under-specified. After accounting for two other games we opted to avoid modeling due to their complexity (one referring directly to the game interface, and control, and another describing several games or levels in the single description we collected) , we arrived at our final dataset of 98 games. We acknowledge the potential arbitrariness of manually translating from natural language to our program representations; we attempted to be maximally faithful to the descriptions and excluded participants whose games required too much subjectivity or interpretation.

Interaction traces: In addition to the game descriptions in natural language, we record traces of participants' interactions with the environment. We record state-action traces to allow us to replay and examine how participants interact with our environment. We record separate traces for each different segment of the experiment (before creating the game; while reporting their game; playing their game; after editing their game), and for each time the participant resets the environment within each segment. We end up with 382 total such traces. Our primary use for them is in implementing a "reward machine," an interpreter for our goal programs, which parses a goal program into a state machine, and iterates through a trace to emit the score of that trace under the goal. We use a limited version of this in our fitness features (see Fitness function methods for additional details) and in some of our model evaluations and ablations (see Supplementary information H for additional details).

Game dataset analyses methods

Common sense through predicate role-filler analysis: We analyze predicate role-filler occurrences, coarsening individual objects to higher-level categories (see the legend on the right of Figure 2b). To split between the two panels of Figure 2b, we categorize each game by whether it includes the following motifs: throwing (e.g., balls into a bin), stacking (e.g., blocks in a building), organizing (e.g., placing objects in specified places), or other. We split the figure into games involving only throwing motifs (left panel) and games involving any other motifs, potentially in addition to throwing (right panel). In games involving only throwing (left panel), participants most often refer to balls, primarily checking whether or not the agent holds a ball or a ball is in motion (as part of quantifying the act of throwing). Other predicates are often used to specify some additional conditions on throwing (such as specifying the bin being on the bed or the agent being next to the desk) and are used with a variety of object categories. Conversely, in games involving other elements (right panel), we see blocks and the generic "any_object" being used far more often, mostly in various placement and stacking constraints.

Compositionality and creativity through abstract structure occurrence: We analyze how often participant games make use of various grammatical structures to showcase both compositional reuse and long-tail creativity. Each structure involves a temporal modal (such as `once` or `hold`) and the predicate expression nested under it, such as `(once (agent_holds ?b))`, where `?b` is a variable quantified earlier. We count structures, abstracting away specific variables and their types – so the expression above would be coarsened as `(once (agent_holds <obj>))`, and would be counted together with any other expression coarsened to this pattern. We encounter a total of 126 unique expressions in our dataset, the most common one with 62 occurrences being `(hold (and (not (agent_holds <obj>))(in_motion <obj>)))`, which maps loosely to "find a sequence of states where an object is not held and is in motion" — that is, is currently moving with the agent touching it, for instance while being thrown or rolled. Of the 126 expressions, exactly half (63) occur only once.

Fitness function methods

Fitness function form: The fitness function used by our model is a learned, weighted linear combination of a set of features extracted programmatically from each game that is optimized to assign high scores to "human-like" games and low scores to everything else. It is a function $f : \mathcal{G} \to \mathbb{R}$ that maps individual games $g \in \mathcal{G}$ to real-valued scores: $f(g) = \theta \cdot \phi(g)$, where θ is a learned vector of weights and $\phi : \mathcal{G} \to [0,1]^F$ is a feature extractor.

Feature extractor and feature set: The feature extractor ϕ represents each game as an 89-dimensional vector (i.e. $F = 89$). Each entry in the vector corresponds to a particular structural or semantic property of the game, from the size and depth of the syntax tree to the apparent feasibility of the game's preferences. We normalize the values of each property to fall within the unit interval by using the observed range of values in our dataset. Many features used in the fitness function are directly computable from the DSL representation of a game (for instance, properties of its syntax tree or the misuse of particular grammatical structures). While these features represent the majority of the 89 features used, we also implement two important sets of features that require additional computation.

The first of these are n-gram features that capture the mean log score of the game under a simple n-gram language model trained over the set of human-generated syntax trees. We fit n-gram models using stupid backoff [10] and a discount factor of 0.4 We compute these scores separately for each game section (i.e. setup, preferences, terminal, and scoring) and also for the game overall, resulting in 5 features.

The second set consists of two features that make use of an interpreter that parses game programs into "reward machines" [41]: finite-state machines that process a trace of player inputs and emit a reward whenever the particular scoring conditions of the game are met. The interpreter programmatically implements each of the predicates in the DSL, which allows us to construct a dataset of which objects were used to satisfy which predicates across our dataset of 382 human *play traces*. The two features query this database in order to get an approximate common sense measure of a game's "feasibility," computing the proportion of a game's predicate-argument combinations that have been satisfied by human players in our dataset (one feature does this for individual predicates, while the other does this for boolean logical expressions over predicates). While these feasibility measures give a sense of whether the objectives of a game can be completed in the physical reality of the simulation, the limited nature of our play trace dataset means they are far from perfect proxies.

The complete set of features used (and accompanying descriptions) is available in Supplementary information B, with the most important features (by their learned weights) highlighted in Supplementary information B.1.

Fitness function learning algorithm: To learn the weight vector θ, we take inspiration from the contrastive learning of energy-based models [16] with the objective of separating a set of *positive* examples (our set of human-generated games) from a set of *negative* examples (and see a summary in Figure ED-2a). To learn an effective fitness function, these negatives must be qualitatively worse than our set of human games without being trivially distinguishable from them. We generate a set of plausible negatives by *corrupting* games from our positive set. To corrupt a game, we select a random node in its syntax tree, delete the node and its children, and randomly re-sample a sub-tree according to the DSL grammar (illustrated in red in Figure ED-2a). This "tree-regrowth" approach [33] generally produces sub-trees that are syntactically valid but semantically "out-of-place," with the severity of the corruption tending to correspond to the height of the re-sampled node in the syntax tree. To account for the variance in the difficulty of distinguishing between a given positive and negative example, we generate a large set of negatives: 1024 for each of the 98 positives, for a total of 100,352 negatives.

We train the fitness function (i.e. optimize θ) using a softmax loss, not unlike the MEE loss used to train energy-based models [49] or the InfoNCE loss [85]. For a positive example g^+ and a set of negative examples $\{g_k^-\}, k \in \{1, 2, \cdots, K\}$, we assign the loss:

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

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 Supplementary information C for full details of our training and cross-validation setups.

MAP-Elites methods

MAP-Elites overview: 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, game programs in the DSL) that form the axes of a multi-dimensional *archive* of cells (and see an overview in Figure ED-2b). At each step, a game g is selected uniformly from among the individuals in the archive (Figure ED-2b, step 1) and mutated to form a new game g' (Figure ED-2b, step 2) . 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 (and replaces the previous occupant) only if its fitness is greater than the current occupant of the cell (Figure ED-2b, step 3) . In this way, the algorithm maintains an "elite" for each possible combination of values under the behavioral characteristics.

Behavioral characteristics: Inspired by prior work on using MAP-Elites for procedural content generation [13], we define a set of integer-valued behavioral characteristics that each indicate how many preferences in each archive game match one of nine archetypal exemplar gameplay preferences (illustrated as the axes of the grid in Figure ED-2b). These include several types of ball-throwing preferences, as well as ones capturing block-stacking, object-sorting, and other miscellaneous activities. We also include two other features, one capturing whether or not the game includes a setup component, and one capturing the total number of preferences. For additional details and descriptions of the exemplar preferences, see Supplementary information D. The 11 total behavioral characteristics result in a total archive size of 2000 games.

Plausibility behavioral characteristic: We include a "pseudo behavioral characteristic" that explicitly captures a few general coherence properties of games. This characteristic captures a conjunction of feature values where we expect either *all* plausibly human-generated games to either exhibit or *none* of them to exhibit. 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 all of these criteria are satisfied. There are a total of 21 features used in this behavioral characteristic, and though it doubles the size of the MAP-Elites archive (from 2000 games to 4000 games), we never evaluate any game from the half of the archive in which this feature is false. See Supplementary information D for additional details.

Mutation operators: 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 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 candidates from the PCFG—*not* with real, human-participant-created games or corruptions thereof that were used to train the fitness function. Further details of the algorithm are available in Supplementary information D.

Human evaluation methods

Evaluation dataset: we select games to be evaluated using the following procedure:

1. `real`: We include 30 participant-created games, each with a different set of behavioral characteristics — that is, each being considered 'different' according to how our model searches through the space of games (see MAP-Elites methods) for additional details).

2. `matched`: For each of the `real` games included above, we include the model-generated game from our final model from the corresponding MAP-Elites archive cell. Each of these games includes the same number of gameplay preferences as the corresponding `real` participant-created games, matching the same exemplar preferences.

3. `unmatched`: We then sample 30 additional games from our model's archive. We sample in a fashion that aims to be balanced across the different preference counts and usage of the different exemplar preferences. That said, given that human games cover only 47 out of the 2000 archive cells, that leaves 1953 potential `unmatched` games to sample; it is difficult to know how representative our set of 30 (which is about 1.5%) is.

GPT-4-based back-translation: Rather than ask participants to interpret our domain-specific language, we use the GPT-4 [63] language model to perform a multi-step back-translation from programs in our domain-specific language to structured natural language. For fairness and consistency, we use this procedure on the `real` games in addition to the model-generated `matched` and `unmatched` games. We first apply a rule-based system to apply templates, translating expressions in the DSL to natural language sentence fragments. We then use GPT-4 to first map the templated fragments to a more natural language, and then to combine the description of each game component (setup, gameplay preferences, terminal conditions, and scoring rules) to a short coherent description. See Supplementary information E for full details and prompts used.

Human evaluations structure: Figure ED-4 presents our human evaluation interface. Following instructions and an understanding quiz, participants evaluated nine total games: 3 `real` ones, the corresponding 3 `matched` ones, and 3 `unmatched` ones. Participants were presented one game at a time and provided two short textual responses, one explaining the game in their own words, and one providing a short overall impression of the game. Participants also answered seven Likert-type questions on 5-point scales, answering the following questions about the italicized attributes:

1. *Understandable:* "How confident are you that you understand the game described above?", where 1: not at all confident, 3: moderately confident, and 5: very confident

2. *Fun to play:* "How fun would it be to play the game yourself?", where 1: not at all fun, 3: moderately fun, and 5: extremely fun.

3. *Fun to watch:* "How fun would it be to watch someone else play this game?", where 1: not at all fun, 3: moderately fun, and 5: extremely fun.

4. *Helpful:* "Imagine that you played this game for several minutes. How fun would it be for learning to interact with the virtual environment?", where 1: not at all helpful, 3: moderately helpful, and 5: extremely helpful.

5. *Difficult:* "Imagine that you played this game for several minutes. Do you think it would be too easy, appropriately difficult, or too hard for you?", where 1: far too easy, 3: appropriately difficult, and 5: far too hard.

6. *Creative:* "How creatively designed is this game?", where 1: not at all creative, 3: moderately creative, and 5: extremely creative.

7. *Human-like:* "How human-like do you think this game is?", where 1: not at all human-like, 3: moderately human-like, and 5: extremely human-like.

