US 20130084999A1

(19) **United States**

(12) **Patent Application Publication**

Costa

(10) **Pub. No.: US 2013/0084999 A1**

(43) **Pub. Date: Apr. 4, 2013**

(54) **GAME CENTERED ON BUILDING NONTRIVIAL COMPUTER PROGRAMS**

(71) Applicant: **Jason Churchill Costa**, New York, NY (US)

(72) Inventor: **Jason Churchill Costa**, New York, NY (US)

(21) Appl. No.: **13/645,507**

(22) Filed: **Oct. 4, 2012**

Related U.S. Application Data

(60) Provisional application No. 61/542,812, filed on Oct. 4, 2011.

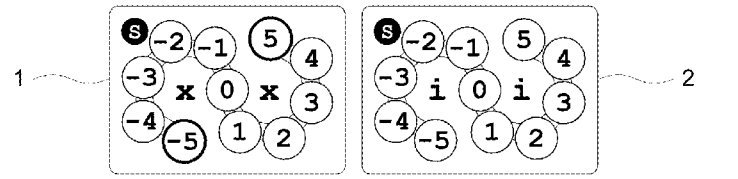
Publication Classification

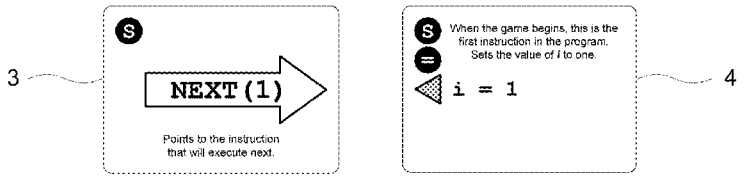
(51) **Int. Cl.**
A63F 9/24 (2006.01)

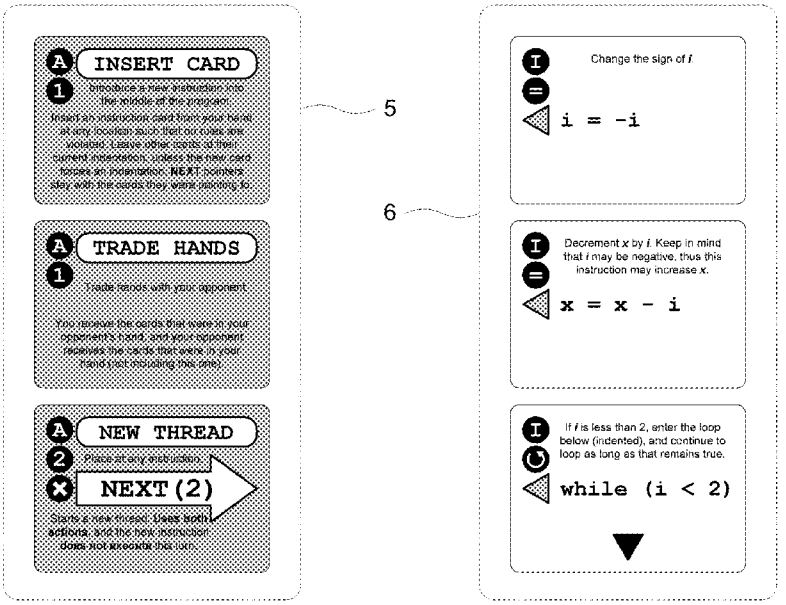
(52) **U.S. Cl.**
USPC **463/43**

(57) **ABSTRACT**

A game in which players modify a nontrivial computer program. Players with competing objectives are given the option of appending instructions (**6**) to the program, as well as performing special actions (**5**) such as inserting, deleting, or moving instructions. As the players modify the program, the program executes, modifying variables and potentially moving players closer to or further from their objectives. In addition to serving as a source of enjoyment, the game may be used in an educational context to teach and reinforce computer programming concepts such as conditional branching, looping, and multithreading.







