Languages of Play

Towards semantic foundations for game interfaces

Chris Martens
North Carolina State University
Raleigh, NC
martens@csc.ncsu.edu

Matthew A. Hammer
University of Colorado Boulder
Boulder, CO
matthew.hammer@colorado.edu

ABSTRACT

Formal models of games help us account for and predict behavior, leading to more robust and innovative designs. While the games research community has proposed many formalisms for both the "game half" (game models, game description languages) and the "human half" (player modeling) of a game experience, little attention has been paid to the *interface* between the two, particularly where it concerns the player expressing her intent toward the game. We describe an analytical and computational toolbox based on programming language theory to examine the phenomenon sitting between control schemes and game rules, which we identify as a distinct *player intent language* for each game.

KEYWORDS

game interfaces, programming languages, formal methods

1 INTRODUCTION

To study how players interact with games, we examine both the rules of the underlying system and the choices made by the player. The field of player modeling has identified the value in constructing models of player cognition： while a game as a self-contained entity can allow us to learn about its mechanics and properties as a formal system, we cannot understand the *dynamics* of that system unless we also account for the human half of the equation. Meanwhile, Crawford [6] identifies the necessity of looking at the complete information loop created between a player and a digital game, defining *interactivity* in games as their ability to carry out a conversation with a player, including listening, processing, and responding, identifying the importance of all three to the overall experience.

Given this understanding of games-as-conversation, we should expect to discover something like a *language* through which games and players converse. In Figure 1, we illustrate the game-player loop as a process which includes an *interface* constituting such a language. The Game Ontology Project [22] describes game interfaces as follows：

> *The interface is where the player and game meet, the mapping between the embodied reactions ofthe player and the manipulation of game entities. It refers to both how theplayer interacts with thegame and how the game communicates to the player.*

The first part, *how the player interacts with thegame,* is called the *input*, which is further subdivided into *input device* and *input method*. Input devices are hardware controllers (mice, keyboards, joysticks, etc.) and input *methods* start to brush the surface of something more semantic： they include choices about *locus of manipulation* (which game entities can the player control?) and direct versus indirect action, such as selecting an action from a menu of options (indirect) versus pressing an arrow key to move an avatar (direct).

However, any close look at interactive fiction, recent mobile games, or rhythm games (just to name a few examples) will reveal that design choices for input methods have much more variety and possibility than these two dimensions. In this paper, we propose a framework to support analyzing and exploring that design space.

Our first step is to refine input *methods* to input *languages*： we are ultimately asking, how can a player communicate their intent, and how does a digital game recognize this intent? So, in linguistic terms, the "phonemes" of such a language are hardware controls such as button presses and joystick movement. Then, the syntax and semantics are defined by each game individually, depending on what meaning they give to each control input. This language defines the verbs of the game, which may include moving, selecting inventory items, examining world items, applying or using items, entering rooms, and combat actions. (Note that such a language is also distinct from a game's *mechanics*： mechanics include system behavior which is out of the player's control, such as falling with gravity, non-player character actions, and other autonomous behavior.) This language is both *afforded* by the game designer—she must communicate to players which verbs are available—and *constrained* by her—she may declare certain expressions invalid.

Since the constraints on such a language are wholly determined by a piece of software (the game interface), we argue that it has more in common with a *programming language*[1] (PL) than a natural language. Accordingly, each game in some sense *defines its own programming language*. In a slogan, we could term this project *games as programming languages*. Specifically, we propose *player intent languages* as a PL-inspired framework for designing playergame interfaces.

This analogy opens up a whole field of methodology to try applying to games. The PL research community has a long and deep history of assigning mathematically formal semantics to languages and analyzing those semantics. As games researchers become more interested in the emergent consequences of the systems they assemble, the tools of PL theory have a lot to offer. For example, PL theory provides an account of *compositionality,* i.e. how fragments of expression fit together to form higher-level meanings. In games, this translates into being able to understand player *skills* or strategies as compositions of player actions, which we demonstrate in

[1]We define programming languages broadly as formal languages whose meaning is fully grounded in a computational system.

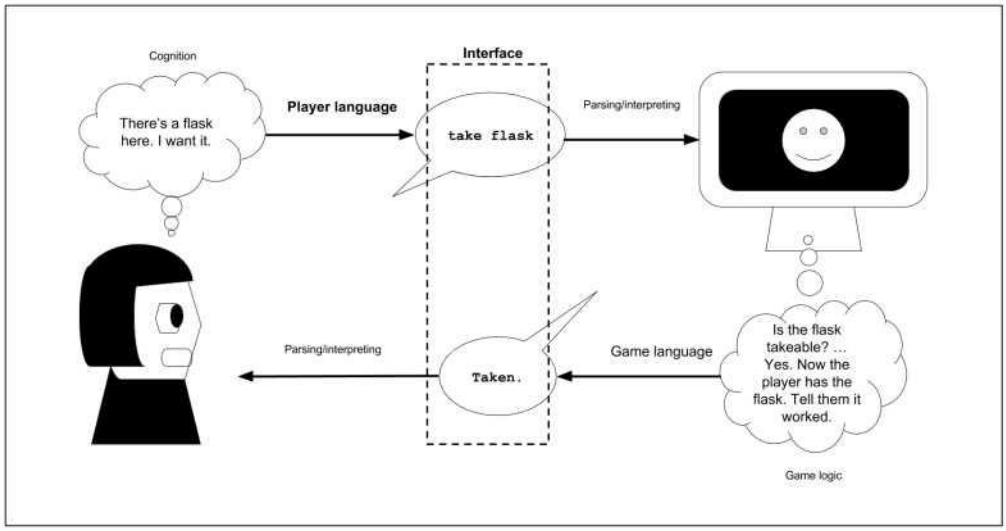


Figure 1: A process diagram of a game loop: player and game conversation as it relates to language, interface, and cognition.

this paper by using a formalized input language as a kind of "player AI scripting language"

Furthermore, by considering a player's language of expression as an object of study in its own right, we center them as a co-designer of the experience afforded by a game. When we treat a player's interactions as not simply an arbitrary sequence of button presses that advances and reveals the designer's intent, but instead as its own 击 stinct *voice* that a game system must listen and respond to, we enable the player to *co-create* with the system, potentially developing deeper systems understanding and emotional investment.

In this paper, we propose *player intent languages,* a programming languages-based approach to designing player-game interfaces as formal objects. In the remainder of the paper, we tour this approach through concrete examples. Specifically, we consider a simple game design space and make points in this space precise by introducing the components of a programming language： abstract syntax (Section 4), type system (Section 6), and operational semantics (Section 5). For each, we give a corresponding concept in the game world. By grounding these game concepts in analogous programming language concepts, we gain powerful PL reasoning tools and design methodologies to benefit the game design process.

We demonstrate the payoff of this line of thought by extending the metaphor with *play traces* as *straight-line programs* (Section 7), and *player skills* as *general programs* (e.g., programs with parameters, branching and looping) (Section 8). These structures give semantic logs and general strategies, respectively, for accomplishing a task in the game world. The framework of player intention languages gives rise to further research directions, which we briefly outline and discuss before concluding (Sections 9 and 10).

2 RELATED WORK

Cardona-Rivera and Young [4] detailed a conceptual framework following the slogan *games as conversation*, grounding the communicative strategies of games in cognitive science for human-to-human conversational understanding, such as Grice's maxims [9]. They offer a linguistic and semiotic approach to understanding how a game communicates affordances (possibilities for action) to a player. For an account of the game's half of the equation, which includes the visual, textual, and audio feedback mechanisms intended to be processed by the player, this application of linguistics, psychology, and design seems appropriate, much like the study of cinematic language for film. On the other hand, we argue that a PL approach better supports understanding of the player-to-game direction, since the language the player speaks toward a digital game is formal and unambiguous.

Researchers have previously recognized the value in formalizing interaction vocabularies, realizing certain interaction conventions as a *single* "video game description language" [7] whose implementation as VGDL [19] has been used in game AI research. We suggest instead that the design space of player languages is as varied as the design space of programming languages and herein give an account of what it would mean to treat each language individually. Our project suggests that an appropriately expressive computational framework analogous to VGDL should be one that can accommodate the encoding of many such languages, such as a *meta-logical framework* like the Twelf system for encoding and analyzing programming language designs [18].

Any investigation into formalizing actions within an interactive system shares ideas with "action languages" in AI extending as far back as McCarthy's situation calculus [13] and including planning languages and process calculi. These systems have been studied in the context of game design, e.g. the Ludocore system [20]; however, AI researchers are mainly interested in these formalisms as internal representations for intelligent systems and the extent to which they support reasoning. Conversely, we are interested their potential to support player expression and facilitate human-computer conversation.

Some theoretical and experimental investigations have been carried out about differences between game interfaces along specific axes, such as whether the interface is "integrated^ (or one might say diagetic), versus extrinsic to the game world in the form of menus and buttons [11, 12]. These investigations suggest an interest in more detailed and formal ontologies of game interfaces, which our work aims to provide.

From the PL research side, we note existing efforts to apply PL methodology to user interfaces, specifically in the case of program editors.

Hazelnut isaformalmodel ofaprogram editor that enforces that every edit state is meaningful (it consists of a well-defined syntax tree, with a well-defined type) [14]. Its type system and editing semantics permit *partial programs,* which contain missing pieces and well-marked type inconsistencies. Specifically, Hazelnut proposes a *editing language,* which defines how a cursor moves and edits the syntax tree; the planned benefits of this model range from better editing assistance, the potential to better automate systematic edits, and further context-aware assistance and automation based on statistical analysis of (semantically-rich) corpora of recorded past edits, which consist of *traces* from this language [15]. Likewise, in the context of game design, we expect similar benefits from the lens of language design.

3 A FRAMEWORK FOR PLAYER INTENT LANGUAGES

In the formal study of a programming langugage, one may define a language in three parts： syntax, type system, and operational semantics.

- The *syntax* is written in the form of a (usually) context-free grammar describing the allowable expressions. One sometimes distinguishes between *concrete syntax*, the literal program tokens that the programmer strings together in the act of programming, and *abstract syntax,* the normalized "syntax tree" structures that ultimately get interpreted.
- An *operational semantics* defines how runnable programs (e.g. a function applied to an argument) *reduce* to values. This part of the definition describes how actual computation takes place when programs in the language are run. It is important to note that the operational semantics need not reflect the actual *implementation* of the language, nor is it specific to a "compiled" versus "interpreted" understanding of the language： it is simply a mathematical specification for how any compiler or interpreter for the language should behave.
- A *type system* further refines the set of syntactically valid expressions into a set of *meaningful* expressions, and provides a mapping between an expression and an approximation of its meaning. Type systems are usually designed in conjunction with the operational semantics to have the property that every expression assigned a meaning by the type system should have a well-defined runtime behavior. In practice, however, type systems can only approximate this correspondence. Some err on the more permissive side-e.g. C's type system will permit invalid memory accesses with no language-defined behavior-and some err on the more restrictive side, e.g. Haskell's type system does not permit any untracked side-effects, at the expense of easily authoring e.g. file input/output (without first learning the details of the type system).

PL concept	Game concept	
Syntax	Recognized player intents	(Section 4)
Operational semantics	Game mechanics	(Section 5)
Type system Straight-line programs General programs	Contextual interface	(Section 6)
	Play traces	(Section 7)
	Player skills	(Section 8)

Table 1: Player intent languages: Formal decomposition (left) and correspondances (right).