Evaluation statistical analyses: For each attribute and each game category (`real`, `matched`, and `unmatched`, we report the mean score assigned by all participants to games in that category for that attribute. We then also aggregate these attribute scores by category and report a nonparametric Mann-Whitney U-test [57] for differences in outcomes, as appropriate for ordinal data. See Supplementary Table SI-2 for the full table including test statistics and P-values. Significance results were highly similar when computing two-sample t-tests as an alternative statistical test.

Mixed effect models: We are interested in modeling the relationship between the scores predicted by our fitness function and the attributes human evaluators predicted. To that end, we set up mixed effect regression models [71, 39]. We fit separate models for each measure as the dependent variable, regressing a continuous latent score (e.g., s^i_{fp} for the fun-to-play measure, equation (2) below). We include fixed effects for the fitness score (x^i) and for membership in the `real` ($\mathbb{1}^i_{\text{real}}$) and `matched` ($\mathbb{1}^i_{\text{matched}}$) groups, treating the `unmatched` group as a baseline. We include random effects for the individual participants ($\epsilon_p^{p_i} \sim \mathcal{N}(0, \sigma_p^2)$) and evaluated games ($\epsilon_g^{g_i} \sim \mathcal{N}(0, \sigma_g^2)$). We also fit a sequence of cut-points (equation (3)) that transform the latent score to the observed ordinal rating y^i_{fp} (equation (4)). We suppress the subscript for each measure below:

$$s^i = \beta_{\text{fit}} x_i + \beta_{\text{real}} \mathbb{1}^i_{\text{real}} + \beta_{\text{matched}} \mathbb{1}^i_{\text{matched}} + \epsilon_p^{p_i} + \epsilon_g^{g_i} + \epsilon^i, \qquad \epsilon^i \sim \mathcal{N}(0, \sigma^2) \tag{2}$$

$$-\infty \equiv c_0 < c_1 < c_2 < c_3 < c_4 < c_5 \equiv \infty \tag{3}$$

$$c_{k-1} < s^i \leq c_k \Rightarrow \text{ observe } y^i = k \tag{4}$$

Models without either random effect performed worse than the full model, so we report results including both random effects. We fit cumulative link models for ordinal regression [4, 35] using the `ordinal` package [17] in **R** [70]. The results of these mixed-effect models are summarized in Table ED-1 (and see Supplementary information G.2 and Table SI-3 for additional details).

Sample similarity comparison methods

For each `real` game and its corresponding `matched` game from those included in the human evaluations, we examine which of our recorded participant interactions (see Dataset collection methods above) fulfills one or more gameplay elements. We treat the setup (if specified) and each gameplay preference as a gameplay element — our aim here is to quantify which participant interaction traces 'play' a part of the game. We do this using our "reward machine" — our implementation of an interpreter for goal programs in this domain-specific language. For each pair of games, we then check which particular interactions either (a) 'play' parts of both games, (b) only fulfill components in the `real` game, or (c) only fulfill components in the `matched` game. We color these proportions in purple, green, and blue respectively in Figure ED-5

References

[1] C. Addyman and D. Mareschal. Local redundancy governs infants' spontaneous orienting to visual-temporal sequences. *Child Dev.*, 84(4):1137–1144, Feb. 2013.

[2] K. R. Allen, F. Brändle, M. M. Botvinick, J. Fan, S. J. Gershman, a. gopnik, T. L. Griffiths, J. Hartshorne, T. U. Hauser, M. K. Ho, J. d. Leeuw, W. J. Ma, K. Murayama, J. D. Nelson, B. v. Opheusden, H. T. Pouncy, J. Rafner, I. Rahwan, R. Rutledge, J. F. Sherson, O. Simsek, H. Spiers, C. Summerfield, M. Thalmann, N. Vélez, A. Watrous, J. Tenenbaum, and E. Schulz. Using games to understand the mind. PsyArXiv, May 2024.

[3] 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 2023.

[4] A. Argesti. *Categorical Data Analysis*. John Wiley & Sons, third edition, 2018.

[5] S. Arora and P. Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artif. Intell.*, 297:103500, Aug. 2021.

[6] J. T. Austin and J. B. Vancouver. Goal constructs in psychology: Structure, process, and content. *Psychological Bulletin*, 120:338–375, 11 1996.

[7] C. Baker, R. Saxe, and J. Tenenbaum. Bayesian theory of mind: Modeling joint belief-desire attribution. In *Proceedings of the annual meeting of the cognitive science society*, volume 33, 2011.

[8] G. Baldassarre, T. Stafford, M. Mirolli, P. Redgrave, R. M. Ryan, and A. Barto. Intrinsic motivations and open-ended development in animals, humans, and robots: an overview. *Front. Psychol.*, 5:985, Sept. 2014.

[9] D. E. Berlyne. Novelty and curiosity as determinants of exploratory behaviour1. *Br. J. Psychol. Gen. Sect.*, 41(1-2):68–80, Sept. 1950.

[10] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 858–867. Association for Computational Linguistics, 2007.

[11] L. Brown. *Psychology of Motivation*. Nova Science Publishers, 2007.

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

[13] 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.

[14] K. Chatzilygeroudis, A. Cully, V. Vassiliades, and J. B. Mouret. Quality-diversity optimization: a novel branch of stochastic optimization. *Springer Optimization and Its Applications*, 170:109–135, 12 2020.

[15] T. Chen, K. R. Allen, S. J. Cheyette, J. Tenenbaum, and K. A. Smith. "just in time" representations for mental simulation in intuitive physics. *Proceedings of the Annual Meeting of the Cognitive Science Society*, 45(45), 2023.

[16] 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.

[17] R. H. B. Christensen. *ordinal—Regression Models for Ordinal Data*, 2023. R package version 2023.12-4.

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

[19] J. Chu, J. B. Tenenbaum, and L. E. Schulz. In praise of folly: flexible goals and human cognition. *Trends Cogn. Sci.*, Apr. 2024.

[20] C. Colas, T. Karch, N. Lair, J. M. Dussoux, C. Moulin-Frier, P. F. Dominey, and P. Y. Oudeyer. Language as a cognitive tool to imagine goals in curiosity-driven exploration. *Advances in Neural Information Processing Systems*, 2020-Decem, 2 2020.

[21] C. Colas, T. Karch, O. Sigaud, and P.-Y. Oudeyer. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. *Journal of Artificial Intelligence Research*, 74:1159–1199, 07 2022.

[22] A. Cully. Autonomous skill discovery with quality-diversity and unsupervised descriptors. *Proceedings of the Genetic and Evolutionary Computation Conference*, page 81–89, 2019.

[23] Y. Du, E. Kosoy, A. Dayan, M. Rufova, P. Abbeel, and A. Gopnik. What can AI learn from human exploration? Intrinsically-Motivated humans and agents in Open-World exploration. In *NeurIPS 2023 workshop: Information-Theoretic Principles in Cognitive Systems*, Nov. 2023.

[24] 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. In *Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202*, 7 2023.

[25] C. S. Dweck. Article commentary: The study of goals in psychology. *Psychological Science*, 3(3):165–167, 1992.

[26] J. S. Eccles and A. Wigfield. Motivational beliefs, values, and goals. *Annu. Rev. Psychol.*, 53:109–132, 2002.

[27] A. J. Elliot and J. W. Fryer. The goal construct in psychology. In J. Y. Shah, editor, *Handbook of motivation science (pp*, volume 638, pages 235–250. The Guilford Press, xviii, New York, NY, US, 2008.

[28] A. Fishbach and M. J. Ferguson. The goal construct in social psychology. In A. W. Kruglanski and E. T. Higgins, editors, *Social psychology: Handbook of basic principles*, volume 2, pages 490–515. The Guilford Press, xiii, New York, NY, US, 2007.

[29] 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.

[30] J. A. Fodor. *The language of thought*. Harvard University Press, 1979.

[31] Q. Gallouédec, E. Beeching, C. Romac, and E. Dellandréa. Jack of all trades, master of some, a Multi-Purpose transformer agent. arXiv, Feb. 2024.

[32] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl – the planning domain definition language, 1998.

[33] N. D. Goodman, J. B. Tenenbaum, J. Feldman, and T. L. Griffiths. A rational analysis of rule-based concept learning. *Cogn. Sci.*, 32(1):108–154, Jan. 2008.

[34] A. Gopnik. Empowerment as causal learning, causal learning as empowerment: A bridge between bayesian causal hypothesis testing and reinforcement learning. PhilSci-Archive, April 2024.

[35] W. H. Greene and D. A. Hensher. *Modeling Ordered Choices: A Primer.* Cambridge University Press, 2010.

[36] L. Grillotti and A. Cully. Unsupervised behavior discovery with quality-diversity optimization. *IEEE Transactions on Evolutionary Computation*, 26(6):1539–1552, 2022.

[37] M. K. Ho and T. L. Griffiths. Cognitive science as a source of forward and inverse models of human decisions for robotics and control. *Annual Review of Control, Robotics, and Autonomous Systems*, 5(Volume 5, 2022):33–53, 2022.

[38] V. Hom and J. Marks. Automatic design of balanced board games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 3, pages 25–30, 2007.

[39] J. Hox, M. Moerbeek, and R. van de Schoot. *Multilevel Analysis (Techniques and Applications)*. New York, NY: Routledge, third edition, 2018.

[40] M. E. Hyland. Motivational control theory: An integrative framework. *Journal of Personality and Social Psychology*, 55(4):642, 1988.

[41] R. T. Icarte, T. Klassen, R. Valenzano, and S. McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. *Proceedings of the 35th International Conference on Machine Learning*, 80:2107–2116, 10–15 Jul 2018.

[42] 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.

[43] J. Jara-Ettinger. Theory of mind as inverse reinforcement learning. *Current Opinion in Behavioral Sciences*, 29:105–110, Oct. 2019.

[44] J. Jara-Ettinger, H. Gweon, L. E. Schulz, and J. B. Tenenbaum. The naïve utility calculus: Computational principles underlying commonsense psychology. *Trends Cogn. Sci.*, 20(8):589–604, Aug. 2016.

[45] 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.

[46] 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.

[47] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman. Building machines that learn and think like people. *Behav. Brain Sci.*, 40:e253, Jan. 2017.

[48] P. H. Le-Khac, G. Healy, and A. F. Smeaton. Contrastive representation learning: A framework and review. *IEEE Access*, Oct. 2020.

[49] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. J. Huang. A tutorial on energy-based learning. In G. Bakir, T. Hofman, B. Scholkopf, A. Smola, and B. Taskar, editors, *Predicting Structured Data*. MIT Press, 2006.

[50] R. V. Lenth. *emmeans: Estimated Marginal Means, aka Least-Squares Means*, 2024. R package version 1.10.0.

[51] B. G. Leon, M. Shanahan, and F. Belardinelli. In a nutshell, the human asked for this: Latent goals for following temporal specifications. *ICLR 2022*, 2022.

[52] A. S. Lillard. The development of play. In L. Liben and U. Mueller, editors, *Handbook of Child Psychology and Developmental Science, Vol. 3: Cognitive Development*, volume 3, pages 425–468. Wiley-Blackwell, 2015.

