G!
A programming language for 2D games
Project Report

Amortya Ray ar2566@columbia.edu
Divya Arora da2254@columbia.edu
Rachit Parikh rnp2102@columbia.edu
Steve Lianoglou sl2585@columbia.edu

COMS W4115: Programming Languages and Translators
Department of Computer Science
Columbia University
December 19, 2006

Contents

Chapter 1
Introduction

1. Introduction

1.1 Motivation

The minutia of game development is a tedious and complicated affair. In general, game developers are forced to write repetitive code that requires a lot of bookkeeping in order to ensure its proper function. If one takes a moment to think about the details involved in writing code to figure out if two objects run into each other, or moving an object on the screen in response to a keyboard press, the details of this process will quickly come to the light.

Enter **G!** The goal of G! is a game developing language which specializes in making interactive 2D games. G! will enable the game developer to focus on the overall game play of the target game instead of the rote details of the game play implementation.

1.2 Background

G! is a hybrid, simple, high-level, versatile, architecture neutral, portable, Newton-aware, rapid development gaming language.

As the language designers who precede us have done, we have taken the liberty to characterize our language with a set of buzzwords. Each buzzword will be explained below along with the problems we are attempting to solve which resulted in their use.

We have found a Java gaming library (http://goldenstudios.or.id/products/GTGE/GTGE) which we will utilize heavily. Our language will offer an elegant syntax for game development while leaving the heavy lifting of collisions and sprite movement to this library.

1.3 Language Goals

Hybrid
Object Oriented Languages have been rising in language over the past several years. Some argue that this is a result of the expressive power that it lends to the programmer. Others would also argue that the previous argument is a fallacy and its true reason for popularity is to mitigate the impact of poorly written code to affect a larger system. We won't take a stand on this issue, per se, but, in general, we think OO-programming is A Good Thing™.

Sometimes OO programming can be too cumbersome and more traditional procedural programming techniques are Good Enough™ to solve a given problem.
G! will let you mix the two techniques in order to leave it to the developer to choose which approach is best suited for a given task.

Simple
What do we mean by simple?

The unique selling point of G! is that ANYBODY with minimal technical skills will smoothly be able to develop their own, unique gaming application. There will be a small number of keywords and constructs to remember with easy and intuitive syntax which will expose extensive libraries of functionality in an intuitive manner. This will simplify the process of writing game code.

High Level
The syntax and semantics of G! will allow the programmer to think of the game more on the lines of objects on a 2D plane as opposed to pixels in a Vector Space. The mechanics involved with writing code to move an image across the screen will be hidden underneath higher-level functions that will take care of these details for the developer.

Versatile
G! is a language which can be used to quickly and easily develop any type of game that can be played in 2D. Its functionality is not limited to writing only board games, card games, or the like.
If the game can be thought of as being played on a 2 dimensional plane, then G! is the right tool for the job.

Architecture Neutral
The game developer can program the game on the computer of her choosing. G! can be developed using Windows, Mac OS X, or Linux running on x86 or PowerPC architecture.

Portable
G! programs are compiled into java byte code, thus enabling the resulting programs to run on any machine that can run the Java Virtual Machine (Version 1.5 required)

Newton-aware
Part of developing games involves working with different bodies and how they react to each other.
While Newton was focused on bodies of the celestial variety, G! deals with bodies that move around in the game. Game objects are collision-aware. No need for the developer to figure out how to determine if one thing touches the other, our game objects allow the developer to code up reactions when they are “touched”.

Rapid Development
Hybrid + High Level = Rapid Development!
In a few short lines of code you can have a rudimentary game going and a few short minutes is more than enough to come up with these few short lines of code.

Avenue for Productive Procrastination
G! makes you feel good about yourself. Are you slamming your head against the wall trying to build a linear classifier for data in n-dimensional hyperspace? Take a break and code up a quick game. You'll have all of your friends fooled as they think you're busily solving the problem at hand, when you're really having fun and making yourself a new game!

Synergistic

According to answers.com, 'Synergy' is defined as the interaction of two or more agents or forces so that their combined effect is greater than the sum of their individual effects."
We believe that co-operative interaction among the G! language code, the libraries and the compiler along with the JVM, creates an enhanced combined effect makes G! truly synergistic.

Chapter 2
Tutorial

A brief tutorial providing some simple examples:

Example 1: Create a simple object that shoots bullets

```
//This example shows how intuitive it is to code in G!

PlayField pf;       //needed to hold objects
Sprite plane, bullet;

pf->properties(background=blue);   //set the background
color
plane->properties(image="plane.png", x=300, y=250);
bullet->properties(image="bullet.png" x=(-50), y=(-50));

when(KeyPress('A')) {
     shoot();
}

when(KeyPress(left)) {
     plane->x = plane->x - 0.1
}

when(KeyPress(right)) {
     plane->x = plane->x + 0.1
}

func void shoot() {
     bullet->x=plane@center;
     bullet->y=plane@top;
     while(bullet->y < 500) {
          bullet->y = bullet->y + 0.2;
     }
}
```

Example 2: Create 2 objects. When they collide, play a sound and one object disappears depending on which side they bang from

```
PlayField pf;
Sprite object1, object2;
Sound s;
pf->properties(background=white, height=200, width=300.5);
object1->properties(image="obj1.png" x=234, y=234);
object2->properties(image="obj2.png" x=150, y=150);
s->properties(file="sound.wav");

when(KeyPress('A')) { object1->x = object1->x - 0.2; }
when(KeyPress('D')) { object1->x = object1->x + 0.2; }
when(KeyPress('W')) { object1->y = object1->y - 0.2; }
when(KeyPress('X')) { object1->y = object1->y + 0.2; }

when(KeyPress(left)) { object2->x = object2->x - 0.2; }
when(KeyPress(right)) { object2->x = object2->x + 0.2; }
when(KeyPress(up)) { object2->y = object2->y - 0.2; }
when(KeyPress(down)) { object2->y = object2->y + 0.2; }

when(object1 !left object2) {
     s->play();
     object1->visible=false;
}

when(object1 !right object2) {
     s->play();
     object2->visible=false;
}
```

Chapter 3
Language Reference Manual

3.1 Lexical Conventions

3.1.1 Tokens

The lexical conventions used in G! can be roughly divided into 6 categories- tokens, comments, white space, identifiers, keywords, operators, line separators, line terminators and constants.

3.1.2 Comments

G! allows for two types of comments. Inspired by C/ Java, we allow the programmer to include single line comments using the double forward slash '//'. The comments that begin with the '//' token run till the end of the line. G! also follows the C/ Java convention for multi-line tokens. Multi-line comments begin with a forward slash '/', followed by an asterisk '*'. They run till the next asterisk '*' followed by the forward slash '/' is encountered.