Providing a formal language definition in programming languages research has several purposes. One is that it enables researchers to explore and prove formal properties of their language, such as *well-typed programs don't go wrong,* or in a language for concurrency, a property like deadlock freedom. However, an even more crucial advantage of a language specification is not mathematical rigor but human capacity to do science. A language definition is a *specification,* similar to an application programmer interface (API) or an IEEE standard： it describes an unambiguous interface to the language along an *abstraction boundary* that other human beings may access, understand, and implement, without knowing the internals of a language implementation. It is a necessary component of reproducibility of research, and it allows researchers to build on each other's work. We believe that an embrace of formal specification in games research can play a similarly important function.

Having provided loose definitions of these terms, we now wish to draw out the analogy between a *language* specification and a *game* specification. To treat a game in this manner, we wish to consider player affordances and actions, as well as their behavior (mechanics) in the context of the game's running environment. We summarize the components of this correspondence in Table 1.

We will use as a running example a minimal virtual environment with two player actions： (1) movement through a discrete set of rooms in a pre-defined map (move); (2) acquiring objects placed in those rooms to store in a player inventory (take). We consider five (somewhat aribitrary) possibilities in the design space of interfaces for such a game, summarized visually in Figure 2：

- **Point-and-Click:** A first-person viewpoint interface where the meaning of each click is defined based on the region the cursor falls in. Clicking near any of the four screen edges moves in that direction; clicking on a sprite representing an item takes it.
- **Bird's-Eye:** A top-down viewpoint interface where the player can see multiple rooms at once, and can click on rooms and objects that are far away, but those clicks only do something to objects in the same room or adjacent rooms.
- **WASD+:** A keyboard or controller-based interface with directional buttons (e.g. arrow keys or WASD) move an avatar in the correspondingi direction, and a separate key or button expresses the take action, which takes any object in the same room. (This interface may be used for either of the two views described above.)
- **Command-Line:** The player interacts by typing free-form text, which is then parsedinto commands, such as take lamp and move north.
- **Hypertext**: A choice-based interface where all available options are enumerated as textual links from which the player chooses.

In the following sections, we will consider these possibilities in light of design choices relevant to the specified aspect of PL design.

4 PLAYER INTENT AS SYNTAX

The *syntax* of a game is its space of recognized player intentions. Note that *intention* is different from *action* in the sense that we don't necessarily expect each well-formed intention to change anything about the game state： a player can intend to move north, but if there is no room to the north of the player when she expresses this intent, no change to the game's internal state will occur. Nonetheless, depending on the design goals of the

game, we may wish to recognize this as a valid intent so that the game may respond in some useful way (e.g. with feedback that the player cannot move in that direction).

In our example game, the choice of syntax answers questions such as: can the player click anywhere, or only in regions that have meaning? Can the player type arbitrary commands, or should we provide a menu or auto-complete text so as to prevent the player from entering meaningless commands? In PL, we can formalize these decisions by describing an *abstract syntax* for our language, which is typically assumed to be context-free and thus specified as a Bachus-Naur Form (BNF) grammar. Our examples below follow the interfaces shown visually in Figure 2.

WASD+ Interface: One way of writing the BNF for the WASD+ interface is:

direction ::= north | south | east | west
intent ::= move ⟨direction⟩ | collect

The hardware interface maps onto this syntax quite directly: each arrow key maps onto a move action in the corresponding direction, and the specified other key maps onto collect.

Bird's Eye View Mouse Interface: On the other hand, a clickingbased interface to a top-down map could enable the player to click on any room on the map and any item within a room. This syntax would look like:

room ::= courtyard | library | quarters | lab
item ::= flask | book
entity ::= room | item
intent ::= click ⟨entity⟩

Note that this syntax, compared to that for WASD+, describes a larger set of possible utterances, even though it has the exact same set of permitted game behaviors (a player may only move into adjacent rooms and take items that share a room with them).

Command Line Interface: The command-line interface would have an even larger space of expressible utterances if we consider all typed strings of characters to be valid expressions, but that syntax is too low-level for linguistic considerations. Supposing we interpose a parsing layer between arbitrary typed strings and syntactically-well-formed commands, we can define the abstract syntax as follows (where *direction* and *item* are assumed to be defined as they were in the previous examples):

intent ::= *move* ⟨*direction*⟩ / take ⟨item⟩

Assuming the player "knows the language," i.e. knows that move and take are valid commands, and in fact the *only* valid commands, and assuming that she knows how to map the visual affordances (e.g. image of the flask) to the typed noun (e.g. flask), the experience afforded by this interface is quite similar to the WASD+ interface. The main difference is that the player must specify an *argument* to the take command, asking the player to formulate a more complete (and unambiguous) intent by actually naming the object she wishes to take.

Hypertext interface: Finally, we consider the intent language for the hypertext interface. This is one of the most difficult interfaces to formulate in linguistic terms, because it either requires that we formalize link selection in an acontextual way (e.g. as a numeric index into a list of options of unknown size) or that we formulate each link *from each page* as its own separate command, each of which has meaning in only one specific game context (namely, when the player is on the page containing that link). The former feels like a more general formulation of hypertext that is not relevant to any particular game, and since we are aiming to provide a correspondence between specific games and languages, we opt for the latter:

intent ::= select ⟨choice⟩
choice ::= take_flask_from」ab
| take_book_from」ibrary
| go_south_from」ab
| go_east_from」ab
| go_west_from」ab
| go_north_from」ibrary
| . . .

Some hypertext authors putalot of effort into scaffolding the choicebased experience with a richer language, e.g. by repeating the same set of commands that behave in consistent ways across different pages, or by creating menu-like interfaces where text cycles between options on an otherwise static page. In this way, hypertext as a medium might be said as providing a platform for designers to create their own interface conventions, rather than relying on a set of pre-established ones; by the same token, hypertext games created by inexperienced interface (or language) designers may feel to players like being asked to speak a foreign language for each new game.

Additive and subtractive properties of syntax

By now we are able to observe that, just like the rest of a game's rules, its syntax has both additive and subtractive properties. It provides the menu of options for which hardware interactions are *relevant,* i.e. likely to result in meaningful interaction with the game system, but it also establishes which utterances within that set are *disallowed,* or ill-formed—e.g. that it is not meaningful to say "take" without providing an object to the command, or that "take north" is ill-formed.

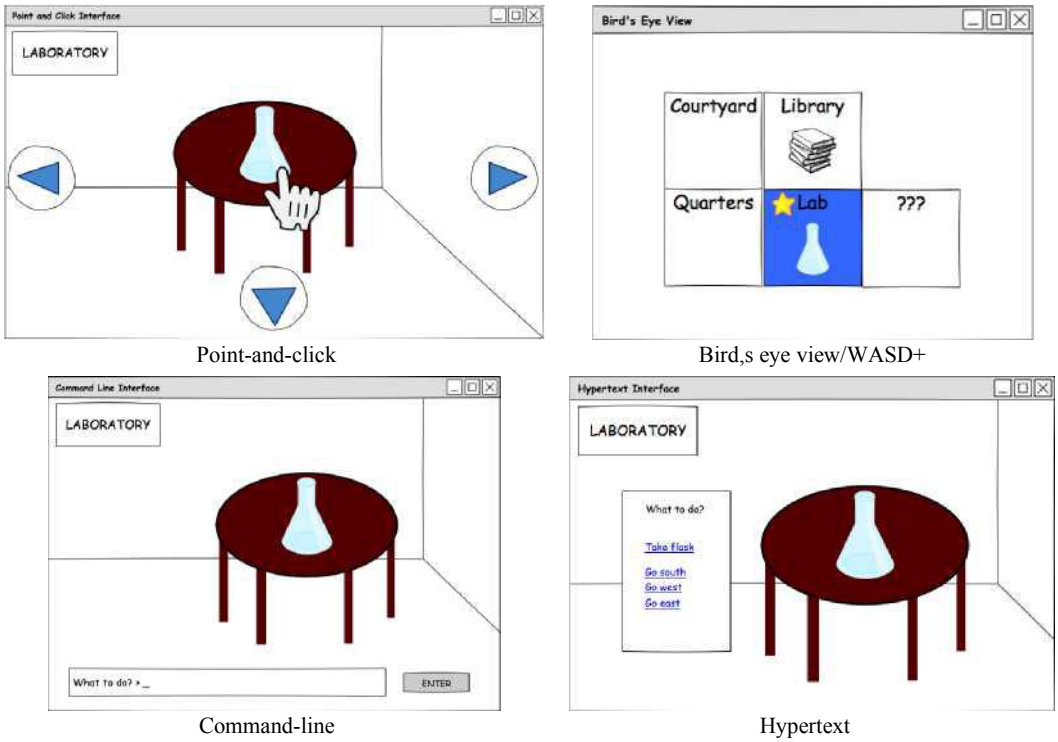


Figure 2: Four different user interfaces for the move/take game.

Correspondingly, an important decision that impacts game design is (a) how discoverable the additive affordances are (e.g. can the player determine that "examine" is a meaningful verb without already possessing literacy in the game's genre?) and (b) the extent to which the user interface makes meaningless expressions impossible to form. For example, in a hypertext interface, all links lead somewhere—so every intent the player can form, i.e. clicking a link on the page, will get a valid response from the game, whereas "take fnord" typed at a command-line interface may be recognized by the parser, but meaningless to a game where "fnord" is not a noun. Decisions about these two (related) dimensions will determine the extent to which *learning the language,* an exploratory but sometimes frustrating process, is a central challenge of the game.

5 GAME MECHANICS AS OPERATIONAL SEMANTICS

Consider the right-hand-side of Figure 1： The game parses and interprets an unambiguous syntax of player intent, which either advances the game state (as shown in the figure), or the game state cannot advance as the player intended, which the game somehow signals to the player (not shown).

We model a *response* to the move-take player intent with the following BNF definition, of *resp*：

$$resp ::= \text{success} \mid \text{failure}$$

Figure 1 shows the case where the player intent of "take flask" (formally, the syntax "take") leads to the game world performing this intent as a successful action, and responding accordingly with "Taken." Formally, we model this *resp* as "success," as defined above. Likewise, if the flask cannot be taken (e.g., the flask is not near the player, or is already in the player's possession, etc.), the game responds with failure.

As with the player's intent, which may exist as both raw input *and* as formal syntax, each formal response can be conveyed as raw output in a variety of ways (e.g., as textual words, pictures, or sounds). In real games there are often two two levels of game-toplayer feedback： Feedback through the game world, and feedback *outside* the game world (e.g., a pop-up message with an error, guidance or advice). For simplicity, move-take gives feedback outside of the game state, e.g., as pop-up messages.

To capture the formal relationship between player intent as syntax, and game response as syntax, we introduce a four-place *game-step* relation：

$$\langle G_i; \textit{intent}\rangle \rightarrow \langle G_2; \textit{resp}\rangle$$

This relation formalizes the dynamic behavior of the right-handside of Figure 1. It consists of four parts： An initial game state G_i, a player intent *intent,* a resulting game state G_2 and a game response *resp*.