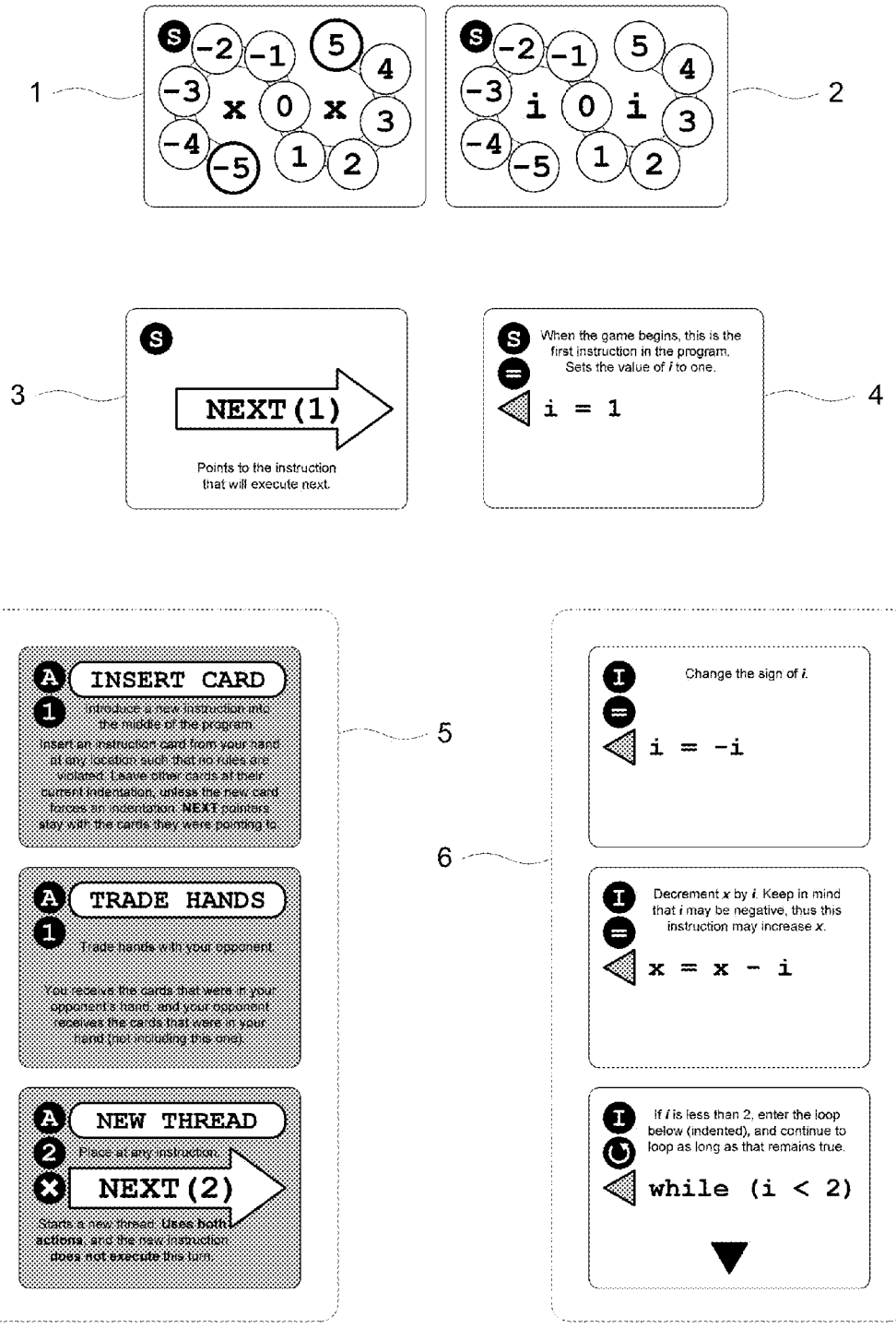

1
S
-2
-1
5
4
-3
x
0
x
3
-4
-5
1
2
2
S
-2
-1
5
4
-3
i
0
i
3
-4
-5
1
2
3
S
NEXT (1)
Points to the instruction that will execute next.
4
S
=
When the game begins, this is the first instruction in the program. Sets the value of i to one.
i = 1
5
A
1
INSERT CARD
Introduce a new instruction into the middle of the program.
Insert an instruction card from your hand at any location such that no rules are violated. Leave other cards at their current indentation, unless the new card forces an indentation. NEXT pointers stay with the cards they were pointing to.
A
1
TRADE HANDS
Trade hands with your opponent.
You receive the cards that were in your opponent's hand, and your opponent receives the cards that were in your hand (not including this one).
A
2
NEW THREAD
Place at any instruction.
NEXT (2)
Starts a new thread. Uses both actions, and the new instruction does not execute this turn.
6
I
=
Change the sign of i.
i = -i
I
=
Decrement x by i. Keep in mind that i may be negative, thus this instruction may increase x.
x = x - i
I
If i is less than 2, enter the loop below (indented), and continue to loop as long as that remains true.
while (i < 2)