Example:

```
// This is an example of a single-line comment in G!

/* This is
an example
of a multi-line
comment in
G! */
```

Nesting comments doesn't really make much of a difference. In other words, the multi-line comment begins at the first slash-star '/*' and ends at the following star-slash '*/' irrespective of whether it encounters a subsequent slash-star '/*' in between. Hence the following nested comment is redundant.

Example:

```
/* This is an
example of
//This is a nested comment
nesting comments */
```

3.1.3 White Space

White space is ignored by the G! compiler. White space includes spaces, new line characters '\n', tabs '\t'. They are only used to separate tokens.

3.1.4 Identifiers

An identifier is a user defined variable that is used by the programmer in the source. As per G! conventions, an identifier has to begin with a alphabet (A-Z, a-z), and can have any character following it (A-z, a-z, 0-9, _). Hence, identifiers cannot start with a number or an underscore. Identifiers are case sensitive. In other words, 'var' is different from 'Var' and 'VAR". Lastly, an identifier cannot be a reserved keyword. Also, identifiers are defined by scope. Only one identifier with a certain name can be defined in any particular scope. However, blocks of statements are allowed to overwrite identifiers found in their parent scope.

Example:
Integer *count*;
Double *var*;
Sprite *car*;
SpriteGroup *deck*;
Sound *soundtrack*;
(The data types Integer, Double, Sprite, SpriteGroup and sound are explained later in the manual.)

3.1.5 Keywords

Keywords are special reserved identifiers. An identifier cannot have the same name as a keyword.
Following are the keywords in G!.
for
if
else
while
include
func
break
continue
return
void
when
print
null
G! tries to provide the programmer with a minimum number of keywords, while at the same time trying to maximize the utility provided by the keywords.

3.2 Operators

G! includes the following 8 categories of operators:

1. ! operator
2. Arithmetic
3. Relational
4. Logical
5. PROP operator (->)
6. Colon Operator
7. Function call, Assignment, comma and Array Reference operator
8. @ operator

Here's a brief on the different operators and their Associativity and Precedence:

3.2.1 The ! (Bang) Operator

The ! operator has its origins in the G! evolution and is essentially a binary boolean operator that operates on two Sprite objects, Sprite being a datatype in G!. Since G! is a language specialized in game development and games are incomplete without collisions and explosions, we provide the user with a convenient way to denote such collisions using the ! operator. The ! operator can take 5 different forms as follows:

Operator	Syntax	Description
!	SpriteA *!* SpriteB	SpriteA collides with SpriteB
!left	SpriteA *!left* SpriteB	SpriteA collides with SpriteB on the left
!right	SpriteA *!right* SpriteB	SpriteA collides with SpriteB from the right
!top	SpriteA *!top* SpriteB	SpriteA collides with SpriteB from the top
!bottom	SpriteA *!bottom* SpriteB	SpriteA collides with SpriteB from the bottom

The above ! expressions are generally used in the conditional part of the "WHEN" statements which are asynchronous statements that are executed whenever a particular condition is true, irrespective of where these when statements are positioned in the program. They are left-to-right associative. This is explained in more detail in section 6.

An example of the use of ! operator would be:

```
when (hero !left wall) {
// when the hero bangs into the left side of the wall ...
hero.setHorizontalSpeed(0);
}
```

3.2.2 The @ Operator

The @ Operator was developed to be used in conjunction with the "Coordinate" datatype in G! which represents the x and y coordinates of a sprite on the 2-D playfield. However, due to lack of time, we were unable to implement the functionality of this very useful operator. Nonetheless, we would like to present the general idea behind the operator.
The @ Operator is a postfix, right-to-left associative, unary operator used in the form Spritename @<location> such that it returns the coordinates of the specified location of the Sprite operand on which it operates, given that the sprite is represented by a 2-D image.

For example:

```
when (bullet ! enemy) {
// calls method to play an explosion Sprite at the
// center of the enemy
play_sprite_once(sprite: explosion, pos: enemy@center);
}
```

The location could be left, right, center, top, bottom, topleft, topright, bottomleft, bottomright as follows:

Operator	**Meaning**
@top	Coordinates of top middle of sprite on left of @
@right	Coordinates of right middle of sprite on left of @
@bottom	Coordinates of bottom middle of sprite on left of @
@left	Coordinates of left middle of sprite on left of @
@center	Coordinates of center of sprite on left of @
@topleft	Coordinates of top left corner of sprite on left of @
@topright	Coordinates of top right corner of sprite on left of @
@bottomleft	Coordinates of bottom left corner of sprite on left of @
@bottomright	Coordinates of bottom right of sprite on left of @

3.2.3 Arithmetic Operators

Multiplicative Operators

Multiplicative operators are binary operators which operate on two operands and perform their respective operations on the values of both the operands. The following is a table ofmultiplicative operators in G!, their syntax and their description:

Operator	Syntax	Description
*	A * b	a times b
/	a / b	a divided by b
%	a % b	Remainder of a/b
^	a ^ b	a to the power of b

The multiplicative operators associate from left to right.

Additive Operators

The additive operators + and – are binary and associate from left to right. The result of the + operator is the sum of the operands. The result of the - operator is the difference of the operands.

The operands must be both arithmetic type.

Operator	Syntax	Description
+	A + b	a plus b
-	a - b	a minus b

3.2.4 Relational Operators

The relational operators in G! are mainly comparison operators that associate from left to right. They include the operators <, >, <=, >= and == which represent less than, greater than, less than or equal to, greater than or equal to and equal to, respectively. They all yield a result of type *integer* with value 0 if the specific relation is false and 1 if it is true.

Operator	Syntax	Description
<	a < b	1 if a < b; 0 otherwise
>	a > b	1 if a > b; 0 otherwise
<=	a <= b	1 if a <= b; 0 otherwise
>=	a >= b	1 if a >= b; 0 otherwise
==	a == b	1 if a equal to b; 0 otherwise

3.2.5 Logical Operators

Logical operators are binary operators, which perform logical operations on the values of the two operands and return a Boolean value (1 for true or 0 for false). G! supports the following left-to-right associative logical operators:

Operator	Syntax	Description
AND	a AND b	Logical AND of a and b (yields 0 or 1)
OR	a OR b	Logical OR of a and b (yields 0 or 1)
NOT	NOT a	Logical NOT of a (yields 0 or 1)

The motivation behind using the words AND, OR and NOT instead of the more traditional &&, || and ! is that the words are more intuitive and do not require the programmer to remember which out of the variety of symbols apply here.