As is standard in PL formalisms, we give the rules that define this relation as inductive *inference rules,* which can each be read as a logical inference. That is, given evidence for the premises on the top of the rule, we may conclude the bottom of the rule. For instance, here are two example rules：

$$\frac{G_i \text{ 1- playerNear flask} \qquad \text{playerTake}\langle G_i, \text{ flask}\rangle \equiv G_2}{\langle G_i;\ \text{take}\rangle \longrightarrow \langle G_2;\ \text{success}\rangle}$$

$$\frac{G \text{ i- not(playerNear flask)}}{\langle G;\text{take}\rangle \longrightarrow \langle G;\text{failure}\rangle}$$

The first rule formalizes the case shown in the RHS of Figure 1. The second rule formalizes the opposite outcome, where the flask cannot be taken. Notably, the first rule has two premises： To be taken by the player, it suffices to show that in the current game state G_i, the flask is near the

player (first premise), and that there exists an advanced game state G2 that results from the player taking this flask (second premise). In the second rule, there is only one game state G, since the flask cannot be taken and consequently, the game state does not change.

Like the syntax of player intent and game responses, these rules are also unambiguous. Consequently, we view these rules as a mathematical definition with an associated strategy for constructing formal (and informal) proofs about the game mechanics.

For instance, we can formally state and attempt to prove that for all player intents *intent* and game worlds, Gi, there exists a corresponding game world *G2* and game response *resp.* That is, the statement of the following conjecture:

$$\forall G_i,\ intent.\ \exists G_2,\ resp.\ (G_i;\ intent) \rightarrow (G_2;\ resp)$$

Using standard PL techniques, the proof of this conjecture gives rise to an abstract algorithm that implements the game mechanics by analyzing each possible case for the current game-state and player intent. Indeed, this is precisely the reasoning required to show that an implementation of the game is *complete* (i.e., there exists no state and input that will lead the game into an undefined situation).

Reasoning about this completeness involves reasoning about when each rule is applicable. For instance, the rule for a successful player intent of take requires two premises: Gi - playerNear flask and playerTake(Gi, flask) ≡ G?. The first is a *logical judgment* about the game world involving the proposition playerNear flask, which may or may not be true, but which is computable. The second is a *semantic function* that transforms a game state into one where the player takes a given object; in general, this function may be undefined, e.g., if the arguments do not meet the function's *pre-conditions*. For instance, a precondition of the function playerTake(—, -) may be that the object is not already in the possession of the player. In this case, to show that the game mechanics do not "get stuck", we must show that playerNear flask implies that the player does not already posses the flask. (Otherwise, we should add another rule to handle the case that the player intends to take the flask but already possesses it). The design process for programming language semantics often consists of trying to write examples and prove theorems, and failing; these experiences inform systematic revisions to the language definition.

In the next section, we refine intents and semantics further by introducing the notion of a *context.*

6 CONTEXTUAL INTERFACES AS TYPE SYSTEMS

Decisions about interface syntax can, to some extent, limit or expand the player's ability to form intents that will be met with failure, such as moving through a wall or taking an object that does not exist. But sometimes, whether an utterance is *meaningful* or not will depend on the runtime game state, and can be considered a distinct question from whether it is well-formed. For example, whether or not we can *take flask* depends on whether the flask is present, but if the flask is an object somewhere in the game, we must treat this command as well-formed *syntax* and relegate its failure to integrate with the runtime game environment to the *mechanics* (operational semantics).

However, some user interfaces nonetheless restrict the set of recognized utterances in a way that depends on current game state. Consider a point-and-click interface that changes the shape of the cursor to a hand whenever it hovers over an interactable object, and only recognizes clicks when it is in this state. Alternatively, consider the hypertext interface, which only recognizes clicks on links made available in the current page. Providing the player with *only the option of saying* those utterances that "make sense" in this regard corresponds to a strong static type system for a programming language.

Type systems are typically formalized be defining a relation between expressions *e* and contexts r. Contexts are sets of specific circumstances in which an expression is valid, or well-typed. Usually, these circumstances have to do with *variables* in the program. For example, the program expression $x + 3$ is only well-typed if x is a number. "x is a number" is an example of a fact that would be contained in the context. Its well-typedness could be represented as x:num - x + 3 ok.

In the move-take game, we can include aspects of game state in our context, such as the location of the player and the adjacency mapping between rooms in the world. An example of a typing rule we might include to codify the "only present things are takeable" rule would be:

$$\frac{r - playerIn(R)\quad r - at(O, R)}{r - take\ (O)\ ok}$$

We then need to define a relation between concrete game states G and abstract conditions on those states, r. We might write this relation G : r. After such rules are codified, we can refine the "game completeness" conjecture to handle only those utterances that are well-typed:

$$\forall G_i,\ intent.\ (G_i : r)\ \wedge\ (r - intent\ ok)$$
$$\exists G_2,\ resp.\ (G_i;\ intent) \rightarrow (G_2;\ resp)$$

This is nearly what we want to know about our game mechanics. However, we want to apply this reasoning iteratively as the game progresses, so that we reason next about the player intention that leads from game state G2 to another possibly different game state G3; but what context for player intent describes state G2?

For this reasoning to work, we generally need to update the original context r, possibly changing its assumptions, and creating r We write r ⊂ r' to mean that r' *succeeds* r in a well-defined way. Given that state Gi and *intent* agree about the context of assumptions r, we wish to show that there exists a successor context r' that agrees with the new game state G2:

$$\forall G_i,\ intent.\ (G_i : r)\ \wedge\ (r - intent\ ok)$$
$$G_2,\ resp.\ ((G_i;\ intent) \rightarrow (G_2;\ resp))$$
$$\wedge(G_2 : r',)\wedge(r \subset r',)$$

This statement closely matches the usual statement of *progress* for programming languages with sound type systems.

7 PLAY TRACES AS STRAIGHT-LINE PROGRAMS

If we consider the analogy of game interfaces as programming languages, the natural question arises, what is a *program* written in this programming language? We want to at least consider individual, atomic player actions to be complete programs; the preceding text provides such an account. But typical programs are more than one line long—what does it mean to sequence multiple actions in a game language?

In a typical account of an imperative programming language, we introduce a sequencing operator ; where, if c_i and C_2 are commands in the language, then c_i； C_2 is also a command. The operational semantics of such a command involves the composition of transformations on states *a*:

$$\frac{\langle b;\ C_i\rangle \rightarrow b_{i\,01};\ C_2\rangle \rightarrow g}{\langle a;\ (C_i;\ C_2)\rangle \rightarrow b_2}$$

However, interactive software makes this account more complicated by introducing the program response as a component. Instead of issuing arbitrary commands in sequence, the player may wait for a response or process responses in parallel with their decisions. In this respect, a player's "programming" activity more closely resembles something like live-coding than traditional program authoring. Execution of code happens alongside its authorship, interleaving the two activities. If we consider a round-trip through the game loop after each individual command issued, then what we arrive at is a notion of program that resembles a *play trace*： a log of player actions and game responses during a play session, e.g.

PLAYER： go north
GAME： failure
PLAYER： take flask
GAME： success
PLAYER： go south
GAME： success

Depending on the richness of our internal mechanics model, this play trace may contain useful information about changes in internal state related to the preconditions and effects of player actions. But the main important thing to note is that, despite the informal syntax used to present them here, these traces do not consist of strings of text entered directly by the player or added as log information by the game programmer—they are structured terms with abstract syntax that may be treated to the same formal techniques of interpretation and analysis as any program. And this syntax is at a high level of game-relevant interactions, not at the level of hardware inputs and engine code.

Researchers in academia and the games industry alike have recently been increasingly interested in play trace data for the sake of analytics, such as understanding how their players are interacting with different components of the game and responding to this information with updates that support player interest [8]. For the most part, this trace data is collected through telemetry or other indirect means, like game variable monitoring, after which it must be analyzed for meaning [3]. More recently, systems of structured trace terms that may be analyzedwith logical queries have been proposed [i6], identifying as a benefit an ability to support automated
testing at the level of design intents. Our PL analogy supports this line of inquiry and warrants further comparison and collaboration.

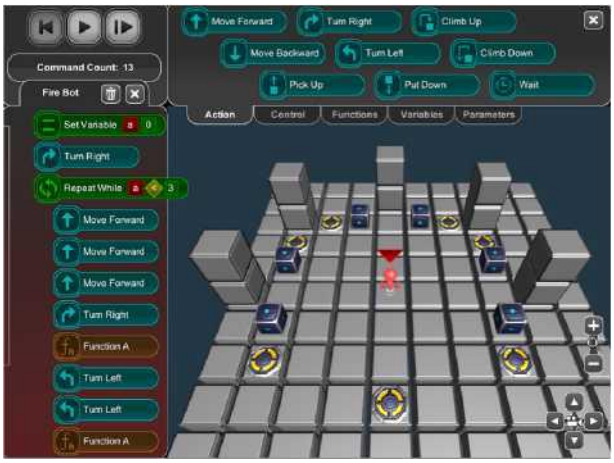


Figure 3: A screenshot from BOTS, an educational game in which players write programs to direct a player avatar.

8 PLAYER SKILLS AS GENERAL PROGRAMS

While considering "straight-line" traces may have some utility in player analytics, a more exciting prospect for formalizing game interactions as program constructs is the possibility of encoding *parameterized* sequences of actions that may carry out complex tasks. After all, games with rich player action languages afford modes of exploratory and creative play： consider item crafting in Minecraft [i7], puzzle solving in Zork [2], or creating sustainable autonomous systems like a supply chain in Factorio [2i], a farm in Stardew Valley [i], or a transit system in Mini Metro [5]. Each of these activities asks the player to understand a complex system and construct multi-step sequences of actions to accomplish specific tasks. From the player's perspective, these plans are constructed from higher-level activities, such as *growing a crop* or *constructing a new tool,* which themselves are constructed from the lower level game intent language.

A language, as we have formalized it, gives us the atomic pieces from which we can construct these sequences, like Lego bricks can be used to construct reconfigurable components of a house or spaceship. *Compositionality* in language design is the principle that we may understand the meaning and behavior of compound structures (e.g. sequences) in terms of the meaning and behavior of each of its pieces (e.g. actions), together with the meaning of how they are combined (e.g. carried out one after the other, or in parallel). In this section, we describe how we might make sense of *player skills* in terms of programs written in a more complex version of the player language.

Such programs might be integrated into a game's mechanics so that a player explicitly writes such programs, as in the BOTS game, an interactive programming tutor that asks players to write small imperative programs that direct an avatar within a virtual

Figure 4: A screenshot from Stardew Valley showing the player's farm, inventory, and avatar.

world [10] (see Figure 3), or Cube Composer[2], in which players write functional programs to solve puzzles. However, for now, we primarily intend this account of player skills as a conceptual tool.

8.1 Example: Stardew Valley

Our initial ⟨move, take⟩ example is too simple to craft really compelling examples of complex programs, so here we examine Stardew Valley and its game language for the sake of considering player skills. In Stardew Valley, the player has an inventory that permits varied interactions with the world, beginning with a number of tools for farming (axe, hoe, scythe, pickaxe) which do different things in contact with the resources in the surrounding environment; most include extracting some resource (wood, stone, fiber, and so on), which themselves enter the player's inventory and can be used in further interaction with the game world. The player's avatar is shown on-screen, movedby WASD. There are also context-sensitive interactions between the player and non-player characters (NPCs), interfaces through which new items may be purchased (shops), and mini-games including fishing (fish may also be sold at high value). See Figure 4 for a typical player view of the game.

While a full account of the language that this game affords the player is beyond the scope of this paper, we include a representative sample of the actions and affordances found in this game that may be used to construct player skills. These include： directional avatar movement within and between world "rooms" point-and- click actions for selecting items in one's inventory, and interacting with in-room entities.