[53] E. G. Liquin, F. Callaway, and T. Lombrozo. Developmental change in what elicits curiosity. *Proceedings of the Annual Meeting of the Cognitive Science Society*, 43(43), 2021.

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

[55] M. Liu, M. Zhu, and W. Zhang. Goal-Conditioned reinforcement learning: Problems and solutions. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22) Survey Track*, Jan. 2022.

[56] S. Liu, N. B. Brooks, and E. S. Spelke. Origins of the concepts cause, cost, and goal in prereaching infants. *Proc. Natl. Acad. Sci. U. S. A.*, 116(36):17747–17752, Sept. 2019.

[57] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.*, 18(1):50–60, 1947.

[58] D. Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. Henry Holt and Co., Inc., USA, 1982.

[59] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[60] G. Molinaro and A. G. E. Collins. A goal-centric outlook on learning. *Trends Cogn. Sci.*, 27(12):1150–1164, Dec. 2023.

[61] G. B. Moskowitz and H. Grant, editors. *The psychology of goals*, volume 548. Guilford Press, New York, NY, US, 2009.

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

[63] OpenAI. GPT-4 technical report. arXiv, Mar. 2023.

[64] P.-Y. Oudeyer, F. Kaplan, and V. V. Hafner. Intrinsic motivation systems for autonomous mental development. *IEEE Trans. Evol. Comput.*, 11(2):265–286, Apr. 2007.

[65] S. Palan and C. Schitter. Prolific.ac—A subject pool for online experiments. *Journal of Behavioral and Experimental Finance*, 17:22–27, Mar. 2018.

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

[67] L. Pervin. *Goal Concepts in Personality and Social Psychology*. Psychology Library Editions: Social Psychology. Taylor & Francis, 2015.

[68] S. T. Piantadosi, J. B. Tenenbaum, and N. D. Goodman. Bootstrapping in a language of thought: a formal model of numerical concept learning. *Cognition*, 123(2):199–217, May 2012.

[69] J. K. Pugh, L. B. Soros, and K. O. Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3, 2016.

[70] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2023.

[71] S. W. Radenbush and A. S. Bryk. *Hierarchical Linear Models. Applications and Data Analysis Methods*. Thousand Oaks, CA: Sage Publications, second edition, 2002.

[72] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-maron, M. Giménez, Y. Sulsky, J. Kay, J. T. Springenberg, T. Eccles, J. Bruce, A. Razavi, A. Edwards, N. Heess, Y. Chen, R. Hadsell, O. Vinyals, M. Bordbar, and N. de Freitas. A generalist agent. *Transactions on Machine Learning Research*, 2022. Featured Certification, Outstanding Certification.

[73] A. Ruggeri, O. Stanciu, M. Pelz, A. Gopnik, and E. Schulz. Preschoolers search longer when there is more information to be gained. *Dev. Sci.*, 27(1):e13411, Jan. 2024.

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

[75] 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.

[76] E. S. Spelke and K. D. Kinzler. Core knowledge. *Dev. Sci.*, 10(1):89–96, Jan. 2007.

[77] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[78] F. Taffoni, E. Tamilia, V. Focaroli, D. Formica, L. Ricci, G. Di Pino, G. Baldassarre, M. Mirolli, E. Guglielmelli, and F. Keller. Development of goal-directed action selection guided by intrinsic motivations: an experiment with children. *Exp. Brain Res.*, 232(7):2167–2177, July 2014.

[79] H. Tang, D. Key, and K. Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. *arXiv preprint arXiv:2402.12275*, 2024.

[80] 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.

[81] A. Ten, P. Kaushik, P.-Y. Oudeyer, and J. Gottlieb. Humans monitor learning progress in curiosity-driven exploration. *Nat. Commun.*, 12(1):5972, Oct. 2021.

[82] A. Ten, P.-Y. Oudeyer, C. Moulin-Frier, I. C. Dezza, E. Schulz, and C. M. Wu. Curiosity-Driven exploration. In *The Drive for Knowledge: The Science of Human Information Seeking*, pages 53–76. Cambridge University Press, June 2022.

[83] 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.

[84] T. D. Ullman, E. Spelke, P. Battaglia, and J. B. Tenenbaum. Mind games: Game engines as an architecture for intuitive physics. *Trends Cogn. Sci.*, 21(9):649–665, Sept. 2017.

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

[86] J. Velez-Ginorio, M. H. Siegel, J. B. Tenenbaum, and J. Jara-Ettinger. Interpreting actions by attributing compositional desires. In G. Gunzelmann, A. Howes, T. Tenbrink, and E. J. Davelaar, editors, *Proceedings of the 39th Annual Meeting of the Cognitive Science Society, CogSci 2017, London, UK, 16-29 July 2017*. cognitivesciencesociety.org, 2017.

[87] 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.

[88] T. B. Ward. Structured imagination: the role of category structure in exemplar generation. *Cognitive Psychology*, 27:1–40, 8 1994.

[89] L. Wong, G. Grand, A. K. Lew, N. D. Goodman, V. K. Mansinghka, J. Andreas, and J. B. Tenenbaum. From word models to world models: Translating from natural language to the probabilistic language of thought. arXiv, June 2023.

[90] C. M. Wu, E. Schulz, M. Speekenbrink, J. D. Nelson, and B. Meder. Generalization guides human exploration in vast decision spaces. *Nature human behaviour*, 2(12):915–924, 2018.

[91] A. Zook and M. Riedl. Automatic game design via mechanic generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

Extended Data

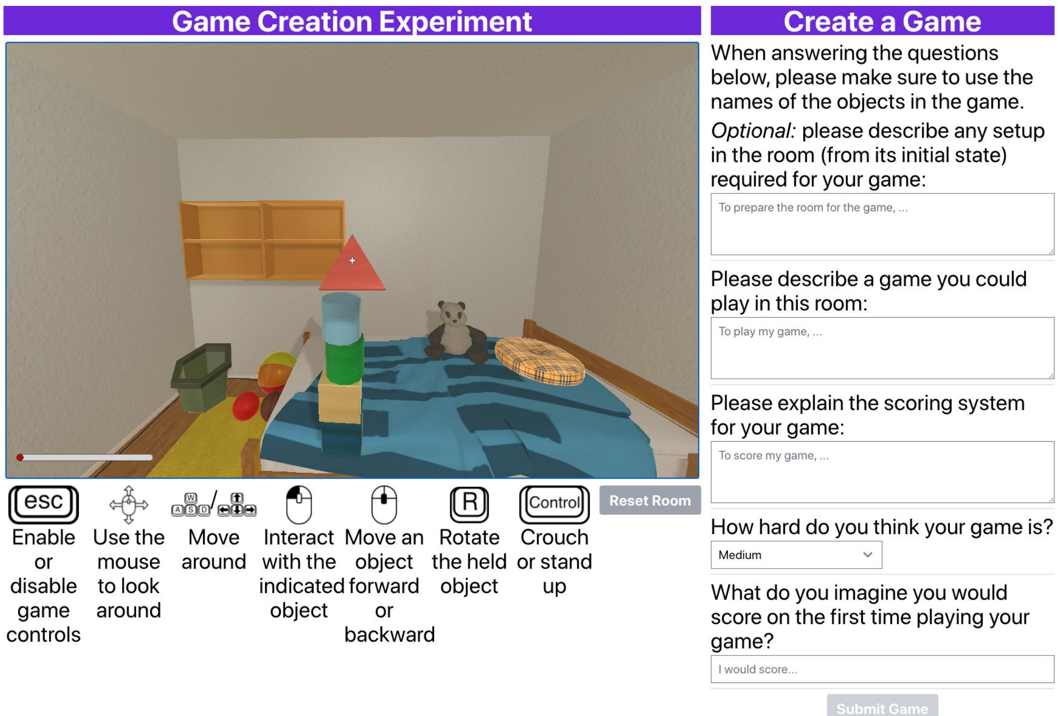


Figure ED-1: Online experiment interface. The main part of the screen presents the AI2-THOR-based experiment room. Below it, we depict the controls. To the right, we show the text prompts for creating a new game (fonts enlarged for visualization). Our experiment is accessible online here.

Table ED-1: Mixed model result summary

Measure	Fitness $\beta_{\texttt{fit}}$	Fitness Significance	$\mathbb{1}$[Matched] $\beta_{\texttt{matched}}$	$\mathbb{1}$[Matched] Significance	$\mathbb{1}$[Real] $\beta_{\texttt{real}}$	$\mathbb{1}$[Real] Significance
Understandable ↑	0.846	***	0.525	-	1.151	***
Fun to play ↑	0.396	**	0.629	*	1.059	***
Fun to watch ↑	0.191	-	0.641	*	0.912	***
Helpful† ↑	-0.189	*	0.349	*	0.232	-
Difficult $\genfrac{}{}{0pt}{}{\downarrow}{\uparrow}$	-0.588	***	0.363	-	-0.250	-
Creative ↑	-0.486	**	0.551	-	0.438	-
Human-like ↑	0.570	***	0.837	**	1.446	***

Fitness scores significantly predict several attributes, including understandability and human-likeness. Fitness scores show (statistically) significant positive effects on the understandability, fun to play, and human-likeness attributes, and significant negative effects on the helpfulness, difficulty and creativity questions. Accounting for the role of fitness, the `matched` group membership shows significant effects only the fun to play and watch, helpfulness, and human likeness questions. The `real` group shows significant effects on understandability, fun to play and watch, and human likeness. See Supplementary Table SI-3 for test statistics and P-values. *: $P < 0.05$, **: $P < 0.01$, ***: $P < 0.001$
†: The full measure description is "Helpful for interacting with the simulated environment."
In most measures, higher scores are better, indicated by the ↑, other than *Difficult* $\genfrac{}{}{0pt}{}{\downarrow}{\uparrow}$, in which 3 means "appropriately difficult", and scores below and above indicate too easy and too hard respectively.