FIG. 1

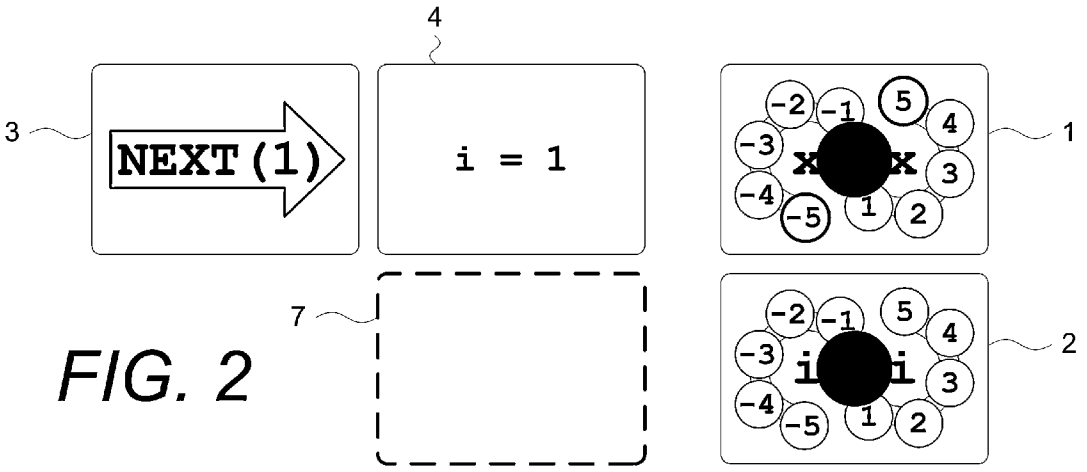


FIG. 2

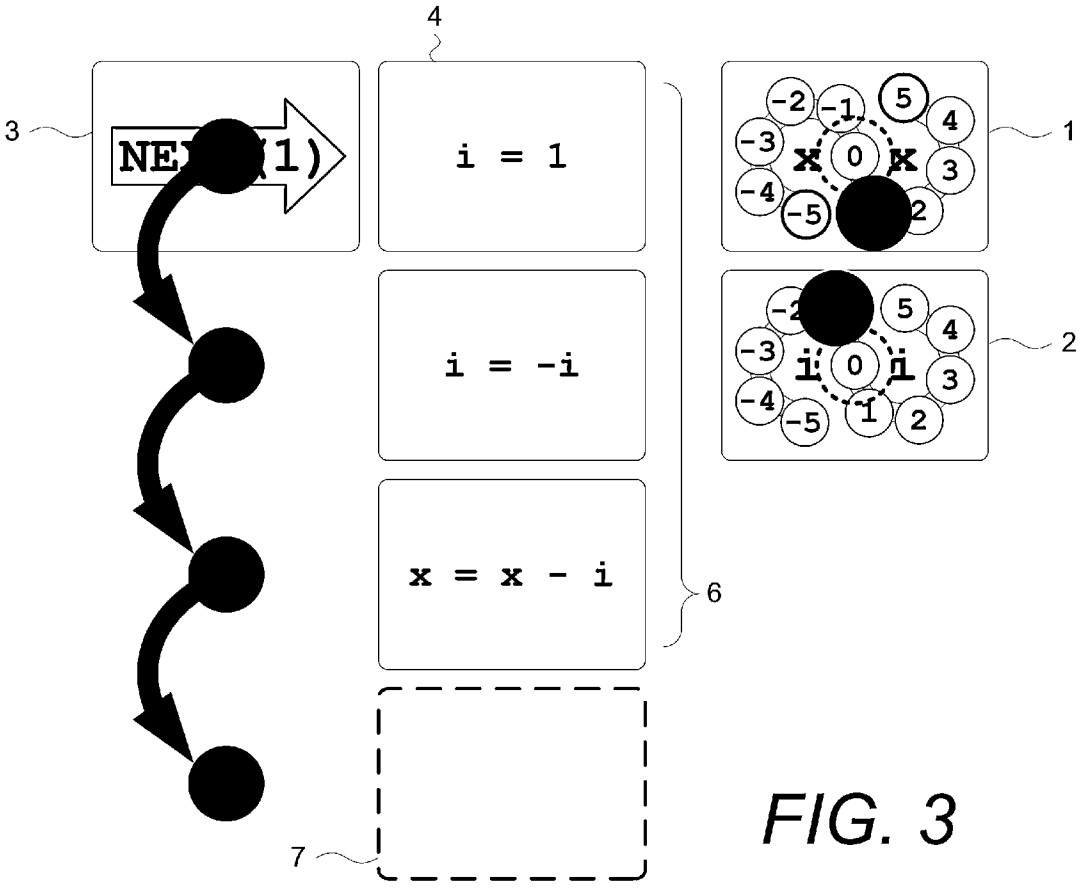


FIG. 3

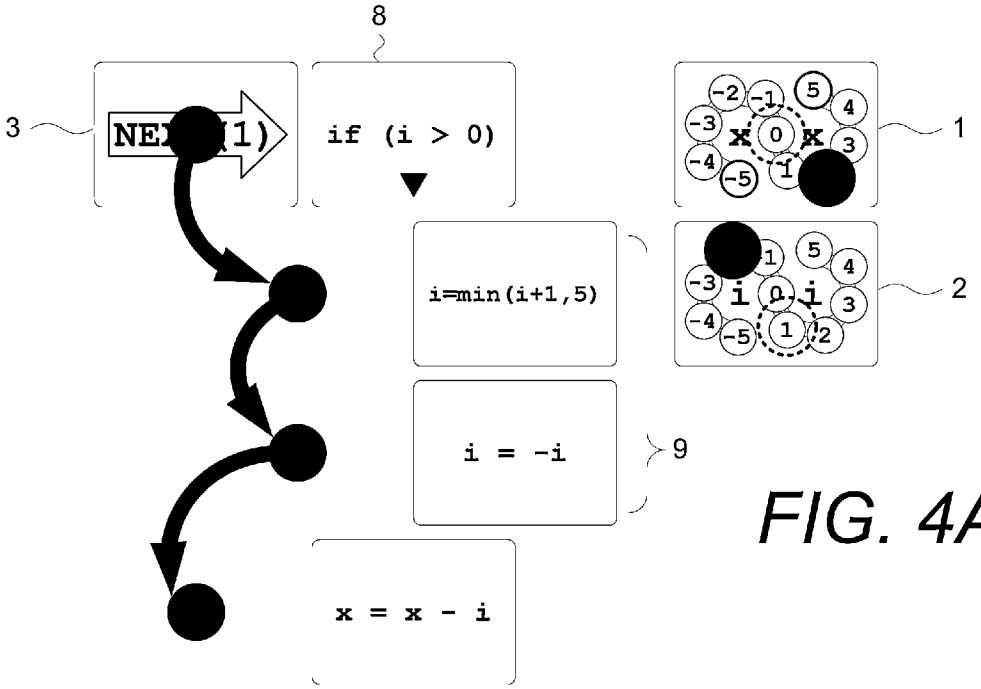


FIG. 4A

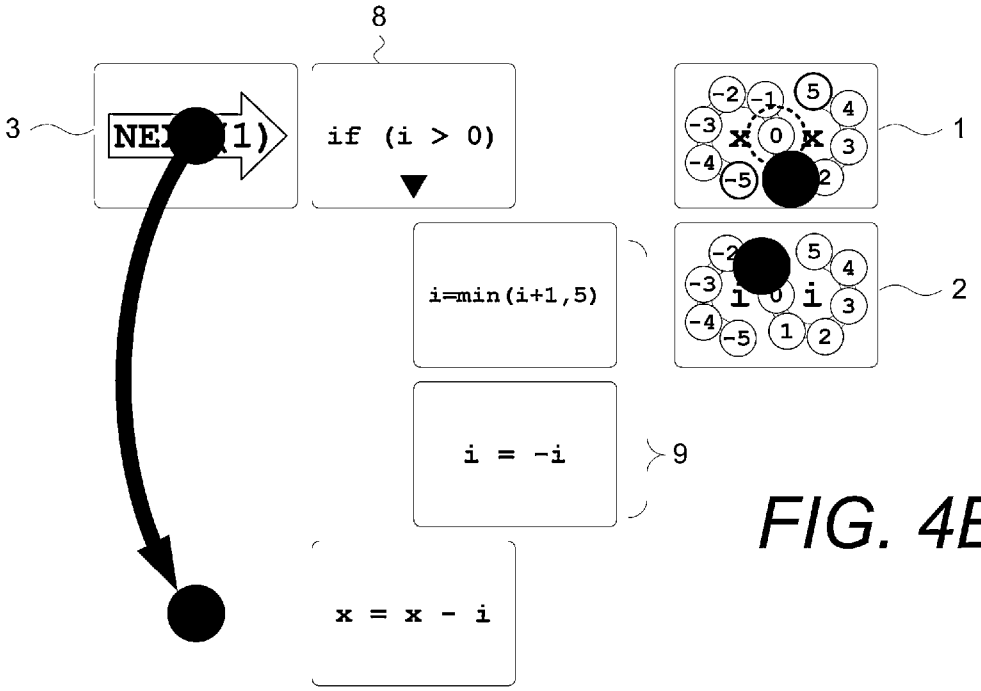