The player avatar must be near an entity for the player to interact with it. They can then either apply the currently-selected inventory item to the in-world entity with a left-button click, or right-button click, which does something based on the entity type, e.g. doors and chests open, characters speak, and collectible items transfer to the player's inventory. We refer to this last action as inquire. We also note that, for the sake of our example, movement towards an entity and movement offscreen (towards another room) are the only meaningful and distinct types of movement, which we refer to as move_near and move_。仟 screen. These actions yield the following syntax：

intent ::= select ⟨item⟩
| apply ⟨entitg⟩
| \nqu\re *(entity)*
| move_near ⟨entity⟩
| move_o 仟 screen ⟨direction⟩

We leave the definition of items and entities abstract, but we could imagine it to simply list all possible items and entities in the world as terminal symbols. From these atomic inputs, we can start to construct higher-level actions performed in the game most frequently— tilling land, planting seeds, conversing with NPCs, and so on. These blocks of code may be assigned names like functions to be called in many contexts：

```
action till = select hoe; move_near hard_ground;
              apply hard_ground
action plant = select seeds; move_near tilled_ground; apply
              tilled_ground
action mine = select pickaxe; move_near rock; apply rock action
talk = move_near npc; inquire npc
action enter_shop = move_near shop; inquire door(shop)
```

In turn, these larger skill molecules may be combined to accomplish specific tasks or complete quests.

8.2 Branching programs as skills and strategies

Note that we have naively sequenced actions without consideration for the game's response. This approach to describing player skills does not take into account the possbility of a failed attempt, such as attempting to mine when there are no rocks on the current screen. One could simply assign a semantics to these sequences of action that threads failure through the program—if we fail on any action, the whole compound action fails.

However, we can go further with describing robust player skills and strategies if we consider the possibility of *handling* failure, a common feature of day-to-day programming and indeed of gameplay. Recall our simplified game response language consisting of two possibilities, success and failure. We can introduce a case construct into our language to handle each of these possibilities as a distinct branch of the program：

```
action mine =
  response = select pickaxe;
  case(response):
    success => {
      response[1] = move_near rock;
      case(response[1]): success => apply rock;
        | failure => fail;
    }
  | failure => fail;
```

However, to avoid handling failure at every possible action, a better approach is to explicitly indicate as parameters the world resources that each action needs in order to complete successfully. The overall action definition for mining, or example, would require as a precondition to the action that a pickaxe is available in the player

[2]http：//david-peter.de/cube-composer/

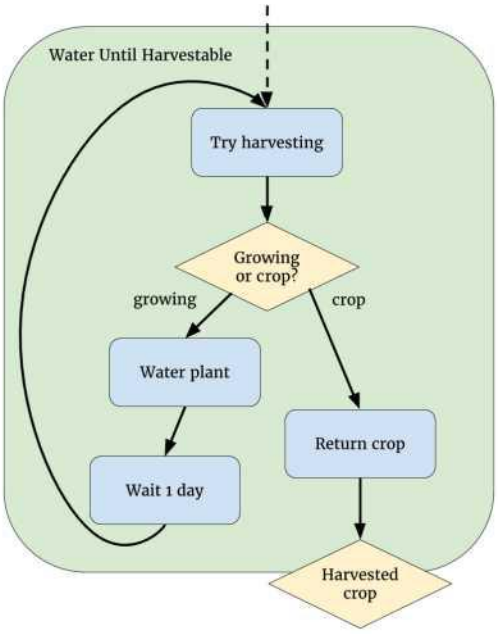


Figure 5: Watering a crop until it may be harvested.

inventory and a rock is in the same room. The actions for selecting the pickaxe and moving near the rock would depend on these resources, and the game response language could include the resources it guarantees as outputs. Then we can write the program using simpler notation that refers to resource dependencies of the appropriate type (using notation resource : type):

```
action mine(p：pickaxe, r：rock)二
  select p; move_near r; apply r
:mineral
```

This notation together with a branching case construct scales to include nondeterminism in the game world, such as the fishing minigame in Stardew Valley： the game always eventually tells the player that something is tugging at her line, but some portion of the time it is a fish while the rest of the time it is trash. These constructs can also account for incomplete player mental models, such as knowing that one must water her seeds repeatedly day after day in order to grow a crop, but not knowing how many times.

Below, we present a notation that accounts for these aspects of player skills: ado ... recv ... notation indicates a command and then binds the response to a *pattern,* or structured set of variables, which can then be case-analyzed. Our first example is watering a crop until it may be harvested:

```
action water_until_harvestable[t](p: planted(t))
:crop(t)二
  do try_harvest(p)
    recv <result: crop(t) + growing(t)>.
      case result of
        c:crop(t) => c
      | g:growing(t)=>
          water(g);
          wait(day);
          try_harvest(g)
```

See Figure 5 for a control flow diagram of this code.

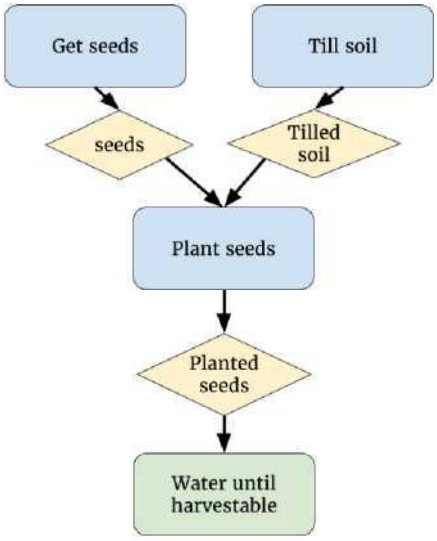


Figure 6: Growing a crop.

The next example shows a parallel construct || , which can be used to compose actions with distinct dependencies, as well as how an action definition may use other action definitions by threading resource dependencies through as arguments:

```
fun grow_crop[t : croptype](s:soil, w:watering_can)
:crop(t)二
  do
    get_seeds(t) || till_soil(s)
  recv <s: seeds(t), g: tilled_soil>.
    do
      plant(s, g)
    recv <p: planted(t)>.water_until_harvestable(p)
```

See Figure 6 for a control flow diagram of this code.

9 DISCUSSION

Having established a vocabulary of syntax and semantics for player languages, we can now revisit the potential benefits of this account mentioned in the introduction and discuss them in more detail.

9.1 Composing play traces, player skills

One of the major things that a programming language account provides is *compositionality*： a system for making sense of meaning of a complex artifact in terms of the meaning of its pieces. This comes up in two places for looking at games：

Structured play traces. With a formalized game language, the sequence of steps along the transition system described by the operational semantics forms a mathematical artifact that is subject to deeper analysis than what can be gained simply from screen or input device recordings. For example, we can carry out causal analysis, asking "why" queries of trace data, e.g. “Why was the player able to unlock the door before defeating the boss?”, as well as filtering traces for desired properties: “Show me a play trace where the player used something other than the torch to light the room.” The recent PlaySpecs project [16] suggests interest in formulating traces this way to support this kind of query.

Player skills as programs. While a play trace may be interpreted as a straight-line program, even more interesting is the idea of latent structure in player actions, such as composing multiple low-level game actions into a higher-level skill, following the cognitive idea of "chunking" We map this idea onto that of *functions* in the programming language that take arguments, generalizing over state space possibilities (e.g.: the red key opened the red door, so for all colors *C,* a *C* key will open a *C* door). Further reasoning forms like case analysis to handle unpredictable game behavior and repeating an action until a condition holds are also naturally expressed as programming language constructs.

9.2 Abstraction boundaries between input and mechanics

Formalizing game interfaces gives us the tools to explore alternative interfaces to the same underlying mechanics, without needing to port game logic between different graphical interface frameworks. For example, the interactive fiction community has been exploring alternatives to the traditional dichotomy of "parser vs. hypertext" for presenting text-based games and interactive story-worlds. An abstraction boundary between the underlying mechanics, map, and narrative of the world, and the view and input mechanisms used to interact with it, could open the doors to research on user interfaces that support players' mental models of a world conveyed in text.

9.3 Enabling co-creative play

Finally, a PL formulation of player actions along with appropriate composition operators (parallel and sequential composition, branching, and passing resource dependencies) provides a "scripting language for free" to the game environment. Such a language can be used to test the game, provided as a game mechanic as in BOTS, or provided as an optional augmentation to the game's mechanics for the sake of modding or adding new content to the game world. Especially in networked game environments, like multi-user domains, massively multiplayer online games, and social spaces like Second Life, the ability for the player to program not just her avatar but parts of the game world itself introduces new opportunities for creative and collaborative play. Our framework suggests a new approach to designing these affordances for players in a way that is naturally derived from the game's existing mechanics and interface. In the spirit of *celebrating the player,* this year's conference theme, we wish to enable the player as a co-designer of her own game experience.

9.4 Future Work

In future work, we intend to build software for realizing game language designs and experimenting with protocol-based game AI developed as programs in these languages. In another direction, we aim to innovate in PL design outside of games, such as read-eval-print loops (REPLs) for live programming that includes rapid feedback loops motivated by gameplay, as well as distributed and concurrent systems that may benefit from the protocol-based approach proposed here.

10 CONCLUSION

We propose *player intent languages,* a systematic framework for applying programming language design principles to the design of player-game interfaces. We define this framework through simple examples of syntax (aka player intents), type systems (aka contextual interfaces) and operational semantics (aka game mechanics). We show how applying this framework to a player-game interface naturally gives rise to formal notions of play traces (as straight-line programs) and player skills (as general programs with branching and recursion). By defining a player intent language, game design concepts become formal objects of study, allowing existing PL methodology to inform and guide the design process.

REFERENCES

[1] Eric Barone. 2016. Stardew Valley. Digital distribution. (2016).

[2] Marc Blank and David Lebling. 1980. Zork I: The Great Underground Empire. (1980).

[3] Alessandro Canossa. 2013. *Meaning in Gameplay: Filtering Variables, Defining Metrics, Extracting Features and Creating Models fOr GameplayAnalysis*. Springer London, London, 255-283. DOI: h 仕 p: rdx.doi.org/10.1007/978-1-4471-4769-5_13

[4] Rogelio E Cardona-Rivera and R Michael Young. 2014. Games as conversation. In *Proceedings of the 3rd Workshop on Games and NLP at the 10th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.* 2-8.

[5] Dinosaur Polo Club. 2015. Mini Metro. (2015).

[6] Chris Crawford. 2003. *Chris Crawford on game design.* New Riders.

[7] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. 2013. Towards a video game description language. (2013).

[8] M Seif El-Nasr, Anders Drachen, and Alessandro Canossa. 2013. Game analytics. *New York, Sprint* (2013).

[9] H Paul Grice. 1975. Logic and conversation. *1975* (1975), 41-58.

[10] Andrew Hicks. 2012. Creation, evaluation, and presentation of user-generated content in community game-based tutors. In *Proceedings of the International Conference on the Foundations of Digital Games*. ACM, 276-278.

[11] Kristine Jorgensen. 2013. *Gameworld interfaces*. MIT Press.

[12] Stein C Llanos and Kristine Jorgensen. 2011. Do players prefer integrated user interfaces? A qualitative study of game UI design issues.

[13] John McCarthy and Patrick J Hayes. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence* (1969), 431-450.

[14] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *POPL*.

[15] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *SNAPL: The 2nd Summit on Advances in Programming Languages*.

[16] J Osborn, BenSamuel, MichaelMateas, and Noah Wardrip-Fruin. 2015. Playspecs: Regular expressions for game play traces. *Proceedings of AIIDE* (2015).

[17] Markus Persson, Jens Bergensten, Michael Kirkbride, Mark Darin, and Carl Muckenhoupt. 2011. Minecraft. (2011).

