BURG(er) - Build Utterly Rewarding Games Easily and Radiantly

Jacqueline Kong (jek2179)
Jordan Lee (jal2283)
Ashley Nguyen (akn2121)
Adrian Traviezo (aft2115)

Language Description:

BURG(er) is a programming language that enables users to create text-based adventure games in a streamlined way. BURG(er) is optimized for game writing with native data types to represent game players, inventory, scenes, and so forth. BURG(er) is an object-oriented language that has flexible subclass support to enable programmers to create custom game objects.

BURG(er) is ideal for writing command line text-based games with multiple-choice scenes. During each scene, the user might be presented with options that will decide which scene they'll be presented with next. BURG(er) also supports writing multiplayer turn-based games with a custom, easy-to-use server connection API.

Parts of Language:

Primitive Data Types

Type	**Description**
`int`	integer
`char`	a single character
`boolean`	boolean value that can be assigned value of true or false

Supported Data Types

Type	**Description**
`String`	An ordered and non-iterable list of chars.
`List`	Ordered, iterable

`Scene`	An object containing these values:
	`text: (String)`
	`options: (List)`
	`next: (Scene)`

`Inventory`	An object containing these values:
	`items: (list of Items)`
	`capacity: (int)`
	`amount: (int)`

	`display()`

`Item`	something that each Character can have in their Inventory; an object containing these values:
	`name: (String)`
	`quantity: (int)`
	`use()`

`Player`	An object containing these values:
	`name: (String)`
	`address: (String)// ip:port`
	`inventory: (Inventory object)`

`Option`	An object containing these values:
	`selector: (String)`
	`text: (String)`
	`action()`

Operators

Type	**Description**
`+`	addition of integers; also serves to concatenate strings.
`-`	Subtraction of integers
`*`	Multiplication of integers
`/`	Division of integers
`++, --`	Unary operators with the same functionality as in Java
`-=, +=, *=, /=`	Binary operators with the same functionality as in Java.
`[]`	Access specific list indices
`<<`	Less than operator
`>>`	Greater than operator
`<=`	Less than or equals to
`>=`	Greater than or equals to
`==`	Equals to
`has`	Similar to .contains() in jQuery, which serves to check the player's inventory. Functionality example: if (player1 has "weapon"){//fight}

`->`	Binary operator that takes an option as the left operand and a scene as right operand, used for quick mapping of scenes to options

Keywords

Type	**Description**
`if...else`	conditional statement
`loop { … }`	repeatedly iterates through statements within the brackets until told to exit based on a condition within the loop
`// text`	Single-line comments
`/* text */`	Multi-line comments from the opening slash to the last class.
`func`	Allows user to define a function. Can be used to declare an unnamed function on the spot.
`def x mods y`	Allows user to define a new object x, extending the y object
`<[selector], [text], [action]>`	The angled brackets denote an option. [selector], [text], and [action] are replaced with the option's respective selector, text, action function.

Functions

Type	**Description**
`print()`	prints to console
`exit()`	Exits the program
`input()`	Same as python input()
`options()`	Takes in a comma-separated list of options and displays the options for the player

Other features we'd like to include in our language:

- String formatting
- timer/clock API
- Sockets API-like library for server connections

Source code example:

```
Player Player1 = {
  address: local, // omit for default
```

```
  inventory: { // instantiate Item types and their quantities
    items: [Pepperoni Pizza slice(5), umbrella(), muscles(),
phone()],
    capacity: 10,
    shortcut: “inventory”
  }
};

/* within this text block is an example of using the -> operator to
quickly define game paths:
Scene Dominos, Rain, Enemy, Fight, EC, GameOver;
Dominos.text = "Pizza delivery! Bring this baby to EC!";
Rain.text = “It’s raining! What do you do?”;
GameOver.text = “You died, game over.”

Dominos(0, ”Get on bike!”)->Rain(0, ”Take out your
umbrella”)->GameOver();
*/

func START(){  // executes at runtime
  print("You're in dominos and you wanna deliver the pizza.
   it's raining, though. What do you do?");
  Player1.options( //instantiates
    <”umbrella”, "umbrella", GAME_OVER(){
        print("You died!");
        exit;
    }>,
    <”slow down”, "slow down", SEE_RIVAL()>,
    <”go faster”, “go faster", GAME_OVER()>
  );
}

func SEE_RIVAL(){
  print("You have been attacked by a Papa Johns delivery biker! What
do you do?");
  Player1.options(
    <”run”, "run", EC(){
      print("You are now in EC. What do?");
      Player1.options(
        <”eat”, "eat pizza", START()>,
        <”call”, "call customer", FINISH(){
          print("good job! you win!");
          exit; >
```

```
        })
      )
    }>
    <”fight”, "fight", FIGHT()>
  );
}

func FIGHT(){
  int p1_health = 10; //player 1 "health" is just an int
  int rival_health = 5;

  print("rival just threw a pizza at you! what do?");
  loop {
    Player1.options(
      <"throw", “throw pizza”, PIZZA_ATTACK(){
       rival_health -= 2;
       inventory.pizza--;
      }>,
      <"punch", “throw punch”, MUSCLES_ATTACK(){
       rival_health--;
      }>,
      <"cower”, “cower”, COWER(){
        p1_health--;
      }>
    );
    p1_health--;
    if (!p1_health){
      GAME_OVER();
    }
    else {
      EC();
    }
  }
}
```