3.2.6 The PROP Operator (->)

G! utilizes the PROP operator to access variables, or properties. The PROP operator is used to test or set the properties of an object or to execute a method of an object. The PROP operator is left-to-right associative. Generic examples of the PROP operator are:

Object->property_or_method
Instancename->variable
object: An element of the game. This parameter is always to the left of the PROP (.) operator.
property_or_method: The name of a property or method associated with an object. This parameter is always to the right of the PROP (.) operator.

3.2.7 The Colon Operator (:)

The Colon operator in G! is mainly used in the “for” loop to initialize the iterator, provide the terminating condition and to indicate an increment or decrement of value every iteration as in the following example:
for i = 1:10:1
for j = 10:1:-0.5

Here the loop variable i is initialized to 1.The terminating condition is separated by the first colon and the increment or decrement is separated by the second colon. The second colon and the increment/decrement specification is optional, in that if the user doesn't specify the last part of the expression, then it is set to increment by 1,by default. Eg:
for i = 1:10

3.2.8 Function call operator, Assignment operators, the comma operator and the Array Reference operator

Operator	Example	Description/Meaning
( ) = *= /= += -= , []	F() a = b a*=b a/=b a+=b a-=b e1,e2 a[10]	Function call a, after b is assigned to it a equals a times b a equals a divided by b a equals sum of a and b a equals difference of a and b e2 is evaluated after e1 Array reference

The precedence of the above operators is as follows:

Operator Symbol	Operator Function
->	Member selection (object)
[ ]	Array subscript
( )	Function call member initialization
!	Bang operator
@	Coordinate operator
NOT	Logical not
–	Unary minus
+	Unary plus
^	To the power of
*****	Multiplication
/	Division
%	Modulus
+	Addition
–	Subtraction
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equality
AND	Logical AND
OR	Logical OR
:	Colon
=	Assigment
,	Comma

3.3 Functions
3.3.1 Function Definitions

A function definition is the code which explains the execution of that function. Function definitions can occur in any order and in different files, although they cannot be nested and one definition cannot be split across multiple files.
A function definition should have the following format:
func return-type function_name(<data_type param1>, <data_type param2>, ...) {
	function_body
	return statement (not required where return-type is void)
}

A function definition should always start with a *func* keyword. This is followed by a return-type which could either be *void* or one of the data types defined in section 5. A *function_name* has to be unique (no keywords) and no two functions that are used in one file can share the same name. Additionally, zero or more parameters can be passed as arguments to a function. The scope of these arguments, just like the scope of all variables defined within a function, is limited to the function definition. The body of the function is placed with {} and if the return-type is not void, then a value must be returned at the end of the function definition. Note that the value returned by a function is not an lvalue. A function call, therefore, cannot constitute the left side of an assignment operator.

Example:
```
func int findMax(int i, int j) {
	if(i>j)
		return i;
	else
		return j;
}
```

G! also allows the programmer to define functions with optional arguments. This is accomplished by introducing the idea of a *keyword argument*. To define a keyword argument, the argument in the function's definition must be assigned a default value (in the function's parameter list). Every argument *to the right* of the first argument in the function definition that is assigned a default value *must also* be assigned a default value. When the first optional parameter is introduced, the remaining parameters must also be optional.
The caller of a function that is defined with keyword arguments can reference those arguments by name. This also allows for the order of the keyword arguments that the caller uses to be arbitrary. Below is such an example to show how this works by presenting a function which calculates the distance between two objects (or one object and the origin), and optionally plays an alert before it returns.

```
func int calcDistance(Sprite objectA, Sprite objectB = null,
Sound alert = null) {
Coordinate a,b;
a = objectA@center;
if (objectB == null) {
// assume we want to calculate distance from origin
b.x = 0;
b.y = 0;
} else {
b = object@center;
}
```

```
if NOT (alert == null) {
alert.play();
}
return sqrt((a.x - b.x)^2 + (a.y - b.y)^2);
}
// This function can now be called like so:
distance = calcDistance(mySprite); // gets distance to
origin
distance = calcDistance(mySprite, alert: ding); // distance
with alert
distance = calcDistance(mySprite, otherSprite); // dist
betw. 2 objects
distance = calcDistance(mySprite, objectB: otherSprite); //
same thing
// now calculates distance between two objects and
// plays a sound to let us know ... something
// note how the order of alert and objectB have changed from
// the original function definitio
distance = calcDistance(mySprite, alert: ding, objectB:
otherSprite);
```

3.3.2 Function Calls

A function call is a primary expression, usually a function identifier followed by parentheses, which is used to invoke a function. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function. For example, the call the function findMax defined in the example in section 4.1, we can do the following:
int max = findMax(x, y);

3.3.3 Built-in Functions

G! provides a lot of flexibility through its wide range of operators to execute routine gaming tasks. Hence it provides minimum built in functions. These functions are attached to the G! datatypes and are meant to be as easy to use (and remember) for the developer as possible. One of the most useful and common function is the properties(name1: value1, name2: value2, ...) function, which is used to initialize the Higher Level data type (section 5.2) objects. This function is a good example of when using keyword arugment functions is helpful.

Example:
Sprite airplane;
//initialize the airplane object
airplane.properties(image: "image_path, 3, 2", height: 300, width:
150, posx: 235, posy: 150, group: "hero", animate: true, loopanim:
true, <parent:playfield>);

Depending on which properties are set, G! will dynamically determine which type of Sprite eventually instantiate. For example, if the programmer sets the animate attribute to true, then G! will (under the covers) instantiate an AnimatedSprite. The underlying library has several distinct types of Sprites that must be used when the programmer wants specific functionality. G! frees the programmer from having to memorize what type of Sprites can do what, and

figures it out for us at compile time. More built-in functions will be added as and when more functionality is developed in the language

3.3.4 User-defined Functions

G! allows users to add their own functions which implements user-defined functionality. These functions can be defined and used as per the specifications of sections 4.1 and 4.2. Functions do not need to be declared (like variables and objects) before being defined but they need to be preceded by the *func* keyword. Also, a function defined in one file can be used in another file by simply including the file with the function definition in the file where it needs to be used.

3.4 Data Types

The data types in G! can be classified into 2 distinct types.
The first group consists of basic data types:

3.4.1 Basic Data Types

Integer
These are 32-bit integer numbers corresponding to the 'int' data type found in Java

Double
These are 64-bit double precision numbers analogous to the 'double' data type found in Java.

String
Variables belonging to the 'String' data type consist of a sequence of alphanumeric characters.

Boolean
The boolean data type is used to declare variables that consist of one of the following values- the logical *true* or the logical *false*.

3.4.2 Higher level Data Types

The higher level data types provided by G! consist of the following data types:

Sprite
By definition, a sprite is a small graphic that can be moved independently around the screen, producing animated effects. A sprite is primarily a game object that is placed on screen and that can be manipulated on the screen as per the developer's code. It can contain a static image holding the image of a stationary car, or can contain a gif or a png file containing the animated image of an airplane with its propellers rotating.