[18] Frank Pfenning and Carsten Schiirmann. 1999. System description: Twelf-a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*. Springer, 202-206.

[19] Tom Schaul. 2013. A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 1-8.

[20] Adam M Smith, Mark J Nelson, and Michael Mateas. 2010. Ludocore: A logical game engine for modeling videogames. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 91-98.

[21] Wube Software. 2012. Factorio. (2012).

[22] Jose P Zagal, Michael Mateas, Clara Fernandez-Vara, Brian Hochhalter, and Nolan Lichti. 2007. Towards an ontological language for game analysis. *Worlds in Play: International Perspectives on Digital Games Research* 21 (2007), 21.

中国计量大学现代科技学院

毕业设计（论文）外文翻译

学生姓名：朱铸杰　学 号：1630332228

专　　业：计算机科学与技术

班　　级：计算机 162

设计（论文）题目：

基于 layabox 的微信端魔塔小游戏开发

指导教师：黄俊

系　　部：信息工程系

2020 年 3 月 20 日

游戏语言

寻求游戏界面的语义基础

摘要

正式的游戏模型可帮助我们解释和预测行为，从而带来更强大和创新的设计。尽管游戏研究界对游戏体验的“一半游戏”（游戏模型，游戏描述语言）和“人物一半”（玩家建模）都提出了许多形式主义的说法，但两者之间的接口却鲜有关注，尤其是在玩家表达自己对游戏的意图方面。我们描述了一种基于编程语言理论的分析和计算工具箱，以检查控制方案和游戏规则之间的现象，我们将其识别为每种游戏的独特玩家意图语言。

关键词

游戏界面，编程语言，形式方法

1 介绍

为了研究玩家与游戏的互动方式，我们研究了底层系统的规则以及玩家做出的选择。玩家建模领域已经确定了构建玩家认知模型的价值：尽管游戏作为一个独立的实体可以让我们了解其作为形式系统的机制和特性，但除非我们了解该系统的动态性，否则我们无法理解也占等式的一半。同时，克劳福德确定了查看玩家与数字游戏之间建立的完整信息循环的必要性，并将游戏中的互动性定义为他们与玩家进行对话（包括聆听，处理和响应）的能力，并确定了这三个角色的重要性整体体验。

有了对游戏对话的理解，我们应该期望发现一种类似于游戏和玩家进行对话的语言。在图 1, 我们将游戏玩家循环说明为一个过程，其中包括构成这种语言的界面。游戏本体项目描述游戏界面如下：

界面是玩家与游戏相遇的地方，是玩家体现的反应与游戏实体操纵之间的映射。它指的是玩家如何与游戏互动以及游戏如何与玩家交流。

第一部分，即玩家与游戏的互动方式，称为输入，进一步细分为输入设备和输入方法。输入设备是硬件控制器（鼠标，键盘，操纵杆等）和输入方法的表面更具语义：它们包括有关操作轨迹（玩家可以控制哪些游戏实体？）以及直接还是间接动作的选择，例如，从选项菜单中选择一个动作（间接）与按箭头键移动化身（直接）)。但是，只要仔细研究互动小说，近期的手机游戏或节奏游戏（仅举几个例子），就会发现输入法的设计选择比这两个维度具有更多的多样性和可能性。在本文中，我们提出了一个框架来支持对设计空间的分析和探索。我们的第一步是改进输入法来输入语言：我们最终要问，玩家如何传达自己的意图，数字游戏如何识别这种意图？因此，从语言上讲，这种语言的“音素”是硬件控件，例如按钮按下和操纵杆移动。然后，由每个游戏分别定义语法和语义，具体取决于它们赋予每个控件输入的含义。这种语言定义了游戏的动词，可能包括移动，选择库存物品，检查世界物品，应用或使用物品，进入房间以及战斗动作。（请注意，这种语言也不同于游戏的机制：机制包括不受玩家控制的系统行为，例如由于重力坠落，非玩家角色动作和其他自主行为）。在游戏设计师的指导下-她必须向玩家传达哪些动词可用-并且受到限制由她-她可以宣布某些表达无效。

由于对这种语言的限制完全由软件（游戏界面）决定，因此我们认为与自然语言相比，它与编程语言有更多共同点。因此，每个游戏在某种意义上都定义了自己的编程语言。用一个口号，我们可以将此项目游戏称为编程语言。具体来说，我们建议玩家意图语言作为 PL 框架来设计玩家游戏界面。

这种类比打开了尝试应用于游戏的整个方法论领域。PL 研究社区在将数学形式的语义分配给语言并分析这些语义方面有着悠久而深厚的历史。随着游戏研究人员对他们组装的系

统的新兴后果越来越感兴趣，PL 理论的工具提供了很多。例如，PL 理论提供了组成性的说明，即表达片段如何组合在一起以形成更高层次的含义。在游戏中，这转化为能够将玩家的技能或策略理解为玩家行为的组成，我们将在以下内容中进行演示

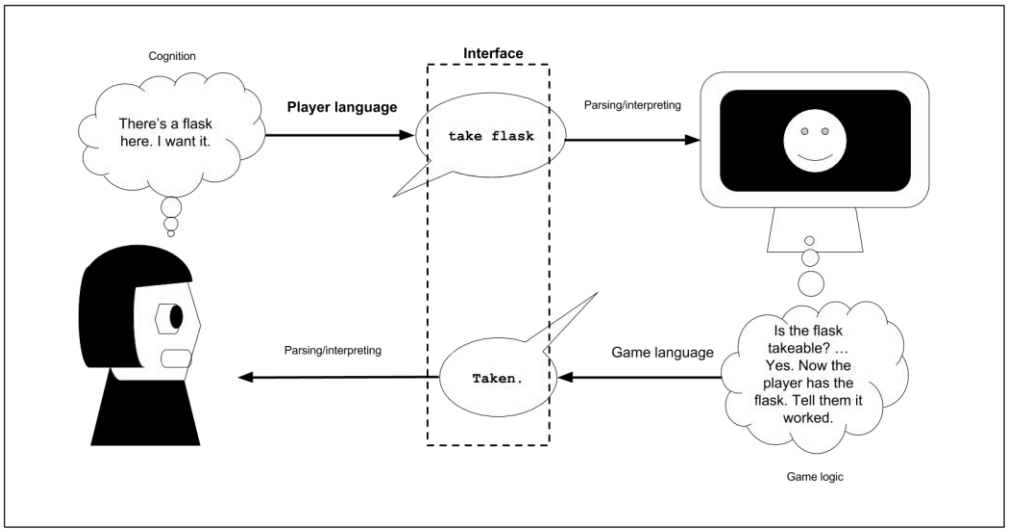


图 1：游戏循环的流程图：与语言，界面和认知有关的玩家和游戏对话。

本文通过使用一种形式化的输入语言作为一种“玩家 AI 脚本语言”。此外，通过将玩家的表达语言本身视为学习的对象，我们将他们作为游戏提供的体验的共同设计者。当我们不仅仅将玩家的互 动视为推动和揭示设计者意图的任意顺序的按键操作，而是将其 视为游戏系统必须聆听和响应的独特声音时，我们使玩家能够与 游戏者共同创造系统，有可能发展更深层次的系统理解和情感投 入。在本文中，我们提出了玩家意图语言，这是一种基于编程语 言的方法，用于将玩家游戏界面设计为形式对象。在本文的其余 部分，我们将通过具体示例介绍这种方法。具体来说，我们考虑 一个简单的游戏设计空间，并通过引入编程语言的组件来精确指 出该空间中的点：抽象语法（本节 4）， 类型系统（部分 6）,和操作语义（部分 5）。对于每种游戏，我们都给出游戏世界中的相应概念。通过将这些游戏概念扎根于类似的编程语言概念，我们获得了功能强大的 PL 推理工具和有利于游戏设计过程的设计方法。

我们通过将隐喻与戏剧轨迹作为直线程序扩展来证明这一思路 的回报（本节 7），和玩家技能作为一般程序（例如，具有参数，分支和循环的程序）（部分 8）。这些结构分别为完成游戏世界中的任务提供了语义日志和一般策略。玩家意图语言的框架 提出了进一步的研究方向，我们在总结之前简要概述和讨论（本节 9 和 10）。

2 相关工作

卡多纳·里维拉和杨详细描述了口号游戏作为对话之后的概念框架，奠定了认知科学中游戏对人与人之间对话理解的交流策略， 例如 Grice 的格言。

它们提供了一种语言学和符号学的方法，以了解游戏如何向玩家传达能力（行动的可能性）。考虑到游戏方程式的一半， 包括打算由玩家处理的视觉，文本和音频反馈机制，这种语言学，心理学和设计的应用似乎是适当的，就像研究电影语言一样。电影。另一方面，我们认为 PL 方法更好地支持了玩 家对游戏方向的理解，因为玩家对数字游戏所讲的语言是正式而明确的。

研究人员以前已经认识到正规化交互词汇的价值，将某些交互约定实现为一种单独的“视频游戏描述语言”，其实现为 VGDL 已用于游戏 AI 研究。相反，我们建议播放器语言的设计空间与编程语言的设计空间一样多，并且在此处说明了分别对待每种语言的含义。我们的项目建议，一个类似于 VGDL 的适当表达的计算框架应该能够容纳许多这种语言的编码，

例如像 Twelf 系统这样的元逻辑框架，用于编码和分析编程语言设计。

对交互式系统中的动作进行形式化的任何调查都与 AI 中的“动作语言”共享思想，这种思想可以追溯到麦卡锡的情况演算，包括计划语言和过程计算。这些系统已经在游戏设计的背景下进 行了研究，例如 Ludocore 系统;但是，AI 研究人员主要对这些形式主义感兴趣，这些形式主义是智能系统的内部表示形式以 及它们在多大程度上支持推理。相反，我们对他们支持玩家表达并促进人机对话的潜力感兴趣。

关于游戏界面之间沿特定轴的差异的一些理论和实验研究，例如界面是否是“集成的”（或者有人会说以菜单和按钮的形式出现在游戏世界之外）。这些调查表明我们对我们的工作旨在提供更详细和正式的游戏界面本体感兴趣。

从 PL 研究的角度，我们注意到将 PL 方法应用于用户界面的现有工作，特别是在程序编辑器中。榛子是程序编辑器的正式模型，它强制每个编辑状态都有意义（它由定义明确的语法树和定义明确的类型组成）。它的类型系统和编辑语义允许部分程序，其中包含缺少的片段和标记明确的类型不一致。具体来说，榛子提出了一种编辑语言，它定义了光标如何移动和编辑语法树。该模型的计划收益包括更好的编辑帮助，更好地自动化系统编辑的潜力，以及基于（丰富语义的）记录的过去编 辑的语料库的统计分析的进一步的上下文感知帮助和自动化，其中包括来自此的痕迹语言。同样，在游戏设计的背景下， 我们期望从语言设计的角度获得类似的好处。

3 一个框架

玩家意图语言

在对编程语言的正式研究中，可以将语言定义为三个部分：语法，类型系统和操作语义。

•该语法以（通常）上下文无关的语法形式描述允许的表达式。有时会区分具体的语法，程序员在编程时将它们串在一起的文字程序令牌和抽象的语法，最终得到解释的规范化“语法树”结构。

•操作语义定义了可运行程序（例如，应用于自变量的函数）如何减少为值。定义的这一部分描述了运行该语言的程序时如何进行实际计算。重要的是要注意，操作语义不必反映语言的实际实现，也不是特定于对语言的“编译”与“解释”理解：它只是用于说明任何编译器或解释器如何使用的数学规范。语言应该表现出来。