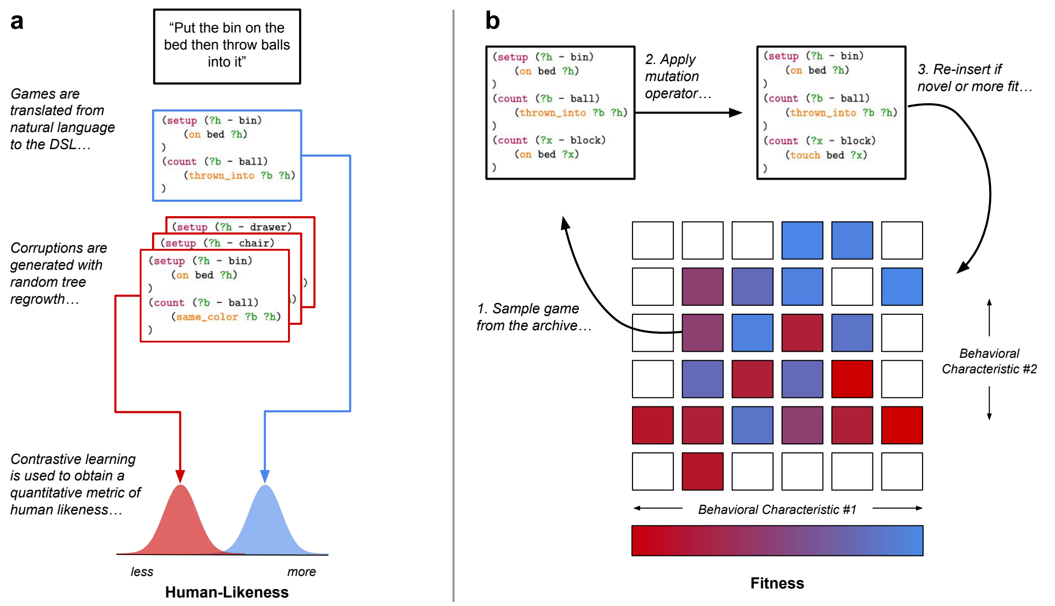


Figure ED-2: Parameter learning (left) and search (right) for the Goal Program Generator model. **Left:** We contrastively learn a quantitative measure of human likeness by maximizing the distance between human-generated exemplar games and a set of corruptions obtained through random tree regrowth. **Right:** This measure is then used as the basis for quality-diversity optimization through MAP-Elites. The algorithm maintains an archive of games that differ across phenotypic "behavioral characteristics." At each step, a game is randomly sampled from the archive (1), randomly mutated (2), and re-evaluated for both fitness and its position in the archive. It is added to the archive only if it would occupy a previously empty position in the archive or if it is more fit than the current occupant (3).

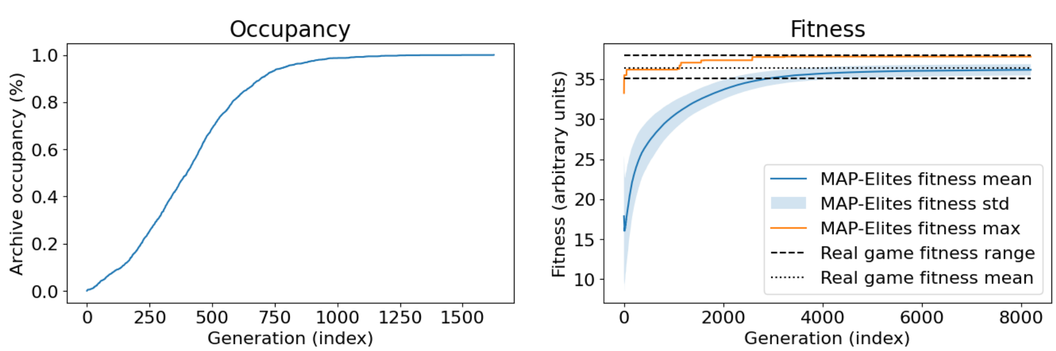


Figure ED-3: Our implementation of the Goal Program Generator model fills the archive quickly and finds examples with human-like fitness scores. **Left:** Our model rapidly finds exemplars for all archive cells (i.e. niches induced by our behavioral characteristics), reaching 50% occupancy after 400 generations (out of a total of 8192) and 95% occupancy after 794 generations—the archive is almost full 1/10th of the way through the search process. **Right:** Our model reaches human-like fitness scores. After only three generations, the fittest sample in the archive has a higher fitness score than at least one participant-created game. By the end of the search, the mean fitness in the archive is close to the mean fitness of human games.

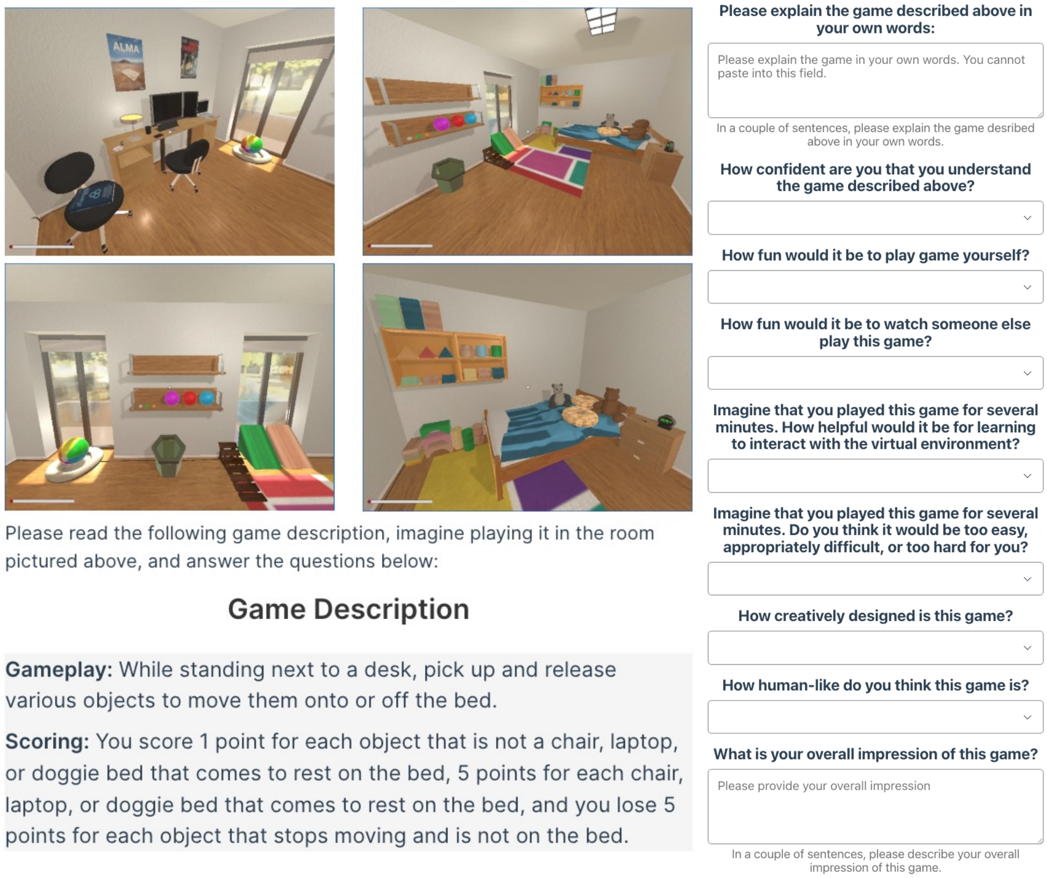


Figure ED-4: Human evaluations interface. For each game, participants viewed the same four images of the environment, followed by the GPT-4 back-translated description of the game (see Human evaluation methods for details. They then answered the two free-response and seven multiple-choice questions on the right. In the web-page based version, the questions appeared below the game description; they are presented side-by-side to save space.

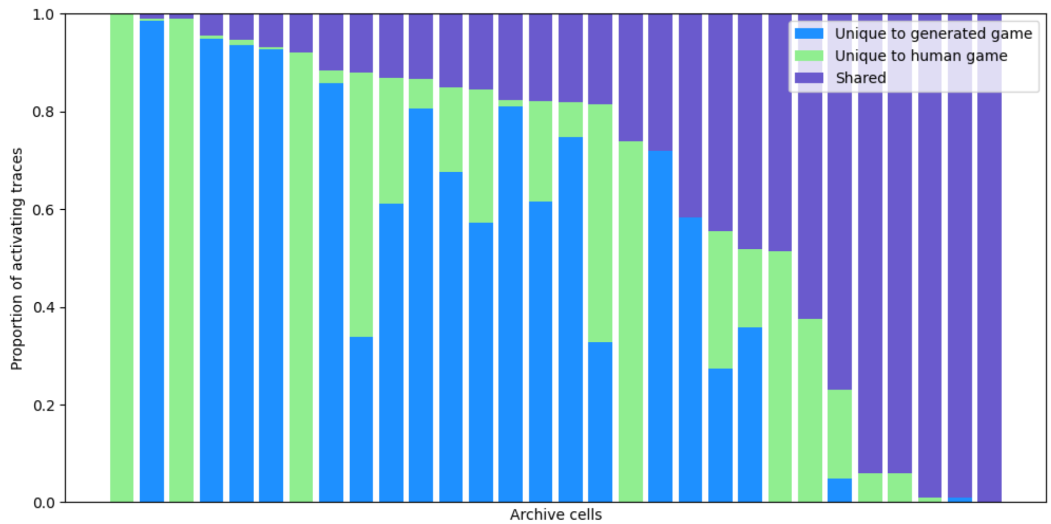


Figure ED-5: Proportion of human interactions activating only `matched` and `real` games in the same cell. Each bar corresponds to a pair of corresponding `matched` and `real` games. In each bar, we plot the proportion of relevant interactions (state-action traces) that are unique to the `matched` game (blue), unique to the `real` game (green), or shared across both (purple). A few games (with the bar mostly or entirely in purple) show high similarity between the corresponding games — under 25% (7/30) share more than half of their relevant interactions. Most games, however, show substantial differences between the sets of relevant interactions, with some showing a higher fraction unique to human games and others to matched model games. The average Jaccard similarity between the sets of relevant interactions for the `matched` and `real` game is only 0.347 and the median similarity is 0.180 (identical games would score 1.0, entirely dissimilar games 0).

A Pseudocode and program summary translation

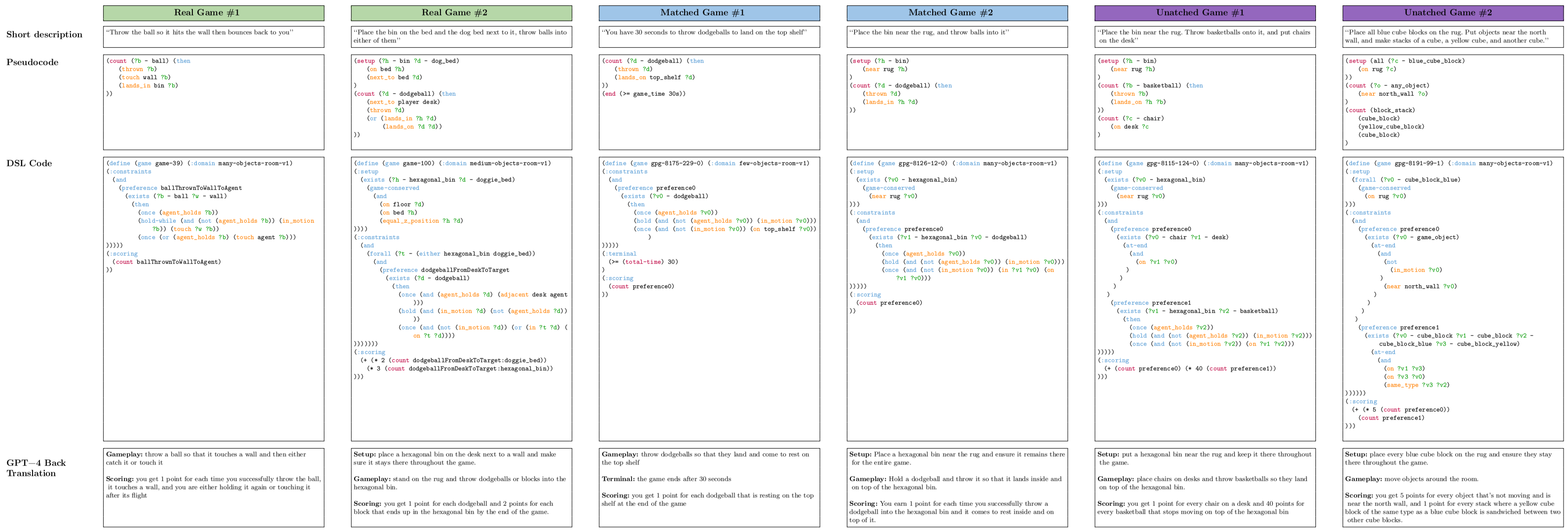


Figure SI-1: Translation of program pseudocode to our DSL. Each column depicts a single goal, starting from its summary description in natural language, the pseudocode program we used to represent it, its program representation in our domain-specific language (see Supplementary information I), and finally a GPT-4 automatic back-translation to natural language (see Supplementary information E). The first two columns(in green) highlight real participant-created games in our experiment mentioned in both Figure 1 and Figure 3. The two middle columns (in blue) describe model-generated goals "matching" the participant-created ones, and the last two columns (in purple) outline two "unmatched" novel model-generated goals.

