SLAG
Structured Language Applied to Gaming

Language Proposal

COMS W4115: Programming Languages and Translators
Professor Stephen A. Edwards
Computer Science Department
Spring 2008 Columbia University

Edward Arnold-Berkovits
ea2124@columbia.edu

Objective
To create a language to facilitate the construction of programs that simulate battle gaming systems.

Background
Many gaming systems use common concepts of at least 2 entities engaged in a battle of some sort with the goal of each of the entities to remove the other from the battle. These gaming systems can be tabletop miniatures games, simulated warfare games, or of course computer games.

Features
The common features of some of the gaming systems are as follows:
- Entity 1 attacking entity 2
- Each entity possessing meters (or values) regarding the state of their health, movement ability, attack ability, defense ability and damage ability
- There are modifiers to each of those values that influence attacks to either increase/decrease the likelihood of an attack succeeding, make an attack do more/less damage, or cause effects after the attack has occurred
- When an entity's health is non-positive (zero or below), that entity is no longer in the battle
- A system to decide if attacks hit. Possible choices are:
    1. Straight-up attack value compared to defense value.
    2. Random element, such as dice for tabletop games, to "make up" the difference between attack value and defense value.
    3. Comprehensive elements, which takes into account an entity's features, such as experience or health, the state of the battlefield, or the state of each entity's team.
    4. Some combination of the above, either with or without interdependencies.

Language Overview
SLAG will incorporate a standard Random element, which will be dice rolls, in the simulation of a tabletop battle game system to resolve attack successes or failures.

Characters will always have a name, and measures for amount of health.
Each phase of a character’s health will have measures for movement (mv), attack (av), defense (dv), damage (dmg). These will be referred to as the Core values. Abilities may be assigned to any phase of their health and can affect any of the Core values. Abilties’ effects on health or attacks would be able to become as complicated as the developer wishes.

Programmers can Create the following: Entities (for now, just Characters), Abilities (for Entities to use), Battles, and Attacks within those Battles. Each of these structures will begin with a letter, followed by letters and/or numbers.

Semicolons and braces divide up statements.
Whitespace is not important.
Variable initialization is done with an “=”.
SLAG is case-insensitive.

Code Samples

```
// Define gaming parameters. This would be useful for purposes of GUIs or comparisons
// of relative strength of characters.
Set game.maxhealth = 11;

// Define characters with Name and amount of Health
// Create <structure> <structure name> = <status> <structure>(“<entity full name>, <health>);
Create Character Spiderman = new Character("Spider-man", 8);

// Define Core values. A list of integers. <entity name>.<ability> = (<integer value array>);
Spiderman.mv = (8,8,8,7,7,7,6,6);
Spiderman.av = (10,10,9,9,8,7,7,6);
Spiderman.dv = (17,17,16,16,15,15,15,14);
Spiderman.dmg = (3,3,2,2,2,2,2,1);

// Define abilities. (Will including Abilities will be Phase II of language development?)
// Set <entity>.<ability> <ability name> (“<ability full name>”, <ability affected>  +/-<integer>)
Set Spiderman.attack webslingers ("Webslingers", av +1);
Set Spiderman.attack incap ("Incapacitate", dmg +1);
Set Spiderman.defense spisense ("Spider Sense", dv +2);

// Define locations of abilities relative to health position
/* This feature has yet to be developed. Choices are inclusion into Core value declaration,
separate declaration, or both. */

// Code snippet for status. Display <entity name>.<ability>;
Display Spiderman.health;
Display All.health;

// This game has battles divided up into attacks. The attacks are divided up into attack
// phase and resolution phase
// Language should allow user to make up whatever game they want & use this language.
// Code snippet to set up a battle. Create <structure> <structure name> (<entity1>, <entity2>);
Create Battle myBattle (Spiderman, Sandman);
// Code snippet to set up an attack. Create <structure> <structure name> (<entity1>, <entity2>);
Create myBattle.attack spiSand (Spiderman, Sandman);      // attacker, defender
Display myBattle.health;       // Show status of all entities in Battle

// Code snippet for attack phase of an attack. Compare <integer1> to <integer2>
if (Spiderman uses webslingers to attack Sandman) {          // not Boolean, how parse?
      Compare (Spiderman.av +1) to Sandman.dv;
} elseif (Spiderman uses incap to attack Sandman) {
      Compare Spiderman.av to Sandman.dv;
} else {
      Compare Spiderman.av to Sandman.dv;
}
```

```
// Code snippet for resolution phase of an attack
if (spiSand.succeeds) {
        Spiderman.health = Spiderman.health – Sandman.dmg;
}

// Code snippet to show variable assignment and do-while loop
maxDefense = 14;
while (Sandman.dv > maxDefense) do {                // simulate repeated sequential attacks
        Compare (Spiderman.av +1) to Sandman.dv;
        Spiderman.dmg = Spiderman.dmg – 1;
}
```

Reserved keywords

if	elseif	else			
while	do				
uses	to	succeeds	All	TRUE	FALSE
movement		attack	defense	damage	health
mv	av	dv	dmg		
Character		Battle	Attack		
Set	Create	Display	Compare		

The dot operator, ".", is reserved for use with <entity>.<value> or <structure>.<substructure>
Braces are reserved to delineate structures.

Operators

Arithmetic operators:

- + (addition) (+ will also concatenate any 2 types if either is non-integer)
- - (subtraction)
- * (multiplication)
- / (division)
- = (assignment)
- == (equal)
- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or equal to)
- != (not equal to)
- // comment
- /* multi-line comment */

Looping constructs:

- if (<*stmt*>) { <*stmts*> } else { <*stmts*> }
- while (<*stmt*>) do { <*stmts*> }
- foreach (<*list*>) { <*stmts*> } // future expansion

Boolean:

- TRUE FALSE

Note that "and", && and "or", || are not included in the language definition at this time.

Testing

Testing will be done using single character battles and multiple character battles. Use of Display statements will show the progress of the battles.

Further expansion concepts

The following concepts may be incorporated into the language as time/ability allows:

- Incorporation of different systems to determine Attack success/failure.
- Incorporation of other Core values, such as Experience or Fatigue.
- Teams as a new form of an Entity. As any Entity has ability to attack any other Entity, this would dictate the usage of Array of Entities and necessitate the inclusion of foreach.
- Abilities that incorporate effects either before an attack (so that the attack cannot be launched or it is modified somehow) or after the attack (what was a successful attack ends up missing or other post-attack effects).
- Turns and attack limits or conditions.
- Information about each entity's locations and proximity to each other which could influence each of the health, attack, defense and damage factors.
- Generalize language and grant ability to create/define various Core values.
- “and”, && and “or”, || use in if-then and do-while constructs.