•类型系统进一步将一组语法有效的表达式精炼为一组有意义的表达式，并提供一个表达式及其含义的近似值之间的映射。通常将类型系统与操作语义一起设计为具有以下属性：类型系统为其分配了含义的每个表达式都应具有定义明确的运行时行为。但是实际上， 类型系统只能近似该对应关系。在较宽松的方面有些错误（例如，C 的类型系统将允许无效的内存访问，而没有语言定义的行为），而在较严格的方面某些错误（例如，Haskell 的类型系统不允许任何未跟踪的副作用），但这样做的代价是轻松编写例如文件输入/输出（无需先学习类型系统的详细信息）。

PL 概念	游戏概念	
句法	公认的球员意图	（部分 4)
操作语义	游戏机制	（部分 5)
类型系统	上下文界面	（部分 6)
直线程序	播放痕迹	（部分 7)
一般课程	球员技巧	（部分 8)

表 1：播放器意图语言：形式分解（左）和对应关系（右）。

在编程语言研究中提供正式的语言定义有几个目的。其中之一是，它使研究人员能够探索和证明其语言的形式属性，例如类型正确的程序不会出错，或者使用并发语言，例如死锁自由。但是，语言规范的一个更为关键的优势不是严格的数学原理，而是人类的科学能力。语言定义是一种规范，类似于应用程序程序员接口。

（API）或 IEEE 标准：它沿抽象边界描述了该语言的明确接口，其他人可能在不了解其

内部的情况下访问，理解和实现该接口。语言实现。它是研究可重复性的必要组成部分，它使研究人员可以相互借鉴。我们认为，在游戏研究中采用正式规范可以起到类似重要的作用。

在提供了这些术语的宽松定义之后，我们现在希望得出语言规范和游戏规范之间的类比。为了以这种方式对待游戏，我们希望考虑玩家的能力和行为，以及他们在游戏运行环境中的行为（机制）。我们在表中总结了这种对应关系的组成部分 1。

我们将以一个具有两个玩家动作的最小虚拟环境作为运行示例：（1）在预定义地图中移动离散的一组房间（移动）；

（2）获取放置在这些房间中的物品以存储在玩家的物品清单中（拿走）。我们在此类游戏的界面设计空间中考虑了五种（某种程度任意）的可能性，在图 2 中直观地进行了总结。

•指向并单击：第一人称视角界面，其中，每次单击的含义是根据光标所在的区域定义的。在四个屏幕边缘中的任意一个附近单击都会向该方向移动；单击代表项目的精灵即可。

•鸟瞰：一种自上而下的视点界面，玩家可以一次看到多个房间，并且可以单击房间和远处的物体，但是这些单击只会对同一房间或相邻房间中的物体产生作用。

•WASD+：具有方向按钮（例如，箭头键或 WASD）的基于键盘或控制器的界面可在相应的方向上移动化身，而单独的键或按钮则表示执行动作，该动作将同一房间中的任何物体作为目标。（此接口可用于上述两个视图中的任何一个。）

•命令行：播放器通过键入自由格式的文本进行交互，然后 将其解析为命令，例如“开灯”和“向北移动”。

•超文本：基于选择的界面，其中所有可用选项都被枚举为玩家选择的文本链接。

在以下各节中，我们将根据与 PL 设计的特定方面相关的设计选择来考虑这些可能性。

4 玩家意图作为语法

游戏的语法是公认玩家意图的空间。请注意，意图与动作有所不同，因为我们不一定期望每个形式良好的意图都会改变游戏状态：玩家可以打算向北移动，但是如果玩家没有向北移动的空间当她表达此意图时，不会改变游戏的内部状态。但是，根据游戏的设计目标，我们可能希望将其识别为有效意图，以便游戏可以以某种有用的方式做出响应（例如，获得玩家无法朝该方向移动的反馈）。

在我们的示例游戏中，语法的选择可以回答以下问题：玩家可以单击任何地方还是仅在具有意义的区域中单击？玩家可以键入任意命令，还是应该提供菜单或自动完成文本，以防止玩家输入无意义的命令？在 PL 中，我们可以通过描述我们语言的抽象语法来形式化这些决策，该抽象语法通常被认为是无上下文的，因此被指定为 Bachus-Naur 形式（BNF）语法。下面的示例遵循图 5 中直观显示的界面 2.

WASD +接口：为 WASD +接口编写 BNF 的一种方法是：

方向::=北|南|东|西方

意图::=move（方向）|收藏

硬件接口非常直接地映射到此语法：每个箭头键映射到相应方向上的移动动作，指定的其他键映射到 collect。

鸟瞰鼠标界面：另一方面，从上到下的地图基于单击的界面可以使玩家单击地图上的任何房间以及房间内的任何物品。该语法如下所示：

房间::=庭院|图书馆|宿舍|实验室

项目::=烧瓶| 图书

实体::=房间|项目

意图::=点击（实体）

请注意，与 WASD +相比，此语法描述了更多可能的发音，即使它具有完全相同的允许游戏行为集（玩家只能进入相邻的房间并拿走与他们共用一个房间的物品）。

命令行界面：如果我们将所有键入的字符串视为有效的表达式，则命令行界面将具有更大的可表达语音空间，但是出于语言考虑，该语法太底层了。假设我们在任意类型的字符串和语法正确的命令之间插入一个解析层，则可以定义抽象语法如下（假定方向和项目的定义与前面的示例相同）：

意图:=move（方向）|进项（项目）

假设玩家“知道语言”，即知道移动和采取是有效命令，并且实际上是唯一有效命令，并假设她知道如何将视觉能力（例如烧瓶的图像）映射到键入的名词（例如长颈瓶），此界面提供的体验与 WASD +界面非常相似。主要区别在于，玩家必须为 take 命令指定一个参数，要求玩家通过实际命名自己想采取的物体来表达更完整（明确）的意图。

超文本接口：最后，我们考虑超文本接口的意图语言。这是用语言表达的最困难的接口之一，因为它要么要求我们以上下文关系的方式来形式化链接选择（例如，作为对未知大小的选项列表的数字索引），要么我们从每个页面作为其自己的单独命令，每个命令仅在一个特定的游戏上下文中有意义（即当玩家在包含该链接的页面上时）。前者感觉像是一种超常规的超文本形式，与任何特定游戏都不相关，并且由于我们旨在提供特定游戏和语言之间的对应关系，因此我们选择后者：

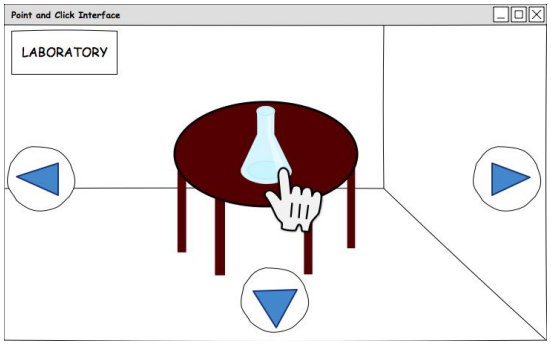
意图:: =选择（选择）

选择:: =从实验室拿瓶| 从图书馆取书|从实验室往南走|从实验室往东走|从实验室往西走|从图书馆往北走|. . .

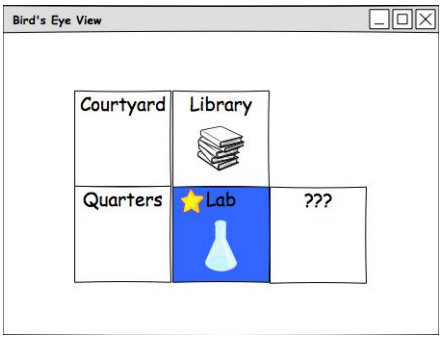
一些超文本作者付出了很多努力，以丰富的语言来支持基于选择的体验，例如，通过重复在不同页面上以一致的方式运行的同一组命令，或者在文本循环之间创建类似菜单的界面，否则为静态页面上的选项。这样，可以说超文本作为一种媒介为设计人员提供了一个创建自己的接口约定的平台，而不是依赖于一组预先建立的约定。同样，经验不足的界面（或语言）设计人员创建的超文本游戏可能会让玩家感到被要求为每个新游戏说一门外语。

语法的加法和减法属性

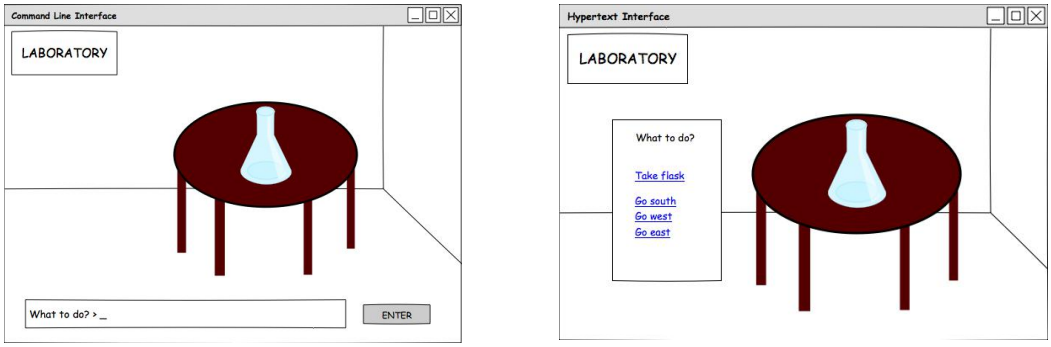
到目前为止，我们已经可以观察到，就像其他游戏规则一样， 其语法具有加法和减法属性。它提供了与硬件交互相关的选项菜单，即可能导致与游戏系统的有意义的交互，但是它还确定了该组中哪些话语是不允许的或格式不正确的，例如说 这没有意义。不向命令提供对象的情况下“采取”，或“采取向 北”形式不正确。



点选



鸟瞰/ WASD +



命令行　　超文本

图 2：用于移动/接听游戏的四个不同的用户界面。

相应地，影响游戏设计的一个重要决定是（a）附加能力的可发现性（例如，玩家是否可以确定“检查”是一个有意义的动词， 而无需具备游戏类型的素养？）和（b）用户界面无法形成毫无意义的表达的程度。例如，在超文本界面中，所有链接都指向某个地方，因此玩家可以形成的每一个意图（即单击页面上的链接） 都会从游戏中获得有效的响应，而在命令行界面中键入“fnord”可以被解析器识别，但是对于“弗洛德”不是名词的游戏毫无意义。关于这两个（相关）维度的决策将确定学习语言的程度，这是一个探索性但有时令人沮丧的过程，是游戏的主要挑战。

5 游戏机制作为操作语义

考虑图的右侧 1: 游戏解析并解释了玩家意图的明确语法，该语法使游戏状态前进（如图示），或者游戏状态无法按照玩家的意图进行前进，游戏会以某种方式向玩家发出信号（未显示）。我们使用以下 BNF 定义对移动运动员的意图进行响应，分别为：

回应::=成功|失败

数字 1 显示了玩家“拿走烧瓶”的意图（正式而言，语法为“take”）导致游戏世界将此意图作为成功动作执行并相应地做出响应的情况“采取。”正式地，我们将此响应建模为“成功”，如上所述。同样，如果无法拿走长颈瓶（例如，长颈瓶不在玩家附近，或者已经在玩家的手中，等等），则游戏将失败进行响应。