Sound
This data type is used to declare variables that point to an audio file having the midi or wav file format. A game would be incomplete without any sound effects. Hence, we provide the sound data type to add an audio element to games.

SpriteGroup
The SpriteGroup data type is used to group a number of sprites as one object to make sprite manipulation manageable.

Playfield
The Playfield data type is used to declare the canvas on which all the gaming action takes place. It is the stage on which all the game objects are displayed. It is can be the board on which a card game is played. It can be the inter-galactic space between the star Orion and the star Vega where the protagonist of the game Captain Langda Tyagi is trying to destroy a bunch of ALF's (Alien Life Forms). The only limit is the developer's creativity.

Coordinate
The Coordinate data type is one extremely useful component in the language. It facilitates the ease of development by providing the programmer with a ready to use data type that refers to the Cartesian X- and Y- coordinates of a point on the Playfield.

3.5 Statements

3.5.1 Structure of a Program

G! programs have a somewhat lenient logical order. One constraint though, is that every object needs to be associated with some properties before it can be used (initialization). After that, the logic can be described using synchronous statements (similar to a lot of common programming languages) as well as asynchronous statements, since this is a very event based language. The compiler will look at the program as a whole to determine the logic of the language. However, the sequence in which variables or object are initialized and modified depend on the order in which they are accessed. Hence, while it is possible to define collisions, key press events or mouse click events for a sprite asynchronously, the x and y positions of the sprite would depend on the order in which those properties are set in the program.

In simpler terms, asynchronous statements don't need to be in any logical order while other statements need to follow their logical execution order. See section 7.2 for more information on types of statements. All statements written inside blocks as well as global declarations must terminate with a semi-colon (;). There are various types of statements in G!

Empty Statement

The Empty Statement does nothing. It's denoted by a semi-colon.
Example:
;

Expression Statements

These statements evaluate an expression and set values of variables. Declarations fall under this category. Expression statements always end in a semi-colon.

Conditionals (if/else)

Conditionals are used to make decisions to decide the flow of the program. G! has the if/else conditional. It is used as followed:

```
if (condition) {
//block 1
} else {
//block 2
}
```

If the condition or conditions defined in condition are fulfilled then block 1 is executed or block 2 will be executed. There are 2 variations of the if/else statements. The first one involves only using the if statement without the else part. The second one involves using else if.

Example 1: //using only the if statement

```
if (x>y) {
print "x is greater";
}
Example 2: //using if/else
if (x>y) {
print "x is greater";
} else {
print "y is greater";
```

```
}
```

Example 3:

```
//using if/else if
if (x>y) {
print "x is greater";
} else if (y>x){
print "y is greater";
} else {
print "x equals y";
}
```

Asynchronous Statements

One of the unique features of G! is the use of asynchronous statements. The when keyword is used to describe an event, which is handled asynchronously in G! To avoid confusion between the if and when statements consider the following example:

```
if(x < y) {
do thing1
}
when (y < z) {
do thing2
}
```

When the above code is executed and the program comes to the line of the if statement, it checks whether x < y. If that is true, then thing1 is done. If that is false, thing1 is not done. On the other hand, if the programmer writes the when statement, then whenever y<z evaluates to true in the whole program, thing2 will be executed at that point. When codeblocks don't wait to be called after the statements that precede it execute, the can defy space and time and the codebock runs *whenever* its boolean conditional evaluates to true.

Since key press, mouse press, collision etc are common in designing games, the incorporation of asynchronous statements should prove a handy tool for the programmer. Asynchronous statements, like synchronous ones, have scope. They are only valid inside the block in which they are defined. However, asynchronous statements defined in the main() method are by default global. One thing to note is that if multiple asynchronous events are mapped to the same event, then they should be done only if the programmer is aware of what he or she is doing. For example, mapping the UP

key press event to different sprites using the when statement within the same block, might give rise to unexpected results in the game.

Loops and Control Statements

There are 2 different loop statements available in G! They are: for loop and while loop. Both of these are defined as blocks as discussed in section 7.5.

The basic structure of a for loop is:

```
for variable = initial value: final value<: increment> {
one or more G! statements
```

```
}
```

This loop is executed from the time when the initial value of the variable goes from initial value to the final value in increments of increment which could be positive or negative. Note that specifying increment is completely optional and the default value is taken to be +1. Unless there is a break or continue statement or unless a statement throws an exception (see section 7.4), all the statements within the loop are executed over and over until the variable equals final value.

Example:

```
for i = 10:100:10 {
print i; //this will print 10, 20, 30,...,100
}
```

Also, variable in a for loop (i in the above example) does not need to be declared as int. G! assumes that variable will always be an int.

The structure of a while loop is:

```
while(condition) {
one or more G! statements
}
```

This block of while loop is executed as long as the condition defined in condition is met. As soon as the condition does not meet, the loop will stop executing. Control statements defined in section 7.4 also determine the abrupt finishing of the block while should be defined as per section 7.5 the condition can include one or more conditions separated by a logical operator. Unlike the for loop, all the variables used in the condition part of the while loop must have pre-declared data types and all the conditions must be defined clearly (no optional parts to the condition).

Example:

```
int x = 0, y = 20;
while (x<10 AND y = 20) {
print x:y;
//this will print 0:20, 1:20, 2:20, ... , 9:20
x=x+1;
}
```

Control Statements

The break, continue and return commands can cause a transfer of control that might prevent a normal completion of statements that contain them. Also, since statements are not defined in a logical order and the compiler looks at the program as a whole to determine the logic (Section 7.1), there might be a case where two or more statements contradict each other cause ambiguity. This might result in a compile time error or run time exception being thrown (due to evaluation of certain expressions), which might result in transfer of control that might prevent normal completion of statements.

Abrupt completion of a substatement will cause abrupt completion of the statement containing it for the same reason. All succeeding steps for the normal completion of the program are

ignored. On the other hand, if all expressions evaluate and all substatements complete normally, then a statement will complete normally.

6.5 Blocks

A block of statement is a group of statements as well as variable and object declaration within braces. Each block should fall under the category of a function, a loop, or a conditional.

```
Block: {
Variable declarations
Statements
}
```

When a block is executed, then each statement is executed before exiting the block. One exception is the use of a control statement. Any control statement might cause the block to not execute completely, which might in turn give rise to exceptions or errors.
Blocks determine the scope of variables. Any variables declared inside a block or captured as function arguments would be accessible only within that block or any sub-blocks defined within that block. Additionally variables can be declared as global outside of any blocks.