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 [10] 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:

ngram: Features using our n-gran model.

1. `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?

2. `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?

3. `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?

4. `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?

5. `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?

play_trace_database: Features using our play trace database.

1. `predicate_found_in_data_prop [p]`: What proportion of predicates are satisfied at least once in our human play trace data?

2. `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?

defined_and_used: Features reflecting whether particular game components are defined, and features capturing whether defined components (such as variables, gameplay preferences, or objects in the setup) are then also used elsewhere.

1. `variables_used_all [b]`: Are all variables defined used at least once?

2. `variables_used_prop [p]`: What proportion of variables defined are used at least once?

3. `preferences_used_all [b]`: Are all preferences defined referenced at least once in terminal or scoring expressions?

4. preferences_used_prop [p]: What proportion of preferences defined are referenced at least once in terminal or scoring expressions?

5. setup_quantified_objects_used [p]: What proportion of object or types quantified as variables in the setup are also referenced in the gameplay preferences?

6. any_setup_objects_used [b]: Are any objects referenced in the setup also referenced in the gameplay preferences?

7. 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)

8. 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)

grammar_misuse: Features capturing various modes of grammar misuse—expressions that are grammatical under the DSL but ill-formed, poorly structured, or whose values cannot vary over gameplay.

1. adjacent_once_found [b]: Are there any cases where the once modal, which captures a single state, is used twice in a row?

2. adjacent_same_modal_found [b]: Are there any cases where the same modal is used twice in a row?

3. once_in_middle_of_pref_found [b]: Are there any cases where the once modal, which captures a single state, is in the middle of a sequence of modals?

4. pref_without_hold_found [b]: Are there any cases where a sequence of modals is specified with no temporally extended modal (hold or hold-while)?

5. 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)?

6. predicate_without_variables_or_agent [b]: Are there any predicates that do not reference any variables or the agent?

7. nested_logicals_found [b]: Are there any cases where a logical operator is nested inside the same logical operator (e.g., a negation of a negation, or a conjunction of a conjunction)?

8. identical_logical_children_found [b]: Are there any cases where a logical operator has two or more identical children?

9. redundant_expression_found [b]: Are there any cases where a logical expression over predicates is redundant (can be trivially simplified)?

10. unnecessary_expression_found [b]: Are there any cases where a logical expression over predicates is unnecessary (contradicts itself, or is trivially true)?

11. repeated_variables_found [b]: Are there any cases where the same variable is used twice in the same predicate?

12. repeated_variable_type_in_either [b]: Are there any cases where the same variable types is used twice in an either quantification?

scoring_grammar_misuse: Features capturing similar failure modes to the above category, but localized to the scoring and terminal sections of the DSL.

1. identical_scoring_children_found [b]: Are there any cases where a scoring arithmetic or logical expression has two or more identical children?

2. redundant_scoring_terminal_expression_found [b]: Are there any cases where a scoring or terminal expression is redundant (can be trivially simplified)?

3. unnecessary_scoring_terminal_expression_found [b]: Are there any cases where a scoring or terminal expression is unnecessary (contradicts itself, or is trivially true)?

4. total_score_non_positive [b]: Do the scoring rules of the game result in a non-positive score regardless of gameplay?

5. scoring_preferences_used_identically [b]: Do the scoring rules of the game treat all gameplay preferences identically?

6. `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)

game_element_disjointness: Features capturing whether particular game elements are disjoint—for example, gameplay preferences using disjoint sets of objects, or temporal modals using disjoint sets of variables.

1. `disjoint_preferences_found [b]`: Are there any preferences that quantify over disjoint sets of objects?

2. `disjoint_preferences_scoring_terminal_types [p]`: Do the preferences referenced in the scoring and terminal section quantify over disjoint sets of object types?

3. `disjoint_preferences_scoring_terminal_predicates [p]`: Do the preferences referenced in the scoring and terminal section use disjoint sets of predicates?

4. `disjoint_seq_funcs_found [b]`: Are there any cases where modals in a preference refer to disjoint sets of variables or objects?

5. `disjoint_at_end_found [b]`: Are there any cases where predicate expressions under an `at_end` refer to disjoint sets of variables or objects?

6. `disjoint_modal_predicates_found [b]`: Are there any cases where modals in a preference refer to disjoint sets of predicates?

7. `disjoint_modal_predicates_prop [p]`: What proportion of modals in a preference refer to disjoint sets of predicates?

counting: Features tracking node count or maximal depth in the four different DSL program sections.

1. `node_count_section [d]`: How many nodes are in the *section*, discretized to five bins with different thresholds for each section.

2. `max_depth_section [d]`: What is the maximal depth of the syntax tree in the *section*, discretized to five bins with different thresholds for each section.

pref_forall: Features capturing whether or not and how well the games use the `forall` over preferences syntax.

1. `pref_forall_used_correct [b]`: For the `forall` over preferences syntax, if it is used, is it used correctly in this game?

2. `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)

3. `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?

4. `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?

5. `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?

6. `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?

7. `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?

8. `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. The last three features all capture the same concept, whether or not a setup section exists. We surmise the diffused weights over them are a result of using weight decay (an L2 penalty) on the model weights:

1. `ast_ngram_full_n_5_score`
2. `ast_ngram_constraints_n_5_score`
3. `predicate_found_in_data_prop`
4. `ast_ngram_setup_n_5_score`
5. `variables_used_all`
6. `preferences_used_all`
7. `ast_ngram_scoring_n_5_score`
8. `max_depth_setup_0` (which indicates a setup section does not exist or is very minimal)
9. `node_count_setup_0` (which indicates a setup section does not exist or is very minimal)
10. `section_doesnt_exist_setup`

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

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

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 $\mathcal{D}^{+} \in \mathbb{R}^{N\times\times 1F}$, 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_{θ}
best model $\leftarrow$ None
best loss $\leftarrow \infty$
last improvement epoch $\leftarrow -1$
for epoch i **do**
  $\triangleright$ Assign negatives randomly to each positive
  Shuffle the first two dimensions of $\mathcal{D}^{-}$
  $\triangleright$ Reorder the positives in each epoch
  Shuffle the first dimension of $\mathcal{D}^{+}$
  for each batch **do**
    $X^{+} \leftarrow$ the next B positives $\triangleright$ X^{+}: $B \times 1 \times F$
    $X^{-} \leftarrow K$ sampled negatives for each positive $\triangleright$ X^{-}: $B \times K \times F$
    $X \leftarrow \mathrm{concat}(X^{+}, X^{-})$ $\triangleright$ X: $B \times (1 + K) \times F$
    $Y \leftarrow f_{\theta}(X)$ $\triangleright$ Y: $B \times (1 + K)$
    $L \leftarrow$ softmax loss(Y) $\triangleright$ L: scalar
    Take backward step on loss and optimizer step
  end for
  epoch validation losses $\leftarrow$ []
  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 $\leftarrow$ mean(epoch validation losses)
  if epoch loss $<$ best loss **then**
    best model $\leftarrow$ copy of $f_{\theta}(X)$
    best loss $\leftarrow$ epoch loss
    last improvement epoch $\leftarrow$ i
  else if $i -$ last improvement epoch $> P$ **then**
    break
  end if
end for
return best model

D MAP-Elites algorithm details

We use a set of 9 exemplar preferences as the basis for our MAP-Elites behavioral characteristics, detailed in Table SI-1. To score each game with respect to each exemplar preference, we count how many of the game's preferences are a close match to the exemplar. We explored matching preferences by edit distance (in string or syntax tree space) but discovered the edit distance is rather easily game-able by the model, producing semantically similar preferences with high edit distance from each other. Instead, we represent each exemplar preference as a binary feature vector, with features for which groups of predicates the preference uses (4 features: `agent_holds` or `in_motion`, `in`, `on`, and `adjacent` or `near` or `touch`), and for which object categories the preference uses (5 features: `balls`, `receptacles`, `blocks` or `buildings`, `furniture` or `room_features`, and `small_items` or `large_items` or the generic `game_object`). Preferences in each archive game are also represented using this feature space. A preference in an archive game is considered a match for an exemplar if it has an L1 distance of 0 or 1 in this space, and if a preference matches more than one exemplar, a match is randomly chosen. Exemplar preferences were iteratively chosen, starting from a seed preference (the first in Table SI-1), and then greedily adding additional exemplars from the preferences defined in participant-created games. At each step, the preference added was chosen to maximize the number of participant-created preferences that would be considered a match (distance of 0 or 1) from the exemplar set. In addition, we include two other behavioral characteristics, one capturing whether or not the game includes a setup component, and one capturing the total number of preferences (up to 4). In total, this set of behavioral characteristics allows for an archive size of 2000 games, of which 20 have one preference (matching one of the 9 exemplars or matching none of them, with and without a setup component), 110 have two preferences, 440 have three preferences, and 1430 have four preferences.

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 used by our 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. This "pseudo behavioral characteristic" doubles the size of the archive (from 2000 games to 4000 games), though we never evaluate any game from the half of the archive in which this feature is false.

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 DSL to natural language back-translation

In order to prepare games for human evaluation, we convert them from the DSL to natural language in a multi-stage process. In order to ensure consistency, we perform this back translation on both generated games and the real games (as opposed to using the original human-authored descriptions).

In the first stage of back-translation, a rule based system converts the DSL into templated language by concretely describing the definition of each predicate and grammatical rule. For instance, the expression `(once (and (agent_holds ?d) (adjacent ?p agent)))` is converted

Table SI-1: Exemplar preferences used as MAP-Elites behavioral characteristics.

