# The *Next* Programming Language: Proposal

Ernesto Arreguin (eja2124), Danny Park (dsp2120),
Morgan Ulinski (mu2189), and Xiaowei Zhang (xz2242)

October 4, 2010

## 1 Language Description

Our language provides a way to easily create a text-based computer game. Users of the language can specify characters, locations and items that will appear in the game, and they can design the general plot of the game that ties these together. This language will be ideal for creating text-based RPGs (or digital "choose your own adventure" stories), but it can also be used to design other types of games.

## 2 Language Elements

### 2.1 Declarations

The program is primarily structured around declarations. The user will create declarations for characters, locations, items, missions, and attributes.

#### 2.1.1 Characters

Characters can appear in locations throughout the game. A declaration for a character will provide the character's name and valid interactions for this character, such as *attack* or *converse*.

#### 2.1.2 Locations

Locations provide a basic map of the game's landscape. These are the "rooms" in which a game player can find characters to interact with, items to pick up, or options for actions. Locations can be nested; for instance, one might want to declare a location “castle” with nested sub-locations “dungeon” or “tower.” The game player moves through locations during the game, and is presented with a list of options at each of these locations. This serves also to define the basic plot of the game. Within each location declaration, the possible actions are listed, as well as the result of taking these actions. Results for action can include moving into a different location, changing attribute values for characters, or displaying new text on the screen. It is possible to attach probabilities to each possible result of an action, to incorporate some randomness into the game.

One location needs to be specified as the “starting” location for the game; this defines where all game play will begin.

### 2.1.3 Actions

Actions can serve to move the character from location to location, or they can provide the ways to interact with other characters and items in the game. For instance, there could be actions to move, attack, defend, pick up objects, or talk with other characters. Creating actions for different directions in the game (*up*, *down*, *left*, and *right*) will allow programmers to create games beyond a basic RPG and into the realm of fighting games or driving games.

### 2.1.4 Items

Items are objects that the game player can pick up and maintain in their inventory. Collecting items can be requirements for missions, or possessing an item might improve a certain attribute such as strength or speed.

### 2.1.5 Missions

A mission is a collection of requirements. When all requirements in the mission declaration are completed, the game player earns a completion of the mission. The requirements can include collecting items, visiting locations, performing certain actions, or meeting certain characters. Missions can then be used as a way to define when the game level should increase.

#### 2.1.6 Attributes

Attributes are characteristics such as strength or speed, which each character in the game possesses, including the main player. Attributes can vary throughout the game, and may be used in conditional statements in a "location" declaration to define what the result of an action should be. For instance, if an attribute "strength" is high, a user may want to define a higher probability of successfully defeating an enemy.

### 2.2 Levels

Our language will also provide a way to handle levels of a game, if desired. The user will be able to decide whether they want to use levels in their game, and what criteria to use for "leveling up." Examples for level-up criteria could be thresholds for number of missions completed, number of locations visited, number of characters interacted with, etc. Attributes such as strength or speed can be enhanced when a player passes a level, to provide some incentive.

### 2.3 Basic Math

Our language will need to provide the tools for performing basic math, such as addition, subtraction, and boolean operations. These will be needed to handle increasing the level, updating the numerical features of attributes, or examining features of attributes for conditional results for actions.