FIG. 4B

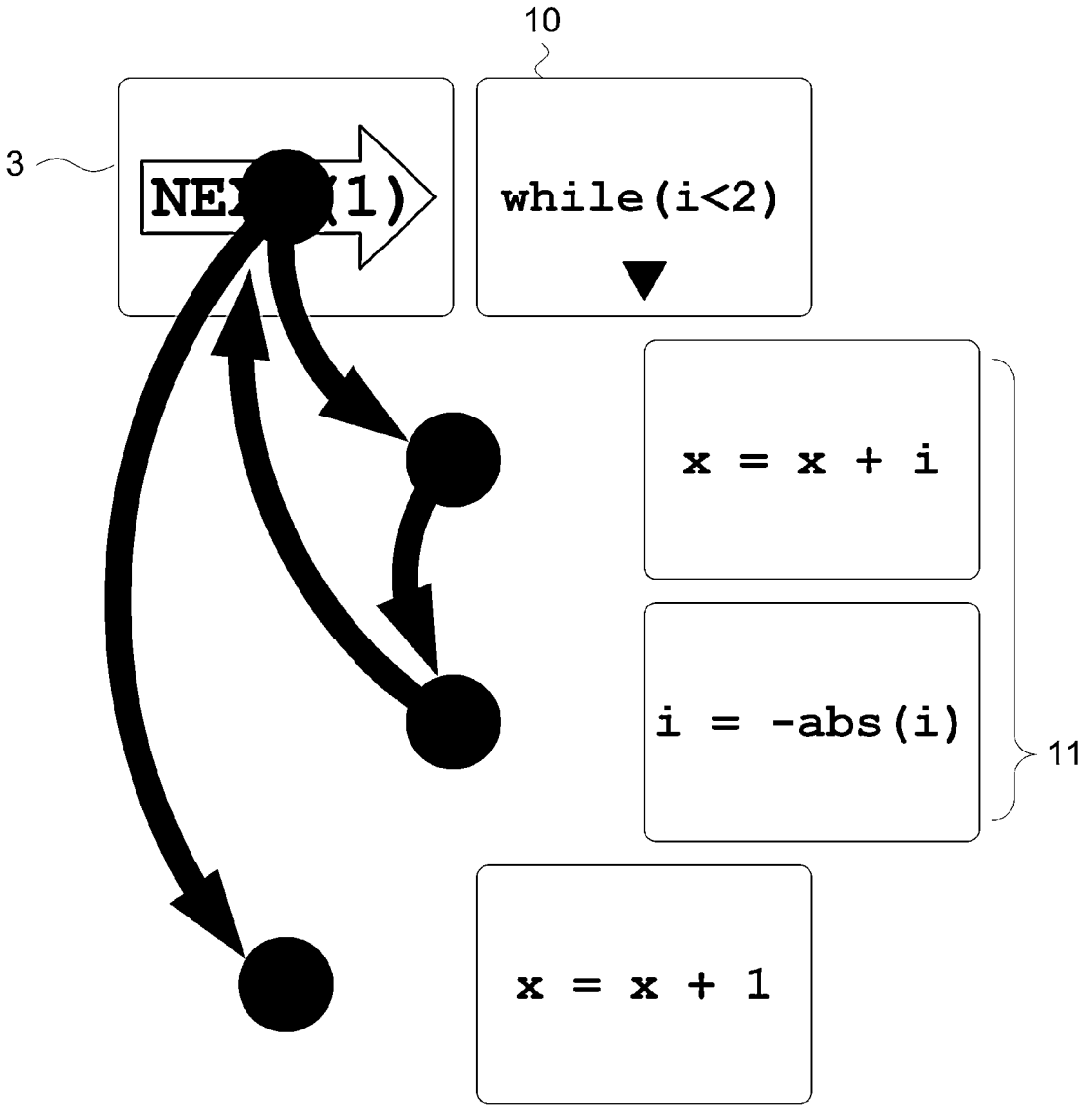


FIG. 5

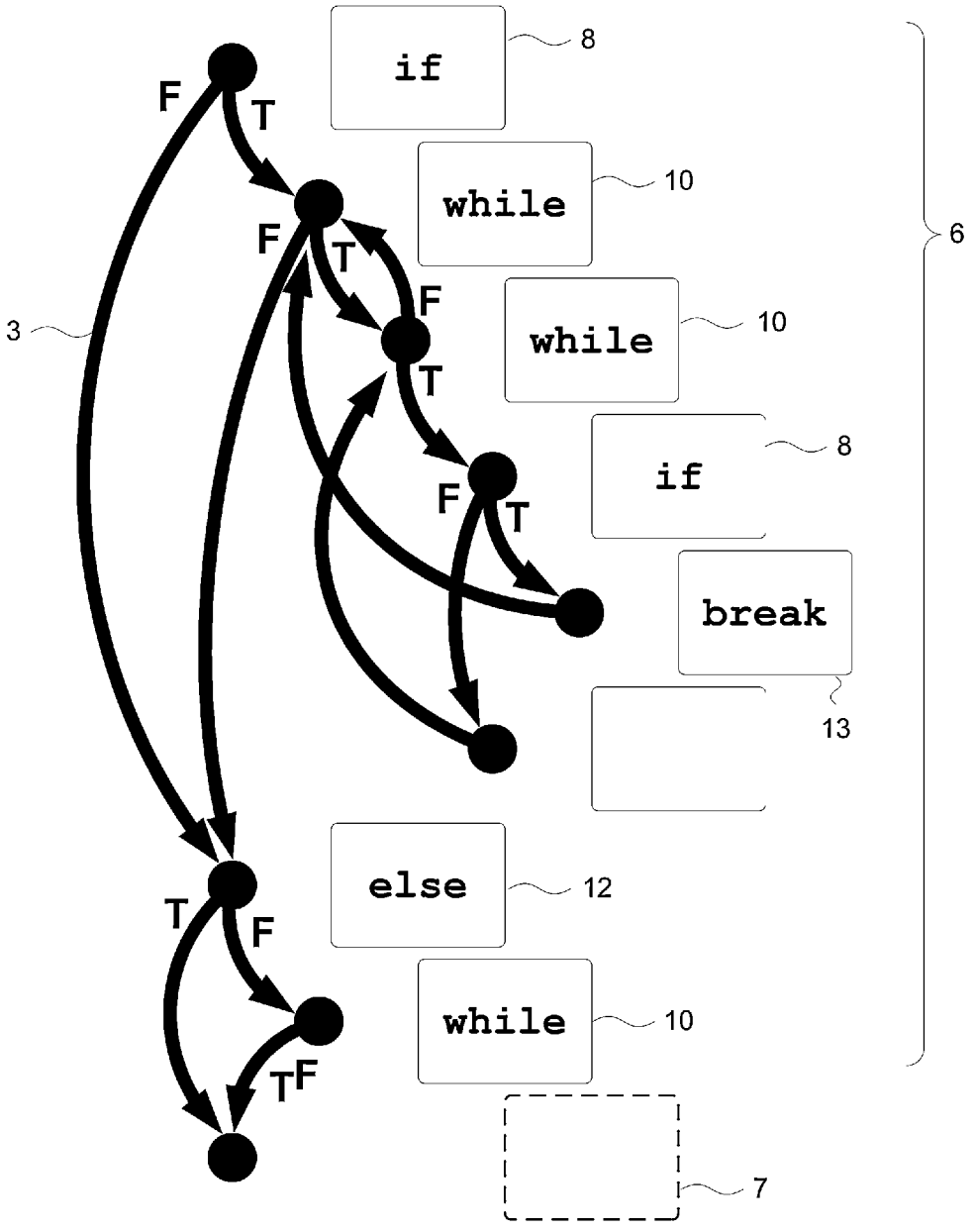

if
8
while
10
while
10
if
8
break
13
else
12
while
10
6
7
3
F
T
F
T
F
T
F
T
T
F
T
F