Exemplar Preference	Description (GPT-4 back-translated)	Exemplar Features
(preference throwAttempt (exists (?b - dodgeball) (then (once (agent_holds ?b)) (hold (and (not (agent_holds ?b)) (in_motion ?b))) (once (not (in_motion ?b))))))	This preference is satisfied when: -first, the agent holds a dodgeball -next, the agent throws the dodgeball -finally, the dodgeball stops moving	Uses predicate agent_holds or in_motion Uses object category balls
(preference throwInBin (exists (?b - ball ?h - hexagonal_bin) (then (once (and (on rug agent) (agent_holds ?b))) (hold (and (not (agent_holds ?b)) (in_motion ?b))) (once (and (not (in_motion ?b)) (in ?h ?b))))))	This preference is satisfied when: -first, the agent is standing on the rug and holding a ball -next, the agent throws the ball -finally, the ball stops moving and is inside a hexagonal bin	Uses predicate agent_holds or in_motion Uses predicate in Uses predicate on Uses object category balls Uses object category receptacles Uses object category furniture or room_features
(preference ballThrownToBed (exists (?d - dodgeball) (then (once (and (agent_holds ?d) (adjacent desk agent))) (hold (and (not (agent_holds ?d)) (in_motion ?d))) (once (and (not (in_motion ?d)) (on bed ?d))))))	This preference is satisfied when: -first, the agent holds a dodgeball while standing next to a desk -next, the agent throws the dodgeball -finally, the dodgeball stops moving and is on the bed	Uses predicate agent_holds or in_motion Uses predicate on Uses predicate adjacent or near or touch Uses object category balls Uses object category furniture or room_features
(preference itemInClosedDrawerAtEnd (exists (?g - game_object) (at-end (and (in top_drawer ?g) (not (open top_drawer))))))	This preference is satisfied when: -at the end of the game, a game object is inside the top drawer and the top drawer is closed	Uses predicate in Uses object category receptacles Uses object category small_objects or large_objects or any_object Uses at_end
(preference watchOnShelf (exists (?w - watch ?s - shelf) (at-end (on ?s ?w))))	This preference is satisfied when: -at the end of the game, a watch is on a shelf	Uses predicate on Uses object category furniture or room_features Uses object category small_objects or large_objects or any_object Uses at_end
(preference gameBlockFound (exists (?l - block) (then (once (game_start)) (hold (not (exists (?b - building) (and (in ?b ?l) (is_setup_object ?b))))) (once (agent_holds ?l)))))	This preference is satisfied when: -first, the game begins -next, throughout the game, the block is not part of a building that is used in the setup -finally, the agent picks up the block	Uses predicate agent_holds or in_motion Uses predicate in Uses object category blocks or building
(preference matchingBuildingBuilt (exists (?b1 ?b2 - building) (at-end (and (is_setup_object ?b1) (not (is_setup_object ?b2)) (forall (?l1 ?l2 - block) (or (not (in ?b1 ?l1)) (not (in ?b1 ?l2)) (not (on ?l1 ?l2)) (exists (?l3 ?l4 - block) (and (in ?b2 ?l3) (in ?b2 ?l4) (on ?l3 ?l4) (same_type ?l1 ?l3) (same_type ?l2 ?l4)))))))))	This preference is satisfied when: -at the end of the game, one building is part of the setup while the other is not -and for any two blocks, neither is inside the building that is part of the setup -if one block is not on top of the other, then there must be two other blocks of the same types inside the building that is not part of the setup, with one of these blocks on top of the other	Uses predicate in Uses predicate on Uses object category blocks or building Uses at_end
(preference ballDroppedInBin (exists (?b - ball ?h - hexagonal_bin) (then (once (and (adjacent ?h agent) (agent_holds ?b))) (hold (and (in_motion ?b) (not (agent_holds ?b)))) (once (and (not (in_motion ?b)) (in ?h ?b))))))	This preference is satisfied when: -first, the agent is next to a hexagonal bin and is holding a ball -next, the agent throws the ball -finally, the ball stops moving and is inside the hexagonal bin	Uses predicate agent_holds or in_motion Uses predicate in Uses predicate adjacent or near or touch Uses object category balls Uses object category receptacles
(preference pillowMovedToRoomCenter (exists (?o - pillow) (then (once (and (agent_holds ?o))) (hold (and (in_motion ?o) (not (agent_holds ?o)))) (once (and (not (in_motion ?o)) (near room_center ?o) (exists (?o1 ?o2 ?o3 - game_object) (and (same_color ?o1 pink) (near room_center ?o1) (same_color ?o2 blue) (near room_center ?o2) (same_color ?o3 brown) (near room_center ?o3))))))))	This preference is satisfied when: -first, the agent picks up a pillow -next, the agent throws the pillow and it is no longer being held by the agent -finally, the pillow stops moving near the center of the room, and there are three other objects near the center of the room as well: one that is pink, one that is blue, and one that is brown	Uses predicate agent_holds or in_motion Uses predicate adjacent or near or touch Uses object category furniture or room_features Uses object category small_objects or large_objects or any_object

to “there is a state where (the agent is holding `?d`) and (`?p` is adjacent to agent).” Each of the game’s setup conditions, preferences, terminal conditions, and scoring rules are rendered in this form, which also includes the mapping from variable names (e.g. `?d`) to the types of objects that can occupy the variable (e.g. “dodgeball”). An example of a game’s preferences described in this form is presented below:

```
The preferences of the game are:

-----Preference 1-----
The variables required by this preference are:
-?p of type pyramid_block
-?d of type dodgeball
-?h of type hexagonal_bin

This preference is satisfied when:
- first, there is a state where (the agent is holding ?d) and (?p is adjacent to agent)
- next, there is a sequence of one or more states where (it's not the case that the agent is holding ?d) and (?d is in motion)
- finally, there is a state where (it's not the case that ?d is in motion) and (?d is inside of ?h)

-----Preference 2-----
The variables required by this preference are:
-?b of type building
-?l of type cube_block
-?f of type flat_block

This preference is satisfied when:
- in the final game state, (?f is used in the setup), (?f is inside of ?b), and (?l is inside of ?b)
```

Next, we use the GPT-4 large language model (LLM) [63] to simplify the templated description into a more naturalistic form (specifically `gpt-4-1106-preview`). The objective of this stage is to re-write any unclear formulations generated by the initial procedure and to replace abstract variable names with their actual referents. We convert each section of the game separately, using a similar prompt for each. The prompt begins with the following message:

“*Your task is to convert a templated description of a game’s* `<setup / rules / terminal conditions / scoring conditions>` *into a natural language description. Do not change the content of the template, but you may rewrite and reorder the information in any way you think is necessary in order for a human to understand it. Use simple language and verbs that would be familiar to a human who has never played this game before.*”

We then include 10 examples of this kind of translation taken from the set of human games not used in our experiments. An example of the same preferences in this simplified form is presented below:

```
The preferences of the game are:

-----Preference 1-----
This preference is satisfied when:
-first, the agent holds a dodgeball while standing next to a pyramid block
-next, the agent throws the dodgeball
-finally, the dodgeball lands inside a hexagonal bin and stops moving

-----Preference 2-----
This preference is satisfied when:
-at the end of the game, a flat block is used in the setup of a building and both a cube block and the flat block are inside the
       building
```

Finally, we use the LLM again to collect the separate descriptions of each section into one a single block, further simplifying the language and expressions. The prompt is similar to that used in the previous stage, and again is followed by 10 selected examples:

“*Your task is to combine and simplify the description of a game’s rules. Do not change the content of the rules by either adding or removing information, but you may rewrite and reorder the information in any way you think is necessary in order for a human to understand it. Use simple language and verbs that would be familiar to a human who has never played this game before. DO describe preferences carefully, such that a player reading the description can easily play the game. DO NOT include explicit references to a game’s preferences (i.e. "Preference 1" or "Preference 2"). DO NOT include descriptions of setup or terminal conditions if they do not appear in the game.*”

Examples of complete translations are available in Figure 4 and Figure 5.

F Model sample and real game edit distance similarity

We analyze our model's results through the lens of the MAP-Elites behavioral characteristics we use, as they functionally define diversity for our model (see MAP-Elites methods and Supplementary information D for additional details). When we present results from the model, such as in Figure 4, we present `matched` model samples alongside the `real` participant-created games that MAP-Elites maps to the same archive cell. However, there are other ways to determine similarity in a high-dimensional space, such as the one our program representations occupy. We wish to offer additional evidence for the degree of distinctiveness of the model-generated samples. One reasonable approach to similarity is an edit distance: for simplicity, we use the string (Levenshtein edit distance). Each program's syntax tree is rendered as a string. We then remove the preamble that includes the game name, and combine consecutive white space tokens to a single space. For each of the six model-generated games we present in figures Figure 4 and Figure 5, we compute the edit distance to all 98 real participant-created games, and present the real game with the smallest edit distance to each of these samples.

In one case (Matched Game #1), this is the same participant game that occupies the same archive cell. In the other two matched games, the nearest game is different. In both cases, the nearest participant-created game retains some high-level similarity, but with different gameplay objectives than the ones our model proposed. We also present the nearest matches for the closest `unmatched` model-created games in figure Figure 5. Here we find the nearest model games further away, both in edit distance and conceptually in the goals the programs represent. We take this as further evidence our model generates creative samples, meaningfully different from participant-created ones.

