BOGuS - The BOard Game Specification language.
A Lanaguage Proposal for COMS W4115
By: Cary Maister

Purpose
Bogus is intended as a relatively easy way to specify a board game, to be played interactively by a user at a computer terminal. Bogus will provide a framework for the programmer to specify a few elements that are common to many board games along with the rules that define how the gameplay proceeds.

Anatomy of a Bogus program
Any Bogus game will include the following elements:

- a board - consisting of a sequence of spaces
- a set of players
- a deck of cards from which plays draw to move around the board

Each space on the board, set of players and card in the deck will be associated with a rule. A rule is a procedure that has access to the current state of the game and can apply arbitrary logic to manipulate that state. When a card is drawn, that card's rule is invoked, which may move one or more players around the board, or modify players' attributes. When a player is moved to a space on the board, the space's rule is invoked.

A Bogus program is a single text file that initializes the board, the players and the deck, possibly using inputs from the players, then invokes a game engine which sequentially gives each player a turn drawing from the deck and using the rules to move the players around the board until the game terminates.

Flexibility
The primary goal of Bogus is to provide a simple language that can be used to specify a wide variety of games with different board layouts and different principles of how players move through the game and how the game ends. A basic example is a game where players draw cards from a deck, and each card tells the player to advance a certain number of spaces, and the first player to reach the final space wins. More complicated rules could be introduced to provide a game like snakes and ladders where players may jump around the board in a non-linear fashion, or a game like Sorry where players can land on each other's piece to force them back to the beginning of the game.

With some creative interpretation of the "board" metaphor, one could create a Bogus choose your own adventure story, where each space on the board represents a place in the story, and the player inputs their choices and move to different events and there may be more than one winning outcome. For truly enterprising, stay-in-your parents' basement eating cheetos types, there is the possibility of a Bogus text based role playing game, where players move around the game board collecting objects, battling dragons, trying to build experience and so forth.

Syntax
The Bogus syntax is a combination of natural English and the math-based syntax common to many programming languages.

Types:
Bogus has five primitive types:
board - contains the sequence of spaces through which players move
playerlist - contains a set of players involved in the game
deck - contains a set of cards which can be drawn by players to advance through the game

number - an integer
string - a sequence of printable ASCII characters

If a number if supplied in a context where a string is required, the number is automatically converted to a string.

Composite types:
A composite type can be declared as follows:
type Player has name(string), colour(string), health(number), location(number).

Memebers of a composite object can be accessed using a double-colon operator:
new Player p1.
p1::name is "Wonko".
p1::colour is "taupe".

Number and String Operators
For operands that are numbers, the language will use some arithmetic and comparison operators from C with their same meanings. Operators will include:
Artithmetic: +, -, *, /
Comparison: ==, !=, <, <=, >, >=

For string operands, the following operators will be available:
Concatenation: +
Comparison: matches, notmatches

The string comparison operators will evaluate whether two strings are identical:
e.g. "marvin" matches "marvin" will be true, but "marv" matches "marvin" will be false.

Two control flow statements operations will be borrowed from other programming lanaguages, with common meanings:
if(logical value) { block of code } else { other block of code }
while(logical value) { block of code }.

Input/Output
There are two input operators and one output operator:
writeLine <string> - writes the specified string, followed by a newline to standard output.
readNumber - reads a valid integer followed by a newline from standard input
readString - reads a string followed by a newline from standard input

Comments
Comments consist of a # character and extend to the end of the line

Sample code
Here are snippets from a sample game of snakes and ladders. This code adds the head of a snake to the board, along with a rule to move the player to the snakes tail if he lands on the head. Note that the space where the tail is located has already been stored in the number tail1.

Note that the following special variables are available inside any rule:
currentPlayer - the player who's turn is currently taking place
currentSpace - the space where the currentPlayer is located.
currentCard - the card the player has drawn (in this example, each side of the die is a "card").

declare types
```
type Player has name(string), location(number).
type Space has target(number).
type Side has value(number).

## initialize game components
new board theBoard(Space). # declaring a board that is a sequence of
                           # spaces
new playerlist thePlayers(Player).
new deck theDie(Side).

...
# set up the die
new rule dieRule
{
     new number dest.
     dest is currentPlayer::location + currentCard::value.

     writeLine "Moving " + currentPlayer::name + " to " + dest.
}.

new Side side.
side::value is 1.
append(theDie, side, dieRule).
side::value is 2.
append(theDie, side, dieRule).
# etc.


...
```

```
new rule snakeRule
{
     new number space.
     space is  currentSpace::target.

     writeLine currentPlayer::name
     + " landed on a snake. Sliding back to space " + space.

     movePlayer(theBoard, currentPlayer, space).
}.

new Space s.
s::target is tail1.

addSpace(theBoard, s, snakeRule).
...
startGame(theBoard, thePlayers, theDie). #startGame is builtin
```