Chapter 4
Project Plan

4.1 Team Responsibilities

The G! initiative officially began on some Tuesday or a Thursday in September 2006, when the team members decided to meet in the Carleton Lounge after PLT class to brainstorm an idea for the language. In spite of the varied schedules of each of the group members, we tried to meet once a week to discuss the progress. Even when the weekly meeting wasn't possible, we tried to coordinate the various activities via email.

The duties and tasks for each member are enlisted below:
Steve Lianoglou – Team Leader, Language Maven; parser and front end
Divya Arora – Project Manager; Architecture, front end and documentation
Amortya Ray – System Architect and Integrator; Documentation, back end and testing
Rachit Parikh – Testing and Documentation; Back end and testing

Steve worked on a Mac and used the Eclipse IDE. Divya, Rachit and Amortya worked on Windows machines with Eclipse 3.2.1. Any changes to the code were first thoroughly tested on the local machine. Only after all the unit tests and the regression test suite passed successfully, were the changes committed to the repository. Also, after each commit, a mandatory email was sent out to the group, alerting everyone about the changes made and the possible repercussions to other people's code.

4.2 Coding Conventions

4.2.1 Antlr Conventions

The colon ':' is placed at the fourth column of the line following the string preceding the colon.
The semi-colon ';' is placed at the fourth column of a new line following the last rule/ action line of the clause.
Token names are in upper case and non-terminals are in lower cases.

4.2.1 Java Conventions

All 4 team members were used to a certain style of programming and documenting the code. We tried to stick as much as possible to standard javadoc documentation conventions.

> All G! datatype classes are prefixed with 'Gb' so as to distinguish them from regular java datatypes.
> Variable names are in lower case.
> The left brace '{' occupies the last position on the line preceding the block it encloses.
> The right brace '}' occupies a full line and is at the same column position as the first character of the block.
> Method names are English words whose first character is in upper case, except for the first word.

4.3 Software Development Environment
All of the code, apart from the lexer and the parser, are written in Java. The lexer and parser are witten in Antlr and translated to Java.

4.3.1 Operating Systems
Since all of the code is eventually translated to Java, the code executes on any machine that runs the Java Virtual Machine.

4.3.2 Java 1.5
We used Java 1.5 for compiling the code. We chose to use Java primarily because the GTGE library that we're using is in Java. Also, the architecture neutral and portable nature of Java facilitates the reaching of two of the goals of G!

4.3.3 Antlr
We developed the language lexer, parser and tree walker using **ANTLR** - *ANother Tool for Language Recognition*, which is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java actions.

4.3.4 Version Control Repository
Subversion is an open-source version control system. Subversion manages files and directories over time. Trac is a web-based software project management and bug/issue tracking system emphasizing ease of use and low ceremony. It provides an interface to the Subversion revision control systems, integrated Wiki and convenient report facilities.

4.4 Project Timeline
Here is the timeline for the key project milestones through the semester.

September 26th, 2006	Language Proposal
October 19th, 2006	Language Reference manual
Last week of October	Grammar
2nd week of November	Front End
1st week of December	Back End
2nd week of December	Error Control, Testing and Documentation
December 19th, 2006	Final Project

4.5 Project Log

September 22nd, 2006	CVS repository initialized
September 24th, 2006	Initial draft of white paper
September 26th, 2006	Final draft of white paper
October 10th, 2006	First draft of LRM
October 16th, 2006	Second draft of LRM
October 19th, 2006	Final draft of LRM
October 22nd, 2006	Work on grammar begins
November 2nd, 2006	First version of parser complete
November 13th, 2006	Second version of parser complete
November 13th, 2006	Work on tree-walker begins

November 29th, 2006	Initial tree walker complete
December 10th, 2006	Final tree walker complete
December 10th, 2006	Code generation begins
December 18th 2006	Documentation begins
December 19th, 2006	Project complete

Chapter 5
Architecture Design

5.1. Compiler Structure:

The main crux of our G! Compiler, like any other compiler, is to accept a source file (*.g! text file) from the user and walk through it to produce a "compiled" program. The compiled program here is actually a java source file which knows how to utilize the Golden T Game Engine Library. Like most languages, G! consists of statements and functional blocks, but unlike any other compiler, the G! compiler has a head on its shoulders and a brain that lies in that head, orchestrating all the body blocks to work in symphony to get the compiler going.

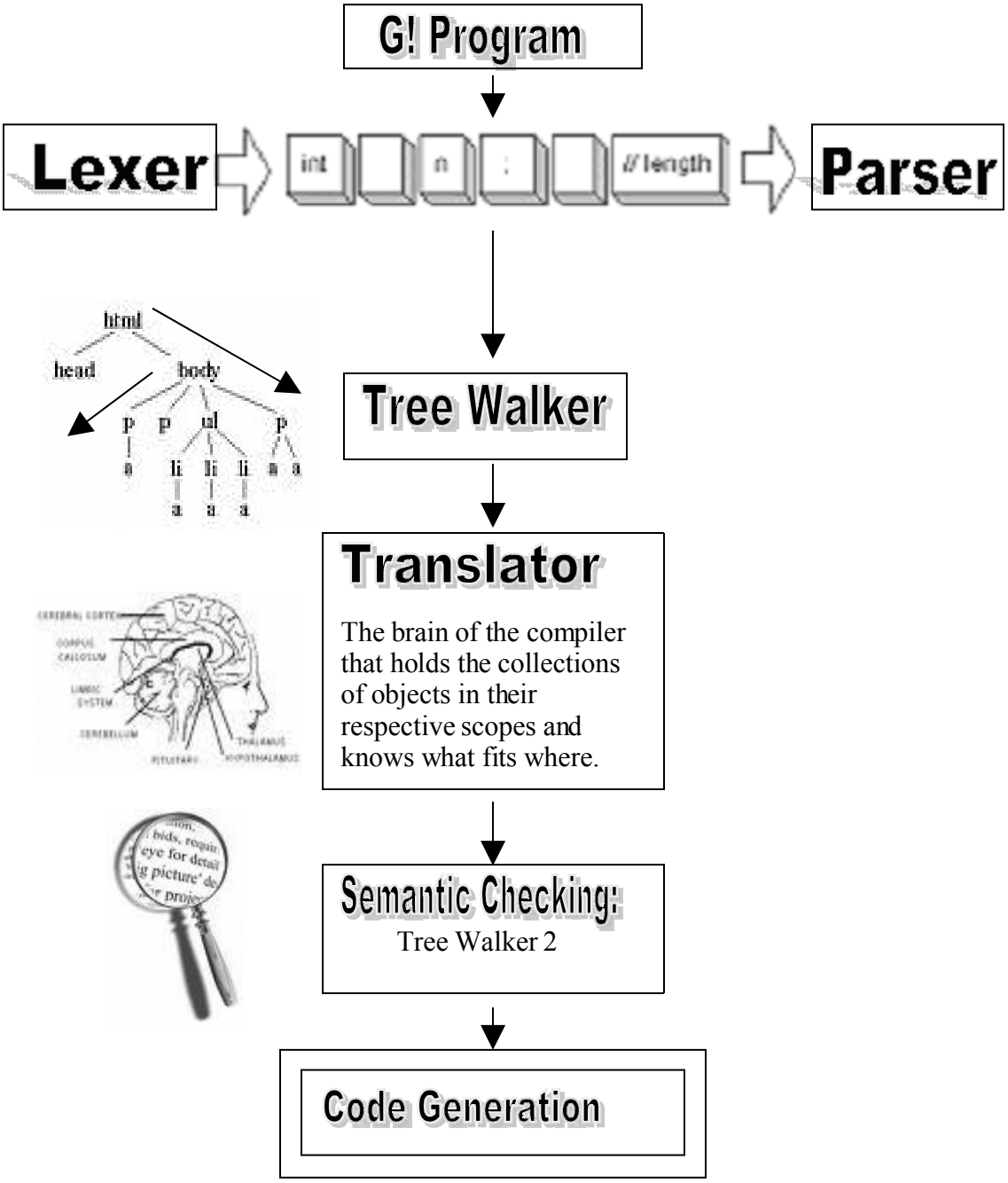
5.1.1. Compiler Workflow:

Here's a very general overview of what really happens within the compiler:

We begin with a .g! file which is passed to the GbLexer.
The GbLexer recognizes each token from the program file and does away with the unwanted components of the program such as the white-space characters,the comments, etc and passes the collection of the tokens to the GbParser.
The GbParser uses the grammar specification provided in the grammar.g file to put the tokens together in a meaningful way and finally produces an Abstract Syntax Tree which is basically a tree representation of the g! program.
This tree is then walked on by the first GbWalker, which is responsible for traversing the tree and building a collection of the various elements that it "sees" while it walks. This is the most important step of our compiler and it includes the handiwork of two main components, the GbWalker and the GbTranslator. The GbWalker can be pictured as a sorter which looks at each node and decides which collection it belongs to. The GbTranslator, on the other hand, acts as the owner of each of the collections thus formed, and it "knows" what every collection is and where it fits in the scheme of things.
At the end of the first walk, we have our g! program completely scrutinized and decomposed into a collection of key elements, which we discuss in more depth later.
At this point, the collections are walked upon by the Semantic Walker, a second walker that does the type checking on expressions and scope checks on variables and function calls, thereby declaring them valid to be written out.
With the checks and collections in place, the last step is to take each object from every one of the above collections, and have the GbTranslator write it out to the appropriate part in the output file. The reason why we need the GbTranslator to know about the details of each collection and where to put it is because G! is free-form whereas Java is not and every program statement in the G! file has a very specific target location in the Java file.

5.1.2 Compiler Components and Phases

The above sequence of events is captured in the block diagram shown below:



The first two blocks, Lexer and Parser were implemented using Antlr which produced the GbLexer.java and GbParser.java files. We shall now discuss the tree walker, the translator, the semantic checking and code generation phases in detail.

5.2. G! v/s its Java Equivalent

To understand what is really happening here, we need to understand how our G! and its Java equivalent differ and what are the things we need to tackle. As mentioned before G! is a free form language. It allows the programmer to write programs that involve variable declarations and assignments, function definitions, if-else statements, while and for loops and an asynchronous statement type "when". The corresponding java program for this free form language has a well defined format, that begins with a bunch of class level declarations for each of the game objects being used in the game, followed by their initializations within an initResources() method, which is basically called once when the application is run to instantiate each of those objects and add them to the game in some user-defined setting. This is then followed by an update method which captures all the when statements, which are asynchronous statements in that they are not a part of the serial execution of the program and the compiler is always on the lookout for the condition expression of the "when", and tells what is to be done when it is seen. An example would be the keyPressed events in the game that control the game objects. The update method is followed by the render() method that is responsible for the graphics of the game, finally followed by the main method that loads the game. Hence translating from the G! program to the Java file would mean much more than just taking one statement at a time and spitting out its Java code. It would mean that we need to somehow make the compiler aware of each and every statement, register what it does, and where it fits into the Java equivalent.

5.3. Compiler Goals

From the above description, we can summarize the goals of our translator as follows:

Find the collection of different statement types in the program:

To be able to correctly map the g! code into the correct Java statements plugged in the correct section of the Java file, it is very essential to know exactly what different types of statements form the program and to collect all such statements in the program. In other words, our compiler has to be content-sensitive.

Preserve the scope of each of these collections:

Consider that the program has a "when" statement, within which lies an "if" statement. If we simply took every new type of statement that we came across and added them to our single common collection, then the translator will put these as two separate statements in the global scope. Hence the scope preservation.

Know what to do with each of these objects in the collection types:

Once we have the collection of objects with their correct scopes ready, we need to know be able to associate each of these statements to its correct target location in the output java file.

Do static/ semantic analysis of the program:

So far we've just considered each statement as an "object" of some particular type. We also need to check if these objects are really "valid" objects of that type, before we pass them into the Java world.

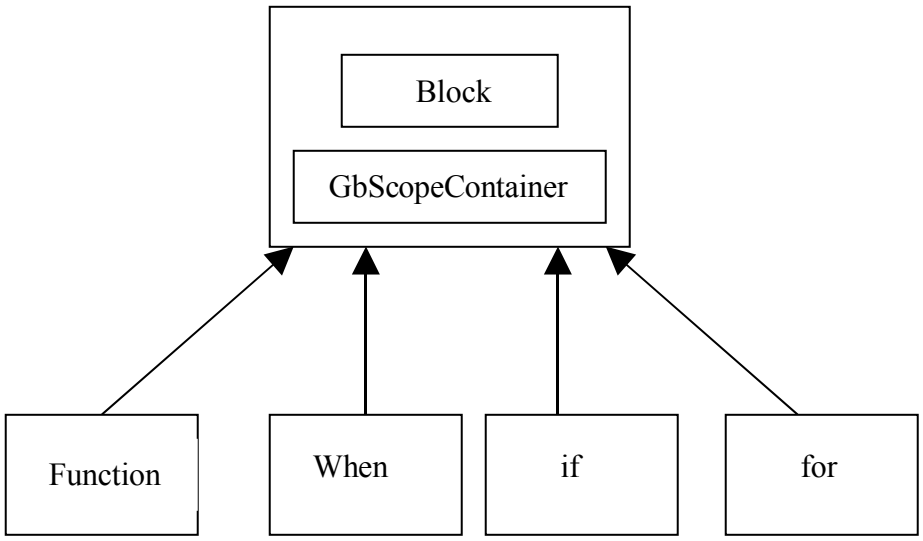
Produce a java equivalent and write out the java file with each of the objects written to the right area of the java file.