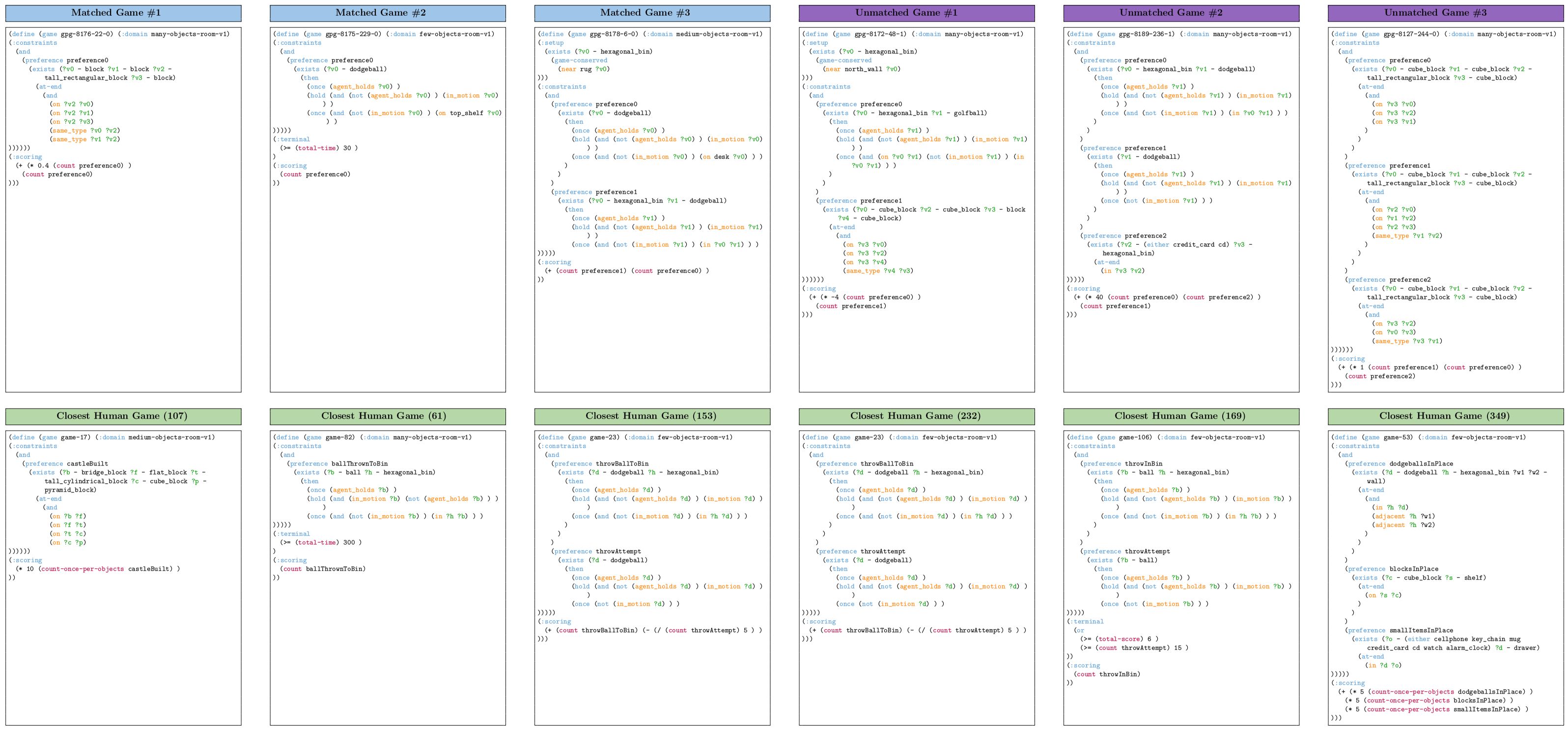


Figure SI-2: Edit distance nearest real programs to selected model samples. For each model-generated sample presented in Figure 4 (first three columns with blue headers) and Figure 5 (last three columns with purple headers), we present below it the most similar real participant-created game, as measured by the Levenshtein edit distance.

G Human evaluations data analysis

G.1 Detailed human evaluation results

Table SI-2: Human evaluation result summary

Attribute	Mean score by category `Real`	`Matched`	`Unmatched`	**`Real` vs. `Matched`** U-stat, P-value	**`Real` vs. `Unmatched`** U-stat, P-value	**`Matched` vs. `Unmatched`** U-stat, P-value
Understandable	3.943	3.923	3.331	45088.0 , $P = 0.906$	55921.5 , $P < 1e{-5}^{***}$	55846.0 , $P < 1e{-5}^{***}$
Fun to play	2.522	2.430	2.068	46752.5 , $P = 0.352$	54040.5 , $P < 1e{-5}^{***}$	52539.5 , $P < 1e{-3}^{***}$
Fun to watch	2.385	2.313	2.024	46169.0 , $P = 0.519$	51636.5 , $P < 1e{-3}^{***}$	50515.0 , $P = 0.001^{**}$
Helpful†	2.997	2.987	2.840	44802.0 , $P = 0.982$	47372.5 , $P = 0.078$	47559.0 , $P = 0.075$
Difficult	2.582	2.660	2.676	42921.5 , $P = 0.326$	42218.5 , $P = 0.419$	44081.0 , $P = 0.947$
Creative	2.318	2.213	2.143	47036.0 , $P = 0.282$	48286.0 , $P = 0.025^{*}$	46615.0 , $P = 0.182$
Human-like	2.813	2.670	2.119	47698.0 , $P = 0.167$	58679.0 , $P < 1e{-5}^{***}$	55434.5 , $P < 1e{-5}^{***}$

Evaluators don't distinguish between participant-created `real` and `matched` model games, but do distinguish `unmatched` games from both. Participants responded to seven Likert questions on a 5-point scale, one for each attribute in the first column (see Human evaluation methods for additional details). We report the Mann-Whitney U test [57] for differences in outcomes, appropriate for ordinal data. *: $P < 0.05$, **: $P < 0.01$, ***: $P < 0.001$
†: The full measure description is "Helpful for interacting with the simulated environment."

G.2 Mixed-effect model analyses

In the Human evaluations section, we briefly describe the mixed effect model we fit to analyze our human evaluation results and analyze the learned regression weights for the fitness function. Here, we build on this analysis to examine the extent to which accounting for the mediating effect of fitness scores, changes our previous observations regarding the differences between groups. Using the `unmatched` group as the baseline, the regression coefficients β_{matched} and β_{real} quantify these differences for each measure. We find statistically significant differences for the `matched` group (i.e. $\beta_{\text{matched}} > 0$) for ratings of fun to play, fun to watch, helpfulness, and human likeness. Similarly, we observe statistically significant differences ($\beta_{\text{real}} > 0$) for ratings of understandability, fun to play and watch, and human likeness. Finally, using the marginal (least-squares) means method[50], we directly compare the `matched` and `real` categories and again find no statistically significant differences (see Human evaluation methods for additional details and Supplementary Table SI-4 below for the full results).

Table SI-3: Mixed model result summary

Attribute	**Fitness** β_{fitness}	Z	P-value	`Matched` β_{matched}	Z	P-value	`Real` β_{real}	Z	P-value
Understandable	0.846	5.625	$P < 1e{-5}^{***}$	0.525	1.766	$P = 0.078$	1.151	4.036	$P < 1e{-3}^{***}$
Fun to play	0.396	2.936	$P = 0.003^{**}$	0.629	2.298	$P = 0.022^{*}$	1.059	4.021	$P < 1e{-3}^{***}$
Fun to watch	0.191	1.469	$P = 0.142$	0.641	2.414	$P = 0.016^{*}$	0.912	3.547	$P < 1e{-3}^{***}$
Helpful†	-0.189	-2.163	$P = 0.031^{*}$	0.349	2.048	$P = 0.041^{*}$	0.232	1.441	$P = 0.15$
Difficult	-0.588	-3.443	$P < 1e{-3}^{***}$	0.363	1.029	$P = 0.304$	-0.250	-0.740	$P = 0.46$
Creative	-0.486	-3.191	$P = 0.001^{**}$	0.551	1.776	$P = 0.076$	0.438	1.467	$P = 0.142$
Human-like	0.570	4.316	$P < 1e{-3}^{***}$	0.837	3.128	$P = 0.002^{**}$	1.446	5.597	$P < 1e{-5}^{***}$

Fitness scores significantly predict several attributes, including understandability and human-likeness. Fitness scores show (statistically) significant positive effects on the understandability, fun to play, and human-likeness attributes, and significant negative effects on the difficulty and creativity questions. Accounting for the role of fitness, the `matched` group membership shows a significant effect only on human likeness. The `real` group shows significant effects on understandability, fun to play to watch, and human likeness. *: $P < 0.05$, **: $P < 0.01$, ***: $P < 0.001$
†: The full measure description is "Helpful for interacting with the simulated environment."

G.3 Marginal Means Analysis

Table SI-4: Mixed model marginal means comparison summary

Attribute	Real − Matched Diff	Real − Matched Z	Real − Matched P-value	Real − Unmatched Diff	Real − Unmatched Z	Real − Unmatched P-value	Matched − Unmatched Diff	Matched − Unmatched Z	Matched − Unmatched P-value
Understandable ↑	0.626	2.055	$P = 0.100$	1.151	4.036	$P < 1e{-3}^{***}$	0.525	1.766	$P = 0.181$
Fun to play ↑	0.430	1.577	$P = 0.256$	1.059	4.021	$P < 1e{-3}^{***}$	0.629	2.298	$P = 0.056$
Fun to watch ↑	0.271	1.025	$P = 0.561$	0.912	3.547	$P = 0.001^{**}$	0.641	2.414	$P = 0.042^{*}$
Helpful† ↑	-0.117	-0.701	$P = 0.763$	0.232	1.441	$P = 0.32$	0.349	2.048	$P = 0.101$
Difficult ↓↑	-0.613	-1.725	$P = 0.196$	-0.250	-0.740	$P = 0.74$	0.363	1.029	$P = 0.559$
Creative ↑	-0.113	-0.364	$P = 0.93$	0.438	1.467	$P = 0.307$	0.551	1.776	$P = 0.178$
Human-like ↑	0.609	2.299	$P = 0.056$	1.446	5.597	$P < 1e{-5}^{***}$	0.837	3.128	$P = 0.005^{**}$

We use the method of marginal (least-squares) means [50] to estimate the mean score for each attribute in each category, holding fitness constant. None of the comparisons between the `real` and `matched` groups are significant, and several (though not all) of the previously significant comparisons remain significant. *: $P < 0.05$, **: $P < 0.01$, ***: $P < 0.001$
†: The full measure description is "Helpful for interacting with the simulated environment."

In most measures, higher scores are better, indicated by the ↑, other than *Difficult* ↓↑, in which 3 means "appropriately difficult", and scores below and above indicate too easy and too hard respectively.

G.4 Matched-real game similarity analysis

To functionally measure similarity, we leverage the fact that our goal programs are interpretable and automatically evaluate them on all gameplay interactions generated by participants in our first experiment. For each `matched` game and its corresponding `real` counterpart, we measure the number of interactions that fulfill a gameplay element in only the `matched` game, only the `real` game, or both. While some pairs of games have their elements fulfilled by the same interactions (suggesting functional similarity), most pairs are not — under 25% (7/30) share more than half of their relevant interactions. Furthermore, the average Jaccard similarity between the sets of relevant interactions for the `matched` and `real` game is only 0.347 and the median similarity is 0.180 (identical games would score 1.0, entirely dissimilar games 0; and see summary in Figure ED-5 and methodological details in Sample similarity comparison methods).

H Model ablations

H.1 Common sense ablation

The domain-specific language we use is underconstrained—many expressions that are grammatical either make no sense at all (e.g., checking a bin is in a ball, rather than a ball in a bin) or violate intuitive physical common sense (e.g., creating a game stacking balls, as opposed to stacking blocks). We primarily operationalize the concept of physical common sense using two of our fitness features, discussed in Fitness function methods and Supplementary information B. Both use a dataset of interaction traces (see Dataset collection methods) to estimate the feasibility of predicate role-filler expressions, by computing the proportion of predicate expressions (and the object types they operate over) that have appeared at least once over the set of interactions of users with the environment. While this condition is not necessary (as it is unlikely experiment participants explored every feasible configuration of objects in the environment), it is sufficient to determine feasibility and, therefore, serves as a good proxy for intuitive common sense. The first feature operates over individual predicates, e.g. estimating that `(on desk ball)` is more likely than `(on desk bed)`. The second feature operates on logical expressions over predicates, and might help catch contradictory predicates that are independently feasible, such as `(and (on desk ball )(on bed ball))`, that is, the ball might feasibly be on the desk or on the bed, but not on both.