FIG. 6

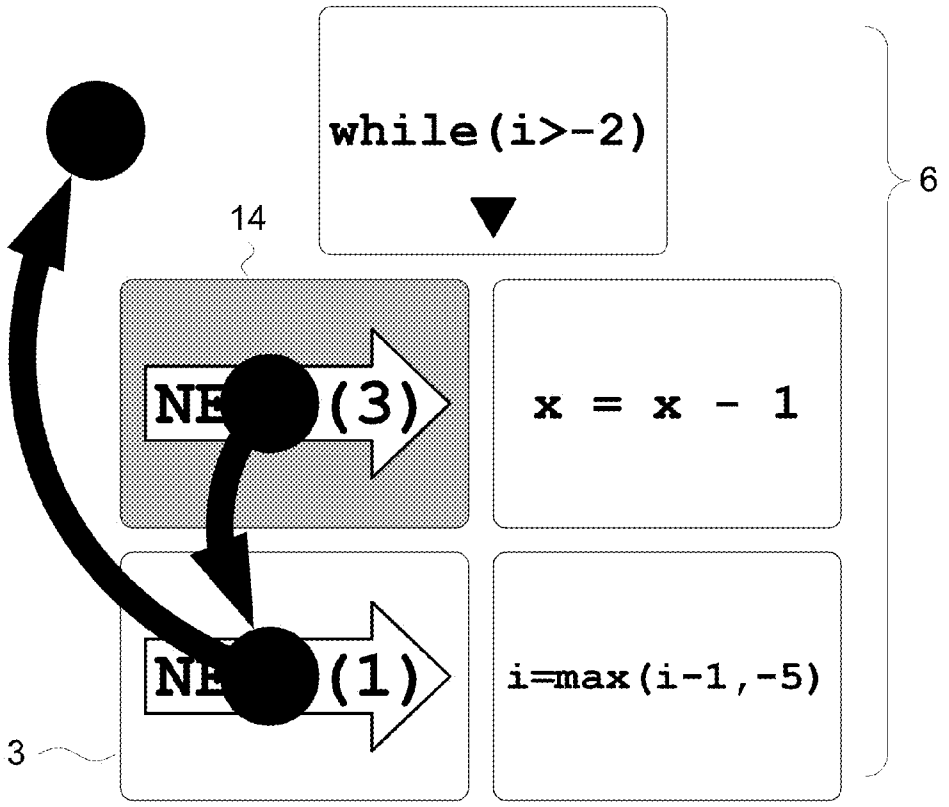


FIG. 7

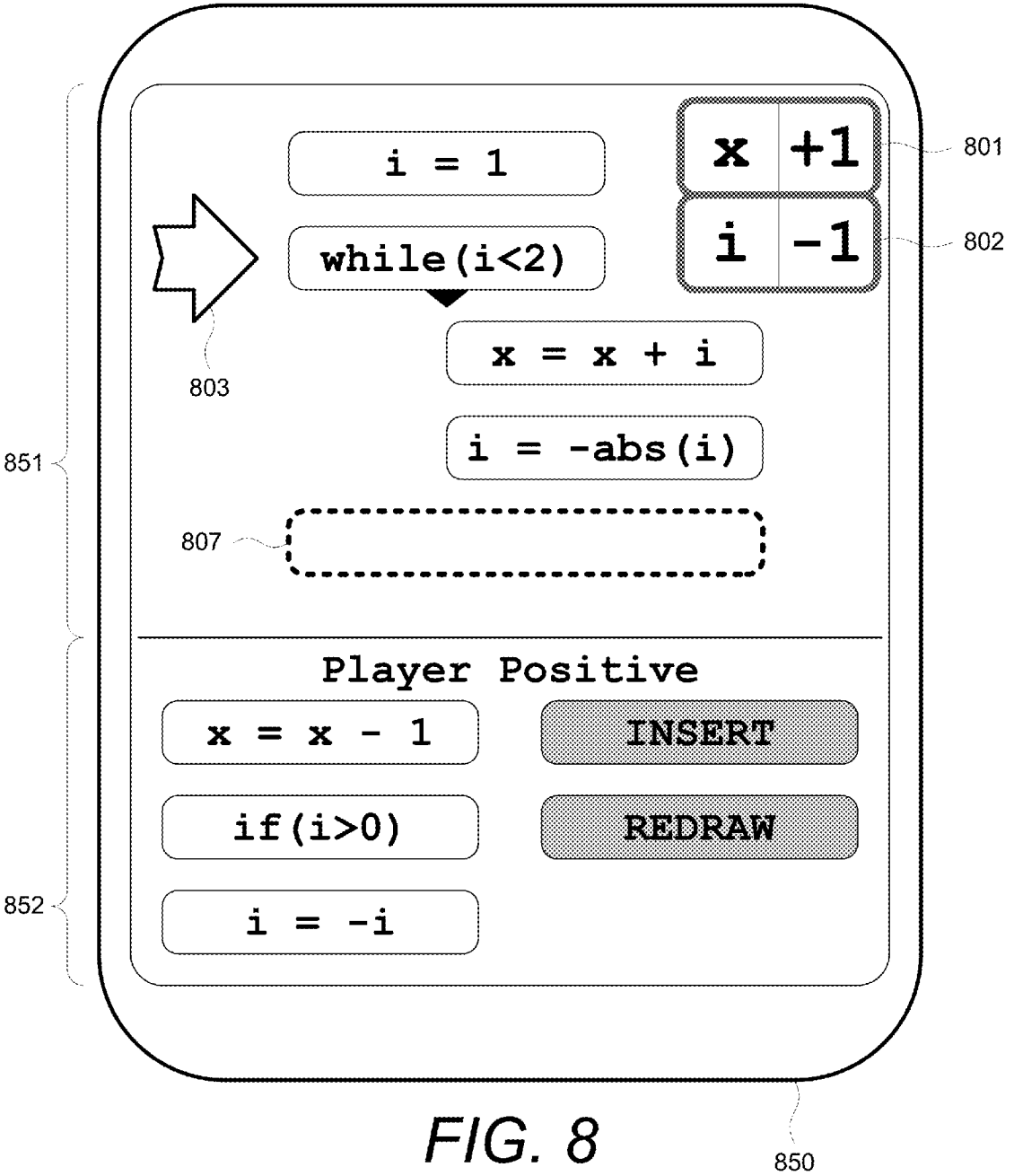

i = 1
while(i<2)
x = x + i
i = -abs(i)
x +1
i -1
801
802
803
807
851
Player Positive
x = x - 1
if(i>0)
i = -i
INSERT
REDRAW
852
850

FIG. 8

GAME CENTERED ON BUILDING NONTRIVIAL COMPUTER PROGRAMS

BACKGROUND

[0001] This invention relates to the field of games, and specifically that of games employing computer programming concepts. Some existing games employ language from the domain of computer programming and attempt to replicate some aspects of real life computer programming. This category of game can provide both enjoyment and educational opportunities. However, these games may not offer an experience that is as rich, engaging, and educational as possible.

[0002] U.S. Pat. No. 6,135,451 to Kholodov (2000) describes a game in which players advance through a computer program, but the program is fixed, and advancement through the program is dictated by a die roll.

[0003] U.S. Pat. No. 5,078,403 to Chernowski (1992) describes a game that uses words and phrases from computer programming in its naming conventions, such as "program", "storage area", "byte", and 'I/O'. However, the concept of a "program" that appears in this game is neither executable nor modifiable.

[0004] U.S. Pat. No. 4,258,922 to Landry (1981) describes a game in which operations are performed on binary digits. Again, no concept of a modifiable computer program exists.

[0005] The game RoboRally published by Wizards of the Coast (1994) allows players to create a simple program that controls the actions of a game piece representing a robot. While this game does feature user-modifiable computer programs, the programs are extremely limited in scope. They are of limited length (5 instructions), and are limited to simple linear execution. That is to say, no control flow concepts such as branching or looping are introduced.

SUMMARY

[0006] The game described herein is one in which users modify a computer program that executes as the users are modifying it. The program is represented by movable instructions (which could be implemented as cards or tiles in a physical embodiment) and can become arbitrarily complex, limited only by the number of instructions available. The program can employ programming constructs such as conditional branching, looping, and multithreading.

[0007] The players have competing objectives tied to the values of variables that are modified as the program executes. Each player attempts to modify the program, or otherwise alter the state in which it is executing, in order to accomplish his or her objective. In order to be successful at the game, players must think like a computer programmer and find ways to make the program accomplish the user's goal.

DRAWINGS

[0008] FIG. **1** depicts samples of cards or tiles used in a physical embodiment of the game.

[0009] FIG. **2** depicts a possible starting configuration for the game.

[0010] FIG. **3** depicts the execution of a simple sequence of instructions.

[0011] FIGS. **4**A and **4**B depict the execution of an "if" block when the relevant condition is true (**4**A) and when the relevant condition is false (**4**B).

[0012] FIG. **5** depicts the execution of a "while" block.

[0013] FIG. **6** depicts the composition of "if" and "while" blocks, creating a nested program structure.

[0014] FIG. **7** depicts the execution of multiple instruction pointers.

[0015] FIG. **8** depicts an electronic embodiment of the game.

DETAILED DESCRIPTION OF DRAWINGS

[0016] FIG. **1** shows elements of a physical embodiment of the game such as a card game. Variable counters (**1**, **2**) are used to update and display the current values of named variables. (In a card game embodiment, a small object such as a coin would be placed on each counter card to indicate the current value of the variable represented by the counter.) A next instruction pointer (**3**) is included in order to indicate the next instruction to be executed. (In a card game embodiment, the next instruction pointer would be positioned to literally point at the next instruction.) A dedicated first instruction (**4**) may be included with the game in order to simplify the process of starting a new game. Special actions (**5**) are tokens (cards in a card game embodiment) that allow the player in possession of the token to perform a special action that the player would ordinarily not be allowed to take. The three sample special actions in the diagram allow the bearer to insert an instruction into the middle of an existing program, trade hands with an opponent, and introduce a new next instruction pointer, respectively. Instructions (**6**) are the movable components that make up the computer programs that are central to the game. The three sample instructions have the effect of negating the variable "i", subtracting the value of the variable "i" from the variable "x", and beginning a while-loop conditional on the variable "i" being less than two, respectively.

[0017] FIG. **2** depicts a possible starting configuration for the game. Variable counters (**1**, **2**) are initialized to a starting value (which is the value zero in this embodiment), depicted as a large black dot. In a card game embodiment of the game, this would be accomplished by placing an object such as a coin on each variable counter card. The first instruction (**4**) is placed in the field of play to begin the program. The next instruction pointer (**3**) points to the first instruction to indicate that it is the next instruction to be executed. The empty space (**7**) below the program is where the next instruction will be added, unless a special action is used to add an instruction to a different location.

[0018] FIG. **3** depicts the execution of a simple, linear sequence of instructions (**6**). When the first instruction, bearing the text "i=1", is executed, the variable counter representing "i" (**2**) is set to the value of positive one, and the next instruction pointer (**3**) is moved to point to the next instruction. When the second instruction, bearing the text "i=−i", is executed, the variable counter representing "i" (**2**) is negated, meaning that its value is modified from positive one to negative one, and the next instruction pointer (**3**) is moved to point to the next instruction. When the third instruction, bearing the text "x=x−i", is executed, the value of the variable "i" is subtracted from the value of the variable "x", resulting in the counter representing "x" (**1**) being set to the value of positive one, and the next instruction pointer (**3**) is moved to point to the empty space below the last instruction (**7**). At this point, the next instruction pointer cannot advance until either an instruction is added to the program, or a special action is played which alters the location of the next instruction pointer.