5.4 Our approach for the solution:

5.4.1. Collection of Program Components:

In our walker phase, we walk through each node to find a match to the start of one of the possible types of statements. As soon as we find this match, we take the entire subtree starting from that point, and pass it to the corresponding class for that statement type, so that it can form an object of that type of statement. The classes for the different statement types, internally know how to treat the subtree to obtain its various components. For example, while walking the tree, say we come across a node that reads “if”. The walker recognizes this as the beginning of the If Statement, and passes the subtree from that point to the If class constructor. The If class constructor knows that whatever follows the “if” keyword, must be the condition clause, which should be an expression, and it calls the Expression class constructor to return the object of the type expression, which it assigns to its condition component. Next, the if class knows that whatever follows the condition, is the body of the if clause and it calls the “Block” constructor to return a “block” object which is essentially a block of statements.It binds the returned block object to the body of the “if” statement. The if class thus returns and object of the type “if” to the walker. The walker then adds this “if” object to the collection of statements in the appropriate scope.

Since we have four different types of statements which make use of the block object, namely “if”,”for”,”while” and ”when”, we create a class “Block” which takes care of creating a block of statements, and we make the classes for each of the four statement types extend this block class.



5.4.2. Scoping:

Since we’re not interpreting the code or translating it as we walk and instead making collections of the different statement types, the compiler must know which statements fits in where, when it is to be written out finally. This means that apart from storing information about the type of statement, we also need to store information about the scope

of the information. We do this by using a GbScopeContainer interface which is implemented by each of the different statement classes. Say we have an "if" statement. Now when the walker walks on the node "if", it recognizes that it is an "if" statement and calls the constructor of the "if" class to return a statement object of the type "if". The constructor in turn knows that whatever follows the if, is an expression and calls the expression class to form the condition. Similarly it knows that the remaining part of the tree is going to be the body of the "if" and calls the Block class to make an object of the type block which is a collection of the block statements. Now essentially the block class has to recognize each statement within the remaining part and call the corresponding classes to build those statement objects. Once the statement objects are built, the block class then adds these statements to a collection. But instead of adding it to the global collection of statements, it add it to something called as the current scope. Now what is this current scope. The current scope is variable that each of the classes implementing the interface GbScopeContainer have access to and now, whenever such a class is called, it is capable of setting the current scope to its local scope at the beginning of the construction and resetting it back to the parent scope when it leaves the constructor. This enables all the classes and the walker to be independent of worrying about the scope and they all use a single statement this.currentscope.addStatement and the statements magically go into their respective scope. At the end of the first walk, we have a comprehensive collection of statements and within their scopes any sub-statements that they might have and so on.

5.4.3. Know Target Destinations of our collections:

Like we already described in the G! v/s Java section, each of the objects in our collection of statement objects, has a very specific place where it fits into the corresponding Java equivalent. The statement may need to be decomposed into more than one statements going to different parts of the target Java program. Here's a brief summary of what goes where:

The Global Scope Declarations enter the very first target area, the class level declarations within our target game class.

The assignments to all these global scope variables and the instantiation of all the global GBang objects happens in the initResources() Method of the target file.

Now statements of the type if, for and while will also make it to the initResources but they shall follow the assignments. These statements are normally used for setting up the Game Field, where the user specifies what goes on the playfield and where.

Then come the When statements. The when statements typically simulate asynchronous executions in a synchronous world. They go into the update method() of the Java program.

Some special operators like the ! operator break into a set of statements that go to various parts of the program. Typically, a bang expression requires the following:

- Declaration of a collision manager for its operands at the end of the global scope declarations.
- Instantiating the Collision Manager in the initResources()
- A new class for it following the main Game Class, within which a method called collide() provided by the GTGE library may be overwritten with the code in the When body.

5.4.4. Static Semantic Analysis:

The GbangWalker does everything differently and the Static Semantic Walker is no exception. So instead of having the semantic checking done during the first walk itself or have a second walker make a complete pass over it after the first pass, the requirements of G! dictated a very unique semantic checking phase. So as concluded, the first phase has a complete collection of statement objects collected by the first walker. In this collection, at the lowest level of granularity lie the expressions. During the first walk, all we did with the expressions, was made expression objects which mainly held back on the expression tree and did nothing much with it. The reason behind this was that the expressions held those operators that dictated the code generation phase, apart from other general operators. So once we are done with the first phase, its time to resolve these expressions to get the code generation phase rolling. It is at this point that our GbangTranslator calls on a checkSemantics() method that passes very specific components to the Semantic walker to check if they are semantically sound and legal. The checkSemantics() method is overridden by every class, such that each class will take care of recognizing what components it holds and accordingly calls the correct part of the semantic tree to walk on it. Within the static semantic checking, we basically type check each kind of expression to see if all the operands are compatible with the operators, if the right hand side of an assignment is legal for the corresponding left hand side, if a function call actually calls a predefined function found within Function Table of the global scope, if the variables being used have been defined or not, whether they are accessible from the current scope or not. We also have some checks made during the first walk to check if certain types of nested loops are legal. For example, we do not allow a "when" loop to be within a function definition but we do allow those within "if" statements.

5.4.5 Code Generation

Code is generated through the GbTranslator.java file. This class calls various methods which are defined in the Global Template class. At this point, since we have already walked though the code and have stored everything within statement and block classes, all we need to do is spit out the code at the right places in the java program. It is done with the help of these static methods described here:

GlobalTemplate.header(GbTranslatort)

Writes all the import statements to java.

GlobalTemplate.instanceVars(GbTranslatort)

Declares all the global variables in java

GlobalTemplate.initResources(GbTranslatort)

Generates the initResources() method in java which has all the initialization code

GlobalTemplate.updateMethod(GbTranslatort)

Creates the update method which handles asynchronous statements

GlobalTemplate.renderMethod(GbTranslator t)

Creates the render method for printing on the game console

GlobalTemplate.renderFunctions(GbTranslator t)

Creates methods that are called through the render() method

GlobalTemplate.footer(GbTranslator t)

Creates the main method in java which initializes the class and calls the start() method which starts the game

GlobalTemplate.collisionClasses(GbTranslator t)

Generates subclasses that override the collided(Sprite s1, Sprite s2) method. For each collision there is a subclass that needs to be defined. So everything that's defined in a when statement with a ! type expression goes in the collided method here.