We know that these features are helpful for our model, as the individual predicate version of these features has the third highest weight of all features that predict real human-generated games (see Supplementary information B.1 for details). To further evaluate the importance of these features, we fit a version of our fitness model that has no access to them, and use it as the objective for our model. Unsurprisingly, when we evaluate samples from this ablated model on the full fitness function (with the interaction trace features), they have statistically significantly lower fitness scores than the samples from the full model (matched-pairs t-test matching by archive cells, $t = -32.66, P < 1\text{e}{-10}$). To offer a more fair comparison, we use the full "reward machine" and dataset of play traces. We assign a binary score to each game from the full and ablated models, 1 if each game component (gameplay preferences and the setup section (if one exists)) is satisfied at least once over the dataset, either in the same trace or in different traces. If at least one game component is never satisfied, we assign a score of 0. We find that 1515 (75.75%) of the games in the full model score 1, while only 584 (29.20%) in the ablated model do. This difference is, as expected, also statistically significant (matched-pairs t-test, $t = -33.29, P < 1\text{e}{-10}$). We conclude that intuitive physical common sense is helpful to our model, as allowing our model to approximate the physical sensibility of predicates helps the model generate games with components that have been satisfied by our participants.

H.2 Compositionality ablation

Evaluating the role of compositionality in our model is challenging as the model operates on a domain-specific language that is inherently highly compositional. Given the nature of program representations, it's difficult to imagine a non-compositional counterfactual DSL to compare to — so we cannot compare to an entirely non-compositional model. Instead, we ablate by varying how compositional we allow our MAP-Elites mutation operators to be. The primary operator embodying compositionality is the crossover operator, which samples two programs from the MAP-Elites archive, randomly selects exchangeable sub-trees from both programs, and creates new candidates with these trees swapped between the programs. We also implemented several custom operators (beyond the evolutionary programming staples of mutation, insertion, deletion, and crossover). Many of these implement targeted variations of crossover that we considered to be plausible higher-level changes a person might make to a game they are creating, such as sampling a preference from another game and then changing the preference's initial or terminal conditions. We report two ablations, one ("No Custom Ops") where we omit the custom operators we implemented (keeping only regrowth, insertion, deletion, and crossover), and a second ("No Custom Ops, No Crossover") where we also remove the crossover operation. We keep all other model details identical, crucially both the set of behavioral characteristics and fitness function, allowing us to directly compare the fitness values of games in the archives in the ablated models.

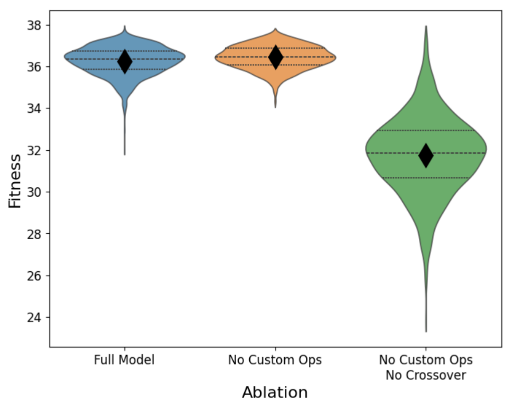
We visualize the results of these in Figure SI-3. While removing our custom operators appears to slightly *increase* the mean fitness of exemplars (Figure SI-3a, orange), removing the crossover operation drastically decreases the fitness of games in that model's archive (Figure SI-3a, green). This provides evidence that allowing our search procedure to take advantage of the compositionality in our domain is greatly beneficial in generating high-quality samples across our archive. If our custom operators do not increase mean fitness, what impact do they have? To quantify this question, we evaluated samples from the ablated models through the full "reward machine." For each sample, we counted how many of the participant interaction traces saw the participant fulfilling one or more gameplay elements from the sample. In other words, how many participants (unwittingly or otherwise) fulfilled at least part of the model-generated goal program? We find that the custom operators help increase this number – samples generated from our full model show the highest number of relevant traces (Figure SI-3b, blue). This could have two interpretations: one is that the custom operators push more goals toward higher feasibility. Another is that this behavior is a form of mode-seeking that helps the model generate goal programs that capture more common behaviors, as opposed to more meaningful variability. In all, we take this as an effect that crossover is crucial to generating fit samples across our MAP-Elites archive, with some cost to diversity which our custom operators help reduce.

H.3 Coherence ablation

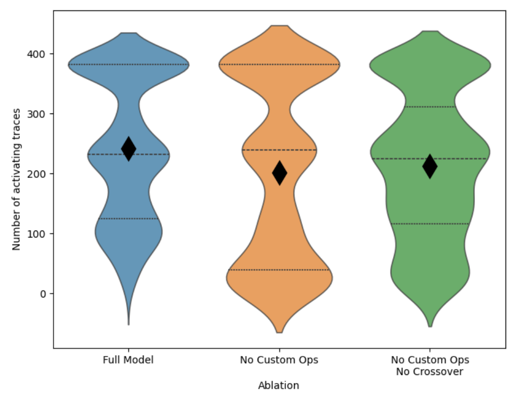
As we iterated on earlier versions of our model, we discovered some 'softer', higher-level issues repeatedly surfacing in model-generated goal programs. Even after implementing features that helped the model avoid some types of low-level mistakes (such as instantiating variables or preferences and never referencing them), and introducing approximations to intuitive physical common sense (discussed above), some aspects of the generated games remained incoherent. A lower-level example might be disjointedness in the arguments of temporal modals. Consider, for instance, a preference whose modals translate to natural language as "start with a state where the agent holds a ball, then find a collection of states where a block is on the bed, and finish with a state where the bin is upside down". The awkwardness in explaining this perfectly grammatical preference (program below) is that each modal (`(once ..)`, `(hold ...)`) refers to a distinct set of objects, and so it feels unnatural to specify a sequential temporal preference over them.

```
(preference preference0
  (exists (?v0 – hexagonal_bin ?v1 – ball ?v2 – block)
    (then
      (once (agent_holds ?v1))
      (hold (on desk ?v2))
      (once (object_orientation ?v0 upside_down))
)))
```

We observed similar, higher-level issues regarding coherence between different gameplay preferences (do they use the same objects and predicates, or distinct sets?). Specifically, we observed cases where game scoring conditions and ending conditions have nothing to do with each other. A game might specify that it ends after a ball has been thrown five times, with points scored for every block placed on the desk. There is nothing wrong per se with this specification, but it feels unnatural—we would expect either the ball-throwing to contribute to scoring, or the block-stacking to allow the game to end, or both. We wrote a collection of fitness features to try to capture occurrences of such incoherence (see `game_element_disjointness` in Supplementary information B). We have some indication that these features are important from observing that our fitness model assigns one of them the third-largest negative weight (predictive of corrupted, negative games). To ablate the effect of this feature group, we perform an ablation similar to the common sense ablation reported above—we fit a fitness model without these features and use it to guide our MAP-Elites search. As a first sanity check, we compute fitness scores under the full fitness model for games generated by the ablated model. We find that scores in the ablated model are consistently lower (matched-pairs t-test, $t = -26.99, P < 1\mathrm{e}{-10}$), indicating that without access to these features, our model would generate programs that violate these coherence considerations. We also evaluate games from this ablated model using the 'reward machine' and play traces dataset, as we did above. As before, 1515 (75.75%) of games in the full model have every component satisfied, while only 1224 (61.2%) in the ablated model do. This difference is also statistically significant (matched-pairs t-test, $t = -9.73, P < 1\mathrm{e}{-10}$).



(a) Removing crossover drastically lowers fitness values. We plot, for each game generated by a model, its fitness score under the full fitness function. **Left:** The distribution of fitness scores in our full model. **Middle:** Removing the custom operators has little effect on the distribution of fitness scores. **Right:** Removing crossover drastically lowers the fitness scores of model samples.



(b) Removing custom operators lowers mean trace coverage; removing crossover undoes some of the effect. We measure, for each game generated by a model, how many participant interaction traces fulfill at least one gameplay element. **Left:** Of the ablations reported, our full model shows the highest number of active traces. **Middle:** removing our custom mutation operators lowers the mean number of active traces. **Right:** Removing crossover as well undoes some of the effect of removing the custom operators.

Figure SI-3: The crossover operator helps generate fit goals, while the custom operators help generate solutions with higher trace coverage. Left: removing the custom operators does hurt mean fitness scores; removing the crossover operator does. **Right:** removing the custom operators leads the model to generate samples covering fewer participant interaction traces on average. This could be evidence of lower feasibility (more samples in the “no custom ops” model are active in barely a few traces) or of mode seeking (more samples in the full model are active in a very high number of traces).

I Full domain-specific language description

J 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.

J.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 Appendix K.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 (⟨*variable-list*⟩) ⟨*setup*⟩)
| (forall (⟨*variable-list*⟩) ⟨*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*⟩
| (⟨*variable-list*⟩) ⟨*super-predicate*⟩)
| (⟨*variable-list*⟩) ⟨*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*⟩

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

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

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

⟨*object-name*⟩ ::= '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'

J.2 Gameplay Preferences

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

The $\langle$*constraints*$\rangle$ elements expands into one or more preference definitions, which are defined using the $\langle$*pref-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 Appendix L 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 Appendix J.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 $\langle$*hold-while*$\rangle$, which offers a variation of the $\langle$*hold*$\rangle$ operator. A $\langle$*hold-while*$\rangle$ receives at least two predicates. The first acts the same as predicate in a $\langle$*hold*$\rangle$ operator. The second (and third, and any subsequent ones), must hold true for at least state while the first predicate holds, and must occur in the order specified. In the example above, if we substitute `(hold (pred_b))` for `(hold-while (pred_b)(pred_d)(pred_e))`, we now expect that in addition to 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 Appendix J.1 above.

J.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 Appendix J.4 below.

J.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*⟩

K Non-Grammar Definitions

K.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>) [84 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>) [312 references] ; Is the object in motion?
(is_setup_object <arg1>) [13 references] ; Is this the object of the same type referenced in the setup?
(near <arg1> <arg2>) [63 references] ; Is the second object near the first object? [implemented as distance below some threshold]
(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>) [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_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?
```

K.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>) [50 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).
```

K.3 Types

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

```
game_object [33 references] ; Parent type of all objects
agent* [100 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* [45 references]
desktop [6 references]
main_light_switch* [3 references] ; The main light switch on the wall
side_table* [6 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 [11 references]
triangular_ramp_green [1 reference]
triangular_ramp_tan [0 references]
---------- (* \textbf{Receptacles} *) ----------
doggie_bed [27 references]
hexagonal_bin [122 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* [15 references] ; The door out of the room
floor* [26 references]
mirror* [0 references]
poster* [0 references]
room_center* [33 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]
```

L Modal Definitions in Linear Temporal Logic

L.1 Linear Temporal Logic definitions

We offer a mapping between the temporal sequence functions defined in Appendix J.2 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.

L.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 $\psi\ \alpha\ \beta \cdots \nu$) := $(\psi M \alpha) X (\psi M \beta) X \cdots X (\psi M \nu) X \psi U \cdots$ where the last $\psi U \cdots$ allows for additional states satisfying 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)) (* \color{blue} α*)
    (hold-while
        (and (not (agent_holds ?b)) (in_motion ?b)) (* \color{blue} β *)
        (touch ?b ?r) (* \color{blue} γ*)
    )
    (once (and (in ?h ?b) (not (in_motion ?b)))) (* \color{blue} δ*)
)
```

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

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

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