[0019] FIG. **4**A depicts the layout of an "if" block, and the sequence of instructions executed when the relevant condition is true. An "if" instruction (**8**) is followed by an indented block of instructions (**9**), in this case consisting of two instructions. Additional instructions (not indented) may follow the "if" block, as is the case in this example. For the purposes of this example, assume that the variable counters representing "x" (**1**) and "i" (**2**) are initially set to zero and positive one, respectively. When the "if" instruction (**8**) is executed, the condition contained within parentheses, i.e. "i>0", is evaluated. Because the condition is true, the next instruction pointer (**3**) is moved to the indented block (**9**) and points to the instruction immediately under the "if" instruction. When the first indented instruction, bearing the text "i=min(i+1,5)", is executed, the variable counter representing "i" (**2**) is incremented from positive one to positive two, and the next instruction pointer (**3**) is moved to point to the next instruction. When the second indented instruction, bearing the text "i=−i", is executed, the variable counter representing "i" (**2**) is negated, meaning that its value is modified from positive two to negative two, and the next instruction pointer (**3**) is moved to point to the next instruction, i.e. the instruction below which is not indented. When the last instruction, bearing the text "x=x−i", is executed, the value of the variable "i" is subtracted from the value of the variable "x", resulting in the counter representing "x" (**1**) being set to the value of positive two.

[0020] FIG. **4**B depicts the same "if" block that appears in FIG. **4**A, but shows the sequence of instructions executed when the relevant condition is false. For the purposes of this example, assume that the variable counters representing "x" (**1**) and "i" (**2**) are initially set to zero and negative one, respectively. When the "if" instruction (**8**) is executed, the condition contained within parentheses, i.e. "i>0", is evaluated. Because the condition is false, the next instruction pointer (**3**) skips the indented block (**9**) and is moved to point to the next instruction at the same level of indentation as the "if" instruction (**8**). When that last instruction, bearing the text "x=x−i", is executed, the value of the variable "i" is subtracted from the value of the variable "x", resulting in the counter representing "x" (**1**) being set to the value of positive one.

[0021] FIG. **5** depicts the execution of a "while" block. A "while" instruction (**10**) is followed by an indented block of instructions (**11**), in this case consisting of two instructions. Additional instructions (not indented) may follow the "while" block, as is the case in this example. When the "while" instruction (**10**) is executed, the condition contained within parentheses, i.e. "i<2", is evaluated. As with an "if" instruction: if the condition is true, the next instruction pointer (**3**) is moved to the indented block (**11**) and points to the instruction immediately under the "while" instruction; if the condition is false, the next instruction pointer (**3**) skips the indented block (**11**) and is moved to point to the next instruction at the same level of indentation as the "while" instruction (**10**). Unlike an "if" block, however, when the last indented instruction is executed, the next instruction pointer (**3**) is moved back to point to the "while" instruction (**10**).

[0022] FIG. **6** depicts the composition of "if" and "while" blocks, creating a nested program structure. "If" instructions (**8**) and "while" instructions (**10**) require that the instruction below be indented by an additional level of indentation, regardless of the number of existing levels of indentation. There is no limit on the number of levels of indentation; both "if" instructions (**8**) and "while" instructions (**10**) can be included within indented "if" and "while" blocks. Also shown are an "else" instruction (**12**), which if present must appear at the same level of indentation as an "if" instruction (**8**), and a "break" instruction (**13**) which if present must appear within a "while" block. The movement of the next instruction pointer (**3**) is determined by composing the rules that apply to "if" and "while" blocks.

[0023] FIG. **7** depicts the execution of multiple instruction pointers. An embodiment may allow for the existence of multiple next instruction pointers. In this illustration, two next instruction pointers (**3**) and (**14**) have different priorities, indicated by a number appearing on each. In order to advance these pointers, firstly the higher priority next instruction pointer (**3**) would be handled: the instruction to which it pointed would be evaluated, and then it would move according to the rules for advancing pointers, in this case moving to a "while" instruction above the instruction just executed. Secondly the lower priority next instruction pointer (**14**) would be handled: the instruction to which it pointed would be evaluated, and then it would move according to the rules for advancing pointers, in this case moving to the instruction just below the instruction just executed.

[0024] FIG. **8** depicts an electronic embodiment of the game. A mobile device (**850**) displays the current program (**851**) and the current player's available instructions and special actions (**852**). Variable counters (**801**, **802**) display the current values of named variables. A next instruction pointer (**803**) indicates the next instruction to be executed. In an electronic implementation, the empty space below the existing program (**807**) can be explicitly visually represented.

REFERENCE NUMERALS

[0025] (**1**) variable counter (for the variable "x")
[0026] (**2**) variable counter (for the variable "i")
[0027] (**3**) next instruction pointer
[0028] (**4**) example first instruction used to start the game
[0029] (**5**) example special actions
[0030] (**6**) example instructions
[0031] (**7**) empty space below computer program
[0032] (**8**) "if" instruction
[0033] (**9**) block indented below "if" instruction
[0034] (**10**) "while" instruction
[0035] (**11**) block indented below "while" instruction
[0036] (**12**) "else" instruction
[0037] (**13**) "break" instruction
[0038] (**14**) secondary next instruction pointer
[0039] (**801**) variable counter (for the variable "x")
[0040] (**802**) variable counter (for the variable "i")
[0041] (**803**) next instruction pointer
[0042] (**807**) empty space below computer program
[0043] (**850**) mobile electronic device
[0044] (**851**) view of current computer program
[0045] (**852**) view of current player's available set of instructions and special actions

DESCRIPTION AND OPERATION OF FIRST EMBODIMENT

[0046] The description below explains one embodiment of the invention: a card game in which two competing players take turns modifying the same computer program until one of them achieves his or her objective. One player, "Positive", has the goal of setting the variable "x", represented by a variable

counter (**1**), to the value of positive five. The other player, "Negative", has the goal of setting the variable "x" to the value of negative five.

[0047] Types of Cards

[0048] This embodiment of the game is played with a special deck of cards, plus marker pieces (such as coins) placed on variable counter cards (**1**, **2**) in order to represent a variable's current value. The cards fall into three categories. It makes sense to use a different color scheme for the faces of each category, to make them easily distinguishable. The categories are described below:

[0049] Start Cards are the cards used to begin the game: variable counters (**1**, **2**), a next instruction pointer (**3**), and a first instruction (**4**). The first instruction, also an Instruction Card, is used to start the game.

[0050] Instruction Cards represent instructions that make up the common program. There are two types of instruction cards: Assignments, which modify the value of a variable, and Control Flow Instructions, which affect the movement of the next instruction pointer (**3**). FIG. **1** includes depictions of sample Instructions (**6**).

[0051] Special Action Cards allow the bearer to perform a special action such as modifying the program. The bearer must expend either one or two actions (explained below), depending on the card, in order to play it. FIG. **1** includes depictions of sample Special Action Cards (**5**).

[0052] Beginning the Game

[0053] The first instruction (**4**) (the Instruction Card bearing the text "i=1") is placed on the gaming table, and the next instruction pointer card (**3**) is placed to its left. As the game is played, additional Instruction Cards will be placed on the table below the first instruction, and this chain of cards will be referred to as "the program". The two variable counter cards, "x" (**1**) and "i" (**2**), are placed to the right of the first instruction (**4**), and a coin is placed at the zero position on each counter card to indicate the current value of that variable. All of the remaining cards (Instruction Cards and Special Action Cards) are shuffled, and each player is randomly assigned a different sign, "Positive" or "Negative", indicating the direction in which they are attempting to move the variable "x". Player "Negative" is dealt four cards and plays the odd numbered turns (first turn, third turn, fifth turn, etc.). Player "Positive" is dealt five cards and plays the even numbered turns (second turn, fourth turn, etc.). The initial setup is depicted in FIG. **2**.