就像玩家的意图一样，它既可以作为原始输入也可以作为形式语法而存在，每个形式上的响应都可以通过各种方式（例如，文字，图片或声音）作为原始输出来传达。在真实游戏中，游戏对玩家的反馈通常有两个两个级别：通过游戏世界的反馈和在游戏世界之外的反馈（例如，带有错误，指导或建议的弹出消息）。为简单起见，举动给出了游戏状态之外的反馈，例如，弹出消息。

为了捕获玩家意图（作为语法）和游戏响应（作为语法） 之间的形式关系，我们引入了四位游戏步骤关系：

（G1;意图）-→（G2;分别）

这种关系形式化了图形右侧的动态行为 1. 它包括四个部分：初始游戏状态 G1，玩家意图，结果游戏状态 G2 和游戏响应。

按照 PL 形式主义的标准，我们给出将这种关系定义为归纳推理规则的规则，每个规则都可以理解为逻辑推理。也就是说，给定规则顶部的前提的证据，我们可以得出规则底部的结论。

第一条规则将图 RHS 中所示的案例形式化 1。第二条规则将相反的结果正式化，即无法拿瓶。值得注意的是，第一个规则有两个前提：要被玩家接受，足以表明在当前游戏状态 G1 中，魔瓶位于玩家附近（第一前提），并且存在一个高级游戏状态 G2 玩家拿下此烧瓶的结果（第二前提）。在第二条规则中，只有一个游戏状态 G，因为不能拿走长颈瓶，因此游

戏状态不会改变。

就像玩家意图和游戏响应的语法一样，这些规则也是明确的。因此，我们将这些规则视为一种数学定义，并带有用于构建有关游戏机制的正式（和非正式）证明的相关策略。

例如，我们可以正式陈述并尝试证明，对于所有玩家意图和游戏世界 G1，存在相应的游戏世界 G2 和游戏响应。也就是说，以下猜想的陈述：

∀G1，意图。　　∃G2，分别。（G1;意图）-→（G2;分别）

使用标准的 PL 技术，这种猜想的证明产生了一种抽象算法，该算法通过分析当前游戏状态和玩家意图的每种可能情况来实现游戏机制。确实，这恰恰是证明游戏实施完成所需的推理（即，不存在将导致游戏进入不确定状态的状态和输入）。

关于此完整性的推理涉及对每个规则何时适用的推理。例如，成功取得玩家意图的规则需要两个前提：G1 玩家靠近烧瓶，以及玩家 Take G1，烧瓶 G2。首先是对涉及命题玩家 1- 近烧瓶的游戏世界的逻辑判断，(这可能是) ＝正确的，也可能不是正确的，但它是可计算的。第二种是一种语义函数，它将游戏状态转换为玩家获取给定对象的状态。通常，此函数可能是未定义的，例如，如果参数不满足该函数的前提条件。例如，功能 playerTake 的先决条件可能是该对象尚未由玩家拥有。在这种情况下，为了证明游戏机制不会“卡住”，我们必须证明 playerNear flask 暗示玩家尚未拥有该烧瓶。（否则，我们应该添加另一个规则来处理玩家打算拿走烧瓶但已经拥有烧瓶的情况）。编程语言语义的设计过程通常包括尝试编写示例并证明定理，然后失败。这些经验有助于对语言定义进行系统的修改。

在下一节中，我们将通过以下方式进一步细化意图和语义介绍上下文的概念。

6 上下文接口作为类型系统

有关接口语法的决策在一定程度上会限制或限制增强玩家形成失败意图的意图的能力，例如穿过墙壁或拿走没有爆炸的物体。但是有时候，话语是否有意义将取决于运行时游戏的状态， 并且可以认为是一种不同的话语。问题来自它是否格式正确。例如，是否可以使用烧瓶取决于烧瓶是否存在，但是如果烧瓶是游戏中某个对象的对象，则必须将此命令视为格式正确的语法，并释放其与运行时游戏集成的失败机械的环境（操作语义）。

但是，某些用户界面仍然以依赖于当前游戏状态的方式限制已识别话语的集合。考虑一个点击界面，当鼠标悬停在可交互对象上时，它将鼠标的形状更改为一只手，并且仅在此状态下才识别点击。或者，考虑超文本界面，该界面仅识别对当前页面中可用链接的单击。仅向玩家提供说出在这方面“有意义”的那些话语的选项，这对应于用于编程语言的强大的静态类型系统。

类型系统通常通过定义表达式 e 和上下文 Γ 之间的关系来形式化。上下文是表达式有效或类型正确的一组特定情况。通常，这些情 况与程序中的变量有关。例如，只有 x 是数字时，程序表达式 x　+ 3 才被正确键入。“ x 是数字”是将包含在上下文中的事实的示例。它的类型正确可以表示为 x：num x + 3 ok。

在举棋游戏中，我们可以在上下文中包括游戏状态的各个方面， 例如玩家的位置和世界上各个房间之间的邻接映射。我们可能包 括的用于编纂“只有当前的事物是可以采取的”规则的打字规则。

然后，我们需要定义具体游戏状态 G 和这些状态 Γ 上的抽象条件之间的关系。我们可以将这种关系写为 G：Γ 。将这些规则整理后，我们可以完善“游戏完整性”猜想，以仅处理类型正确的话语：

∀G1，意图。 （G1：Γ）∧（Γ 1-意向确定）=⇒∃G2，分别 （G1;意图）-→（G2;分别）

这几乎是我们想要了解的游戏机制。但是，我们希望在游戏进行过程中迭代地应用这种推理，以便接下来就从游戏状态 G2 转到另一个可能不同的游戏状态 G3 的玩家意图进行推理。但是玩家意图的哪个上下文描述了状态 G2？

为了使这种推理起作用，我们通常需要更新原始上下文Γ，可能会更改其假设，并创建Γ,。我们写Γ Γ,表示Γ,以明确定义的方式成功了Γ。给定状态 G1 和意图与假设Γ的上下文一致，我们希望证明存在与新游戏状态 G2 一致的后续上下文Γ,：

∀G1，意图。（G1：Γ）∧（Γ 1-意图确定）)=⇒∃Γ,G2，分（G1;意图）-→（G 分别）

该声明与声音类型系统的编程语言的通, 常进度声明, 非常吻合。

7 播放轨迹为直线程序

如果我们将游戏界面的类比视为编程语言，自然就会出现问题， 用这种编程语言编写的程序是什么？我们至少要考虑将单个的， 原子的玩家动作视为完整的程序；前面的文字提供了这样的说明。但是典型的程序长于一行，这意味着用一种游戏语言来排序多个 动作意味着什么？

在命令式编程语言的典型解释中，我们介绍了一个排序运算符； 其中，如果 c1 和 c2 是该语言的命令，则 c1;c2 也是命令。这样的 命令的操作语义涉及状态 σ 上转换的组成：

$$(\sigma \ ; c1) \longrightarrow \sigma 1 \ (\sigma 1; c2) \longrightarrow \sigma 2 \ (\sigma \ ; (c1; c2)) \longrightarrow \sigma 2$$

但是，交互式软件通过将程序响应作为组件引入，使此帐户更加 复杂。代替顺序地发布任意命令，玩家可以等待响应或与他们的 决定并行地处理响应。在这方面，玩家的“编程”活动比起传统的 节目创作，更类似于现场直播。代码的执行与作者身份同时发生， 将这两个活动交织在一起。如果我们考虑在发出每个单独的命令 后在游戏循环中进行往返，那么我们得到的是类似于游戏轨迹的 程序概念：在游戏过程中玩家行为和游戏响应的日志

玩家：向北游戏：失败玩家：向烧瓶游戏：成功玩家：向南游戏：成功

根据我们内部机制模型的丰富程度，此游戏轨迹可能包含有关内 部状态变化的有用信息，这些内部状态变化与玩家动作的先决条 件和效果有关。但是要注意的主要重要一点是，尽管此处使用了 非正式的语法来表示它们，但这些痕迹并不包含玩家直接输入的 文字字符串，也不是游戏程序员作为日志信息添加的文字字符串， 它们是带有抽象的结构化术语可以被视为与任何程序相同的解释 和分析形式技术的语法。而且此语法是与游戏相关的交互的高级 别，而不是硬件输入和引擎代码的级别。

为了进行分析，学术界和游戏行业的研究人员最近都对玩游戏跟踪数据越来越感兴趣，例如了解他们的玩家如何与游戏的不同组件进行交互以及通过支持更新的内容对这些信息做出响应。玩家兴趣.在大多数情况下，这些跟踪数据是通过遥测或其他间接方式（例如游戏变量监控）收集的，之后必须对其意义进行分析.最近，已经提出了可以通过逻辑查询进行分析的结构化跟踪项系统，将支持自动化的能力识别为收益。

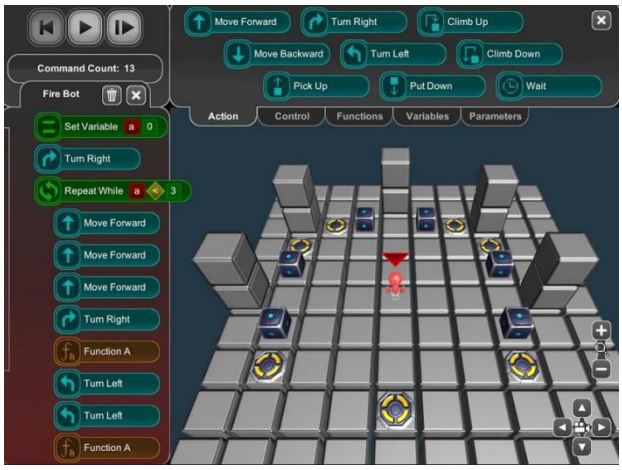


图 3：BOTS 的屏幕快照，这是一款教育性游戏，玩家在其中编写程序来指导玩家头像。

按照设计意图进行测试。我们的 PL 类比支持这条查询线，并保证进一步的比较和合作。

8 普通课程的球员技能

尽管考虑到“直线”轨迹可能在玩家分析中具有一定的实用性，但 随着程序结构的正式化，游戏交互的令人兴奋的前景是对可能执 行复杂任务的动作的参数化序列进行编码的可能性。毕竟，使用 丰富的玩家动作语言的游戏提供了探索性和创造性游戏的模式： 请考虑在 Minecraft 中制作物品，在 Zork，或在 Factorio，位于星露谷或 Mini Metro。这些活动中的每一项都要求玩家理解一个复杂的系统，并构建多步动 作序列以完成特定任务。从玩家的角度来看，这些计划是根 据较高级别的活动（例如种植农作物或构建新工具）构建的，而这些活动本身都是根据较低级别的游戏意图语言构建的。

正如我们已经将其形式化的那样，一种语言为我们提供了可以用来构造这些序列的原子块，例如乐高积木可以用来构造房屋或宇宙飞船的可重构组件。语言设计中的组合性是一个原则，我们可以从复合结构（例如序列）的每个部分（例如动作）的含义和行为以及它们如何组合（例如，组合）的含义来理解复合结构

（例如序列）的含义和行为。一个接一个地执行，或并行执行）。在本节中，我们将介绍如何使用更复杂的播放器语言版本编写的 程序来理解播放器技能。

此类程序可以集成到游戏的机制中，以便玩家明确编写此类程序，例如在 BOTS 游戏中，这是一个交互式编程导师，要求玩家编写小型命令式程序，以在虚拟机中引导虚拟形象。