## 3 Code Sample

The code sample attached is an example of some *Next* code that could create a very simple fighting game.

```
Game = badfighter {"Welcome to bad Fighter. Get Ready!!"}   //name of game and intro statement
location jungle {position: x=50, y =2};                              //location declaration with position
                                                            //attribute (two dimensional from 0 to 50 x and   o to 2 y)


time timer1 = 100;                                          //time declaration and set time
character neil {"Neil", life:100,                           //character declaration with attributes
                attack   punch{a, damage:5, reach:5},       //first letter of attributes is a keyboard command
                attack   kick{s, damage:10, reach:10},
                defense   block{w, punch:0, kick:kick / 2}};
character slave{"Mr. Slave", life:100,                      //another character declaration with attributes


//attack declarations with attributes
                attack   punch{ai, damage:10, reach:3},       //ai is part of the language(system controlled)
                attack   kick{ai, damage:20, reach:8},

//defense declarations explain how they modify attack damage
                defense   block{ai, punch:0, kick:kick / 2},

//declaration of moves, automatically applied to position
                move   left{ai, x =x-1},
                move   right{ai, x=x+1},
                move   jump{ai, y = 2},
                move    duck{ai, y = 0},
                move   stand{ai, y=1},
                move   stay{ai, nothing}};                  //nothing… part of the language

output{"fight"};                                            //output message

//Start jungle scene(or mission), starts timer…
//Also realized that position should be declared here for inbounds checking
//and because in general a player's position is a property of the scene
start jungle and timer with neil{position: 0 1}, slave{position:50 1} end neil.life=0 or slave.life=0 or timer = 0
(

//automatically translate input into specific move and depending on move
//code is executed
        when neil.punch (
if slave.position.x - neil.position.x <= neil.punch.reach and              //if punch connected then
                slave.position.y = neil.position.y

//probability blocks start and end with ?, should check if probs add to 100
//still deciding if the damage statement be written as follows or simply neil.punch..
//either way, the amount is here subtracted from life automatically
                                then (? prob 60 ( neil.punch.damage -> slave.life;
                                                slave.talk {"Oh Jeshush Christh" };
                                                slave.position.x=slave.position.x + 2;)

//here damage is applied to block… the amount is transformed
// to what was specified in defense declaration (here 0 is subtracted from life)
                                prob 40 (neil.punch.damage -> slave.block;
                                                slave.talk {"You can't touch this!"}

//'next' statements exit current code and go to the beginning of the start loop
                                                slave.position.x = slave.position.x + 1;)? next;)
```

```
                    else (? prob 30 (slave.move.left;)
                        prob 45(slave.move.left;
                            slave.move.stand;
                            slave.kick;
                            if slave.position.x - neil.position.x <= slave.kick.reach
                                then (slave.kick.damage ->neil.life;
                                neil.talk{"Ouch"};))
                        prob 10(slave.move.stay;)
                        prob 5(slave.move.jump;)
                        prob 5(slave.move.duck;)
                        prob5(slave.move.stand; slave.move.left;)? next;))

            when neil.kick (if slave.position.x - neil.position.x <= neil.kick.reach and
            slave.position.y = neil.position.y
                        then (? prob 60 ( neil.kick.damage -> slave.life;
                                slave.talk {"Oh Jeshush Christh" };
                                slave.position.x=slave.position.x + 3;)
                        prob 40 (neil.kick.damage -> slave.block;
                                slave.talk {"You can't touch this!"}
                                slave.position.x = slave.position.x + 1;)? next;)
                    else (? prob 30 (slave.left;)
                        prob 45(slave.left;
                            slave.stand;
                            slave.punch;
                            if slave.position.x - neil.position.x <= slave.punch.reach
                                then (slave.punch.damage ->neil.life;
                            neil.talk{"Ouch"};))
                        prob 10(slave.stay;)
                        prob 5(slave.jump;)
                        prob 5(slave.duck;)
                        prob5(slave.stand; slave.left;)? next;))
            when neil.block (? prob 15 (slave.move.left;)
                    prob 15 (slave.move.right;)
                    prob 20(slave.left;
                        slave.stand;
                        slave.punch;
                        if slave.position.x - neil.position.x <= slave.punch.reach
                                then (slave.punch.damage ->neil.block;
                                neil.talk{"Blocked!!"};))
                    prob 25(slave.left;
                        slave.stand;
                        slave.punch;
                        if slave.position.x - neil.position.x <= slave.kick.reach
                                then (slave.kick.damage ->neil.block;
                                neil.talk{"Blocked!!"};))
                        prob 10(slave.stay;)
                        prob 5(slave.jump;)
                        prob 5(slave.duck;)
                        prob5(slave.stand; slave.left;)? next;)
            when input_error (narrator.talk{"Your input is wrong!!"};          //default of when statements
                    next;))          //end of loop
        output {"Game over!!"};
        end badfighter;          //end of program
```