[0054] Turn Sequence

[0055] On a player's turn he or she takes two actions, and then advances the next instruction pointer(s) (in numbered order if more than one pointer exists). The key concept of advancing a next instruction pointer is explained later. The player may choose from five types of actions:

[0056] 1. Play an Instruction Card

[0057] The player places an Instruction Card from his hand on the table at the bottom of the program (beneath the last card), adding to the existing program. If the card above is a Control Flow Instructions such as an "if" or "while" instruction, the new card must be placed at an indent. (A downward arrow might appear on Control Flow Instructions indicating where to align the left side of the next card.) Otherwise, the card may be placed at any level of indentation between that of the first instruction and that of the card above. Two special Instruction Cards must adhere to additional rules: an "else" card must match a preceding "if", and a "break" card can only be played within a while loop (explained later).

[0058] 2. Play a Special Action Card

[0059] Special Action Cards allow the player to do something other than adding an instruction to the end of the program. Examples of special actions include adding or removing Instruction Cards from the program, introducing a new next instruction pointer, and trading hands with one's opponent. The face of each such card contains text explaining the action.

[0060] The Special Action Cards labeled "Insert", "Move", and "Delete" can be applied to any Instruction Card in the program as long as (A) no next instruction pointer points to the card being moved or deleted, and (B) the rules governing the relative positions of instructions, described above, can be satisfied simply by adjusting indentations. (As an example of a modification after which the rules could not be satisfied simply by adjusting indentations: in the case of a matching "if" and "else" instruction pair, the deletion of the "if" instruction would leave an "else" instruction with no matching "if", and thus such a deletion would not be allowed.) Following the modification, the player slides cards up or down so that each card sits below its predecessor (next instruction pointers travel with the card to which they originally pointed). Then the player makes any indentation adjustments required to avoid violating indentation rules, working from the top of the program downward. If a next instruction pointer pointed to the space below the program, then an instruction moved into the empty space will be pointed to and will execute during the "Advance Next Instruction Pointers" phase.

[0061] The Special Action Cards labeled "Set Next" and "New Thread" can be used to position a next instruction pointer at any Instruction Card, regardless of what conditions appear above it. (But that next instruction pointer does not advance or execute during the "Advance Next Instruction Pointers" phase of the current turn.)

[0062] Powerful Special Action Cards such as "Set Next" require the player to spend both actions in order to play the card. (This requirement is indicated on the card itself.)

[0063] After a Special Action Card is played, it and any card removed from the program as a result of the action are deposited into a "discard pile".

[0064] 3. Draw a Card (if Permitted)

[0065] If the player's hand contains less than five cards, he or she may draw a new card. If the deck is empty, the discard pile is reshuffled to become the new deck.

[0066] 4. Discard a Card

[0067] The player may discard a card. (Drawing a replacement card is a separate action.)

[0068] 5. Advance any Next Instruction Pointer

[0069] A player can opt to use an action to advance any one (not all) next instruction pointers. This is in addition to the advancement of pointers that must occur at the end of the turn. When multiple next instruction pointers are in play, this action can be used to e.g. advance a lower priority next instruction pointer (labeled with a number indicating lower priority) before a higher priority next instruction pointer advances naturally.

[0070] Ending the Game

[0071] The game ends when the value of the variable "x" (**1**) reaches or exceeds the value positive five (in which case player "Positive" wins), or negative five (in which case player "Negative" wins). Entering this state immediately ends the game.

[0072] Executing the Program (Advancing the Next Instruction Pointers)

[0073] The most fundamental concept in the game is that of advancing a next instruction pointer, or in other words executing the current program. Unless the next instruction pointer points to the empty space below the program (**7**), this is done by (A) executing (performing the action described on) the Instruction Card pointed to by the next instruction pointer, and (B) moving the next instruction pointer to point to the next instruction to be executed. This is often the following instruction, but not always. A simple sequence of instructions is depicted in FIG. **3**.

[0074] In this example, advancing the next instruction pointer (**3**) consists of executing the instruction to which it points, and then sliding it down to the next card. Advancing it three times results in executing the instruction "i=1" (setting the value of "i" (**2**) to positive one), then executing the instruction "i=−i" (setting the value of "i" (**2**) to negative one), and lastly executing "x=x−i" (setting the value of "x" (**1**) to positive one). This leaves the next instruction pointer pointing to the empty space (**7**) just below the last card in the program. Once the next instruction pointer is pointing to this empty space, it ceases to advance until either a new instruction is played into the empty space, or a special action has the effect of moving the next instruction pointer to point to an instruction.

[0075] Conditionals

[0076] An "if" card represents a conditional: a special instruction that affects the flow of control. If the condition that appears in parentheses after the "if" is true, then when the next instruction pointer advances, its next stop is the instruction indented under the "if" card. This procession is illustrated in FIG. **4**A.

[0077] If the condition is false, however, then the set of cards indented below the "if" (known as the "if" block) is skipped. The next instruction pointer advances as if the indented instructions did not exist, as shown in FIG. **4**B.

[0078] The "else" card is a special card that can only follow an "if" at the same indentation. The next instruction pointer advances from the "else" card to the card indented under it only if the preceding "if" card's condition is false.

[0079] While Loops

[0080] A "while" card represents another type of instruction that affects the flow of control. As with an "if" card, the cards indented beneath the "while" card are only visited if the condition in parentheses is true. Unlike in an "if" block, however, at the end of a closed "while" block, the next instruction pointer returns to the "while" card at the top of the loop. This procession is illustrated in FIG. **5**.

[0081] If the last card in the indented block (**11**) is an Assignment (as must be the case if any cards exist later in the program at the same indentation as or to the left of the "while" card), then the "while" block is a "closed while loop," and from the last card in the block the next instruction pointer advances to the "while" card above. Otherwise, the loop is "open" and the next instruction pointer advances to the empty space below the last card.

[0082] The "break" card is a special card that can only occur in a while loop. From the "break" card, the next instruction pointer advances to the same card that would have been advanced to from the "while" card above had its condition been false.

[0083] Nested Control Flow

[0084] When an "if" or a "while" are played indented beneath another "if" or "while", they are said to be "nested". An example of nesting is shown in FIG. **6**. Nesting enables the construction of complex computer programs, and allows the number of levels of indentation to grow arbitrarily large.

[0085] Next Instruction Summary

[0086] In summary, when advancing a next instruction pointer: if the next instruction pointer points to the space below the program (**7**), nothing happens; otherwise the next instruction visited after executing any instruction other than "break" is determined by the indentation of the card below it.

[0087] Same indentation: Advance to the card below.

[0088] Greater indent (to the right): If the relevant condition is true (false in the case of an "else"), advance to the instruction indented below the current instruction. Otherwise, advance as if the instructions in the indented block did not exist.

[0089] Lesser indent (to the left): If the current instruction is contained in a while loop, advance to the (most inner) "while" card above. Otherwise, advance to the card below.

[0090] No cards below: If the current instruction is an Assignment and is contained within a while loop, then advance to the "while" card above. Otherwise, advance to the empty space below the last card in the program, even if this means proceeding past an "if" or "while" card whose condition is false.

[0091] Multiple Threads (Multiple Next Instruction Pointers)

[0092] In most modern computer systems, and in this game, it is possible for multiple next instruction pointers to simultaneously execute the same program, and to interact with each other. These are known as "threads", and Special Action Cards can introduce new threads. In this game, each next instruction pointer is labeled with a priority number indicating its priority relative to other next instruction pointers; higher priority next instruction pointers execute before lower priority next instruction pointers. If two next instruction pointers point to the same instruction, position the cards such that each arrow's priority number is visible. FIG. **7** depicts a scenario in which multiple threads advance. First the next instruction pointer labeled "NEXT (**1**)" (**3**) executes (the instruction labeled "i=max(i−1,−5)"), and then advances (up to the instruction labeled "while(i>−2)"). Then the next instruction pointer labeled "NEXT (**3**)" (**14**) executes (the instruction labeled "x=x−1"), and then advances (to the instruction labeled " i=max(i−1,−5)").