图 4：Stardew Valley 的屏幕截图，显示了玩家的农场，库存和头像。

世界（见图 3），或 Cube Composer2，其中玩家编写功能性程序来解决难题。但是，就目前而言，我们主要打算将玩家技能作为一种概念工具。

示例：星露谷

我们最初的{ 举动以示例} 为例，它太简单了，难以制作出复杂程序的真正拼写示例，因此在这里，为了考虑玩家技能，我们研究了 Stardew Valley 及其游戏语言。在星露谷，玩家拥有一个清单，可以与世界进行各种互动，首先是使用多种耕作工具（斧头，头，大镰刀，镐），这些工具会与周围环境中的资源进行不同的处理；其中大多数包括提取一些资源（木材，石材，纤维等），这些资源本身会进入玩家的库存， 并可以用于与游戏世界的进一步互动。屏幕上显示了玩家的头像，由 WASD 移动。玩家与非玩家角色（NPC），可以通过其 购买新物品的界面（商店）以及包括钓鱼的迷你游戏（鱼也 可以高价出售）之间还存在上下文相关的交互。见图 4 查看游戏的典型玩家视角。

尽管完整介绍了该游戏提供给玩家的语言，但本文不包括 该游戏提供的代表性动作和能力示例，可用于构建玩家技能。其中包括：在世界“房间”之内和之间进行定向的化身运动， 点击和单击操作以选择一个人的清单中的项目以及与房间中 的实体进行交互。

玩家头像必须位于实体附近，玩家才能与其互动。然后，他们可以通过左键单击或右键单击将当前选择的清单项目应用于世界上的实体，这将根据实体类型执行某些操作，例如，

打开门和箱子，打开角色说话并收集物品物品转移到玩家的库存中。我们将此最后一个动作称为询问。我们还注意到，就我们的示例而言， 朝向实体的移动和屏幕外的移动（朝向另一个房间）是唯一有意义的，截然不同的移动类型，我们将其称为近距离移动和屏幕外移动。这些动作产生以下结果句法：

意图:: =选择（项目）| 申请（实体）|查询（实体）|靠近（实体）|移离屏幕（方向）

我们将项目和实体的定义保留为抽象，但是我们可以想象它只是将世界上所有可能的项目和实体作为终端符号列出。从这些原子输入中，我们可以开始构建游戏中最频繁执行的更高级别的动作-耕种土地，播种种子，与 NPC 交谈等等。可以为这些代码块分配名称，例如要在许多上下文中调用的函数：

行动直到=选择头；move_near hard_ground;申请 hard_ground

行动植物=选择种子；move_nelled_ground 附近;套用耕地地雷行动=选择镐 move_near rock;应用摇滚动作演讲= move_near npc;查询 npc

动作 enter_shop = move_near shop;查询门（店铺）

反过来，这些更大技能的分子可以组合起来以完成特定任务或完成任务。

将程序作为技能和策略进行分支注意，我们对动作采取的是天真排序，没有考虑游戏的响应。这种描述球员技能的方法没有考虑失败尝试的可能性，例如当当前屏幕上没有岩石时尝试进行挖掘。可以简单地为这些操作序列赋予语义，这些语义将线程失败贯穿整个程序-如果我们对任何操作都失败，则整个复合操作都会失败。

但是，如果我们考虑处理失败的可能性，日常编程乃至游戏性的共同特征，则可以进一步描述强大的玩家技能和策略。回忆一下我们简化的游戏响应语言，其中包括成功和失败两种可能性。我们可以在我们的语言中引入一个案例构造，以作为程序的一个独立分支来处理所有这些可能性：

地雷=响应=选择镐案例（响应）： 成功=> {响应'= move_near rock;案例（响应）： 成功=>应用摇滚； |失败=>失败;}|失败=>失败;

但是，为了避免处理每个可能的动作失败，一种更好的方法是将每个动作成功完成所需的世界资源显式指示为参数。例如，挖掘的总体动作定义将要求玩家可以使用镐作为动作的前提条件。

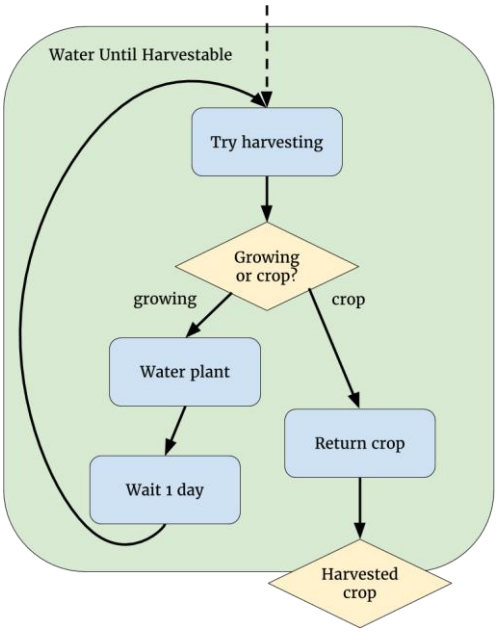


图 5：浇水直到收获。

库存和一块石头在同一房间。选择镐和在岩石附近移动的动作将取决于这些资源，并且游戏响应语言可以包括它保证作为输出的资源。然后，我们可以使用更简单的符号来编写程序， 该符号引用适当类型的资源依赖关系（使用符号 resource：type）：

地雷行动（p：pickaxe，r：rock）=选择 p;move_near r;申请：矿物

这种表示法与分支案例构造一起扩展到包括游戏世界中的不确定性（例如 Stardew Valley 中的钓鱼小游戏）：游戏最终总是告诉玩家某物正在拖拉她的路线，但在某些时候它是鱼，而其余时间则是垃圾。这些构造也可以解释不完整的球员心理模型，例如知道一个人必须日复一日地浇灌种子才能生长庄稼，却不知道多少次。

下面，我们提供一种符号来说明玩家技能的这些方面：do ... recv ...符号表示命令，然后将响应绑定到模式或结构化的变量集，然后可以进行案例分析. .我们的第一个例子是给农作物浇水直到收获为止：

动作 water_until_harvestable [t]（p：种植（t））：作物（t）=做 try_harvest（p）recv < 结果：作物（t）+生长（t）>。案件结果 c:crop(t) => c|g：生长（t）=>水（g）;等待（天）; try_harvest(g)

见图 5 该代码的控制流程图。

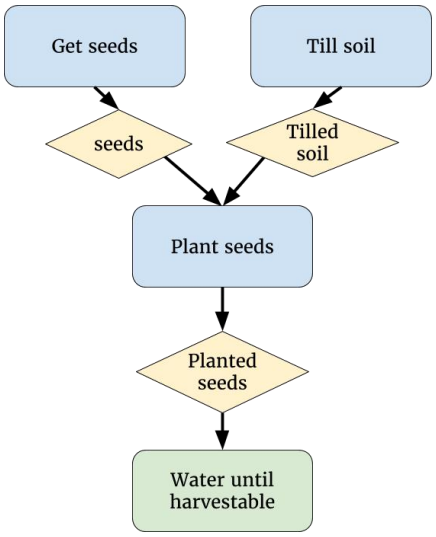


图 6：种庄稼。

下一个示例显示了||并行构造，它可用于构成具有不同依赖性的动作，以及动作定义如何通过将资源依赖性作为参数通过线程来使用其他动作定义：

有趣的 grow_crop [t：作物类型]（s：土壤，w：浇水可以）：作物（t）=做 get_seeds(t) || 耕种土壤 recv <s：种子（t），g：耕种土壤>。做植物（克）recv <p：种植（t）>。water_until_harvestable（p）

见图 6 该代码的控制流程图。

9 讨论区

建立了玩家语言的语法和语义词汇后，我们现在可以重新介绍引言中提到的该帐户的潜在好处，并对其进行更详细的讨论。

9.1 组成比赛轨迹，球员技巧

编程语言帐户提供的主要功能之一是组合性：一种用于根据 复杂工件的含义来理解其含义的系统。看游戏的地方有两个：

结构化的播放痕迹。使用形式化的游戏语言，操作语义所描述的过渡系统上的步骤序列形成了一个数学工件，与仅从屏幕或输入设备记录中获得的结果相比，该数学工件要进行更

深入的分析。例如，我们可以进行因果分析，询问跟踪数据的“为什么”查询，例如“为什么是玩家是否能够在击败老板之前解锁门？”，以及过滤所需属性的 轨迹：“向我显示一个播放轨迹，其中玩家使用了除火炬之外的其他东西照亮房间。”最近的 PlaySpecs 项目表示有兴趣以这种方式来创建跟踪以支持这种查询。

玩家技能作为程序。虽然游戏轨迹可以解释为直线程序，但更有趣的是玩家动作中的潜在结构的想法，例如将多个低级别的游戏动作组合成更高级别的技能，紧随其后。“分块”的想法。我们将此想法映射到带有参数的编程语言函数中， 以概括状态空间的可能性（例如：红色键打开了红色门，因此对于所有颜色 C，C 键都会打开一个 C 门）。诸如案例分析这样的其他推理形式也可以自然地表示为编程语言构造，例如案例分析以处理不可预测的游戏行为以及重复执行一项操作直到条件成立。

9.2 输入与机制之间的抽象边界

正式的游戏界面为我们提供了探索相同底层机制的替代界面的工具，而无需在不同的图形界面框架之间移植游戏逻辑。例如，交互式小说社区一直在探索传统的“解析器与超文本”二分法的替代方案，以展示基于文本的游戏和交互式故事世界。世界的基本机制，地图和叙述之间的抽象边界，以及用于与之交互的视图和输入机制，可以为研究支持用户以文本形式传达的世界心理模型的用户界面打开大门。

9.3 实现共同创造的游戏

最终，玩家行为的 PL 公式以及适当的合成运算符（并行和顺序合成，分支和传递资源依赖项）为游戏环境提供了“免费的脚本语言”。这种语言可以用来测试游戏，可以像在 BOTS 中那样作为游戏机制提供，也可以作为对游戏机械机制的可选增强，以便为游戏世界添加或添加新内容。尤其是在网络游戏环境中，例如多用户域，大型多人在线游戏以及诸如《第二人生》之类的社交空间中，玩家不仅可以对自己的虚拟化身进行编程，而且还可以对部分游戏世界本身进行编程，从而为创新性和协作性游戏带来了新的机遇。我们的框架提出了一种新的方法，以一种自然地从游戏的现有机制和界面衍生而来的方式为玩家设计这些能力。本着庆祝玩家的精神，即今年的会议主题，我们希望使玩家成为自己游戏体验的共同设计师。

9.4 未来的工作

在未来的工作中，我们打算构建用于实现游戏语言设计的软件，并尝试使用基于协议的游戏 AI 作为这些语言的程序开发。在另一个方向上，我们旨在创新游戏之外的 PL 设计，例如用于实时编程的读评估打印循环（REPL），其中包括由游戏玩 法以及分布式产生的快速反馈循环并发系统可能会从此处提出的基于协议的方法中受益。

10 结论

我们提出了玩家意图语言，这是将编程语言设计原理应用于玩家游戏界面设计的系统框架。我们通过语法（即玩家意图），类型系统（即上下文接口）和操作语义（即游戏机制） 的简单示例来定义此框架。我们展示了如何将此框架应用于 玩家游戏界面自然地引起游戏轨迹（作为直线程序）和玩家 技能（作为具有分支和递归的通用程序）的形式化概念。通过定义玩家意图语言，游戏设计概念成为正式的研究对象， 从而使现有的 PL 方法学能够为设计过程提供信息和指导。

11 原文出处

[1]Martens C,Hammer M A .Languages of Play: Towards semantic foundations for game interfaces[J].2017.