DESCRIPTION AND OPERATION OF ALTERNATE EMBODIMENT

[0093] The description below explains an alternate embodiment of the invention: an electronic game in which two competing players take turns modifying the same computer program until one of them achieves his or her objective. (As in the first embodiment, one player, "Positive", has the goal of setting the variable "x" to the value of positive five. The other player, "Negative", has the goal of setting the variable "x" to negative five.) Such an electronic game could be implemented as a mobile device game, a computer game, or a web-based game, but the description below and the drawing FIG. **8** focus in particular on a mobile device game.

[0094] Objects and Visual Representations

[0095] This embodiment of the game is played on a mobile device (**850**). As in the first embodiment, variable counters

(**801**, **802**) display each variable's current value. Instructions are arranged to represent a computer program (**851**), and a next instruction pointer (**803**) indicates the next instruction to be executed. As in the first embodiment, each player has at his or her disposal a set of playable objects (**852**) consisting of instructions and special actions.

[0096] Beginning the Game

[0097] As in the first embodiment, the arrangement of instructions known as "the program" (**851**) initially consists of only a single instruction bearing the text "i=1", pointed to by the next instruction pointer (**803**). As the game is played, players add additional instructions to the program, extending it. Variable counters (**801**, **802**) are initialized to zero and display their current value. Each player is randomly assigned a different sign, "Positive" or "Negative", indicating the direction in which they are attempting to move the variable "x". Each player is then allocated playable objects (**852**), instructions and special actions, randomly drawn from a predetermined set. Player "Negative" is allocated four objects and plays the odd numbered turns (first turn, third turn, fifth turn, etc.). Player "Positive" is allocated five objects and plays the even numbered turns (second turn, fourth turn, etc.).

[0098] Turn Sequence

[0099] On a player's turn he or she takes two actions, after which the game automatically advances the next instruction pointer(s) (in numbered order if more than one pointer exists). The rules governing how a next instruction pointer advances are the same as those explained in the description of the first embodiment. The player may choose from five types of actions:

[0100] 1. Play an Instruction

[0101] The player drags an instruction from his set of objects (**852**) into the empty space at the bottom of the program (**807**), adding to the existing program. If the instruction above is a Control Flow Instruction such as an "if" or "while" instruction, the new instruction must be placed at an indent. (A downward arrow might be used as a visual cue to indicate the indentation requirement.) Otherwise, the instruction may be placed at any level of indentation between that of the top instruction and that of the instruction above. Two special instructions must adhere to additional rules: an "else" instruction must match a preceding "if", and a "break" instruction can only be played within a while loop (explained in the description of the first embodiment).

[0102] 2. Play a Special Action

[0103] Special Actions allow the player to do something other than adding an instruction to the end of the program. Examples of special actions include adding or removing instructions from the program, introducing a new next instruction pointer, and trading sets of objects with one's opponent. Briefly tapping a special action will display additional text on the screen explaining the action. Pressing and holding one's finger on a special action will cause the action to be played, potentially prompting the user to do something to further specify the action. For example if the special action were a "Move" action, the user would then be required to drag an instruction from one location in the program to a different location in the program.

[0104] The Special Actions labeled "Insert", "Move", and "Delete" can be applied to any instruction in the program as long as (A) no next instruction pointer points to the instruction being moved or deleted, and (B) the indentation rules described above can be satisfied simply by adjusting indentations. Following the modification, the game automatically slides instructions up or down so that each instruction sits below its predecessor (next instruction pointers travel with the instruction to which they originally pointed). The game also makes any indentation adjustments required to avoid violating indentation rules, working from the top of the program downward. The game may dynamically rearrange instructions as the user drags an instruction to a new location, in order to illustrate to the user the effect of dropping the instruction at a particular location.

[0105] The Special Actions labeled "Set Next" and "New Thread" can be used to position a next instruction pointer at any instruction, regardless of what conditions appear above it. (But that next instruction pointer is not automatically advanced or executed at the end of the current turn.)

[0106] Note that powerful Special Actions require the player to spend both actions in order to play the action (indicated visually on the representation of the special action).

[0107] 3. Draw a New Playable Object (if Permitted)

[0108] If the player is in possession of less than five playable objects, he or she may draw a new object.

[0109] 4. Discard an Object

[0110] The player may discard an instruction or special action. (Drawing a replacement is a separate action.)

[0111] 5. Advance any Next Instruction Pointer

[0112] A player can opt to use an action to advance any one (not all) next instruction pointers. This is in addition to the advancement of pointers that automatically occurs at the end of the turn.

[0113] Ending the Game

[0114] The game ends when the value of the variable "x" (**801**) reaches or exceeds the value positive five (in which case player "Positive" wins), or negative five (in which case player "Negative" wins). Entering this state immediately ends the game.

[0115] Execution of the Program (Advancing the Next Instruction Pointers)

[0116] In this embodiment, next instruction pointers advance as they do in the first embodiment. Unless the next instruction pointer points to the empty space below the program (**807**), the instruction pointed to is executed, and then the pointer moves to point to the next instruction to be executed. As in the first embodiment, conditionals can cause the next instruction pointer to skip blocks of instructions, while loops can cause the instruction pointer to move up rather than down, and such constructs can be nested within each other. Also as with the first embodiment, special actions can introduce new next instruction pointers, each with a unique priority number. Unlike in the first embodiment, however, in this embodiment, next instruction pointers are advanced automatically by the game itself.

1. A game, with the following features:

moveable representations of computer instructions (such as assignment instructions, conditional instructions, and looping instructions), whether physical or virtual, that may be arranged to define a computer program, and

a system, whether manual or automated, for executing the computer program

2. The game of claim **1** wherein one or more modifiable counters are used to record the current values of variables used in the computer program.

3. The game of claim **2** wherein players have the option of appending instructions from an available set of instructions to an existing program.

4. The game of claim **3** wherein players have the option of taking actions from an available set of special actions such as inserting instructions into the middle of the program, deleting instructions, moving instructions, or directly modifying the values of variables

5. The game of claim **4** wherein the instructions, counters, and special actions are implemented as physical cards comprising a deck of cards.

6. The game of claim **4** implemented as an application running on a computer system wherein the instructions, counters, and special actions are depicted visually on the screen.

7. The game of claim **4** implemented as an application running on a mobile device wherein the instructions, counters, and special actions are depicted visually on the screen.

8. The game of claim **4** implemented as a web application, wherein the implementation of the game exists on a web server and players play the game on devices remotely connected to the web server.

9. The game of claim **4** wherein the actions of the players and the execution of the program occur in a consistent, ordered sequence, i.e. players take turns and program execution is triggered by player actions.

10. The game of claim **4** wherein the actions of the players and the execution of the program are not synchronized, i.e. execution of the program is not suspended while a player is deciding upon his or her next move.

11. The game of claim **4** wherein the instructions and special actions available to each player are determined by chance, for example by drawing such options from a deck of cards.

12. The game of claim **4** wherein the instructions or special actions available to each player are drawn from a predefined set.

13. The game of claim **4** wherein the program is executed as a multithreaded program, i.e. multiple pointers point to a next instruction.

14. The game of claim **4** wherein the set of available instructions includes the ability to perform a function call, and multiple counters are used to store values for a single named variable, allowing simulated program to effect a call stack.

* * * * *