Chapter 6
Testing Plan

6.1 Goal

The goal of the testing phase was to ensure that any code written functioned exactly as it is expected to. Also, any incremental changes to the code should not affect any previously functional code. Hence, the goal of our testing phase was to make sure that we obtained correct output during each phase of the language development process, and didn't trample over previously written code as we made progress towards adding new functionality.

Our regression test suite was born very early in the project, and had a rebirth at about half way through the semester. At first we had had implemented a (seemingly) flexible test runner that would exercise the G! grammar at a very granulate level. This file is located at `gbang.antlr.grammartest.AllProductionGenericTestRunner.` The system would iterate over all of the folders in the test/lang_constructs folder. The name of each folder in that path would match 1-to-1 with a production rule in the G! grammar. The files in those directories would look like the snippet below found in the test/lang_constructs/if_statement folder:

```
if ((a == b)) { // pass
}

if (a OR b OR c AND D) { // pass
/*nothing*/
}

if ((a OR b) OR (c AND D) { // fail
/*nothing*/
}
```

These tests would be run against the "if_statement" production of the grammar. Lines marked with "// pass" were expected to pass, and "// fail" were expected to fail. Testing for syntax that should fail (and have the parser throw an exception) is just as important as testing your parser to ensure that it lets valid constructs go through.

As our grammar developed (and the names of productions changed), this implementation was a bit tedious to maintain since the names of the folders were tied to the name of the production that was under test. Compound that with the fact that our folders were under version control (and thus changing their names to suite the productions of our grammar would be a really bad idea), so unfortunately we had to can this version.

The second version of our grammar regression test suite consists of the AllProgramsTester file. This program would just run through all of the folders (and sub-folders) in a given directory and execute them to ensure that ANTLR didn't throw any exceptions as they were parsed. If no errors were encountered, this would be considered a successful test case.

If the program came across a folder entitled "fail," it would run all of the g! source files in that folder and expect them to throw an error (for some malformed syntax). If the parser

didn't throw an exception on the files in these "fail" directories, then **this would be an error** and the system would report the results to the console.

Since the G! compiler was responsible for writing out java source code, it was also necessary to check that this source was clean.

We had built a program that writes out a temporary java file, which then gets compiled on the fly (gbang.JavaCompiler). This class is invoked by the WalkerTest file and passed the path of a g! source file to compile.

The best we could do, in this case, was to only report system.out and sytem.error from the javac call. There was no other way to programmatically invoke the java compiler and catch its exceptions as it tries to compile. The recently released Java6 does have this functionality, however, which we can use as an improvement for the next release of our language.

Luckily, our development envirnoment (Eclipse) comes with very tight JUnit integration. Ensuring that all tests are run together (not just the "Grammar Production Tests") was easy since we can have one main TestSuite which can call into and across to each other, which allowed for the easy centralization of our all our tests into the click of the green button.

6.2 Implementation

As stated earlier, the purpose of the AllProgramsTester class was to exercise tests against the files in the test/test_programs directory.

The tests are split into two categories:

Tests that should pass; and
Tests that should fail.

Phase 1:
This class will first run through all of the *.g! files located in the test/test_programs/subdirs/*g!

Each one of these files should pass. If there is the parser cannot correctly parse one of the files, it will capture the error and report it after all tests are run.

Note that during this phase of testing, only the first test that fails in the *.g! file will be reported. Hence it is possible that many tests after will also fail, but we can't continue to find them, the solution to this is to:

Fix the reported error and run again;
Move the tests that are below the test that failed into a new test file in this way, the parser can try the new file and report on the errors there

Also, the way this is designed lends to test programs that are small and many. This is why the test/test_programs folder is laid out with sub folders. This will help to keep the tests relatively clean and easy to find.

Phase 2:
The second phase is to test files that have syntax that should fail. Every subfolder in test/test_programs has a folder called ‘fail’. The second phase of this class will descend into each sub-sub folder named ‘fail’ and make sure that whatever is in there should fail to parse cleanly. If it doesn't fail to parse cleanly, it will report this as an error since something that we didn't intend to be ‘clean’ g! syntax fools the parser.

Chapter 7
Lessons Learnt

Steve:
Designing languages is hard! The decisions we made early on in the semester stuck around to haunt us as progress continues on the project. I can easily say that I've gained a greater appreciation for the importance of design plays in development. Usually we can "hack and move" through a project, but when designing a language is at stake, each decision is so important because it will manifest itself in other aspects of the language (either in its implementation, or its "feel"). Language design is definitely hard, but easily one of the more rewarding school projects thus far. Usually we just make a piece of software for a project and just sort of "look at it." Now that we have a language, it's something that we can use in new ways every time we sit in front of the computer. The fact that it's a gaming language makes it even better.

Amortya:
Among the several things that I learnt while working on this project, the most important of those would be how to handle group dynamics! Our group comprised of very different people- 2 Masters students, an undergrad and another Masters student whose schedule was hugely different from the others. Hence scheduling meetings became a major pain! Also, being one of the biggest projects that I worked on ever, it taught me the importance of communication between group members. Also, the decisions we made at the start of the semester have severely affected our efficiency or rather crippled it later during the crunch time. Hence, a better grip over the time management and analysis phase would have helped.

Rachit:
This course gave me a really good insight into what goes on behind the scenes (atleast at the lexer / parser/walker levels) when a piece of code is executed. My appreciation for programming has grown as a result of this course, since now I have a much better idea of how the code that I write would be translated and compiled and how I will get the desired output. Writing a good grammar and walker is definitely not a simple thing to do and I was able to learn the nitty gritty of compilers. Besides now when I come across a new language, I'll have a fairly good idea of what code to test to make sure the language is robust!

Classwork aside, I really enjoyed working with my team and our project turned out to be fun and appreciably successful only due to the teamwork and understanding between all four members. I would like to thank Prof. Edwards, the TAs and the G! team for making this a really memorable experience of my undergrad career.

Divya:
The PLT project was one of the most fulfilling projects in my engineering career considering it was the first ever project where we built something of a magnitude so large. Putting together all the different things that go into a compiler makes one appreciate the fine nuances that go into designing and it was an eye-opener of sorts. The most important lesson that I learnt was everything that ever touches your head is important. I remember times when the four of us would just sit together joking about something and it turns out

that most of our project is inspired by such jokes! What I also learnt was that each person brings different strengths and perspectives to a project and working in a group taught me what wonders synergism can bring in achieving something so complex. At the end of the day, I am glad that I took this course.

Chapter 8
Language Syntax

8.1 Lexical Rules

```
class GBangLexer extends Lexer;
options {
      k = 2;
      exportVocab = GBang;
      charVocabulary = '\3'..'\377';      // LATIN charset only (sorry
international unicode)!
      testLiterals = false;

      // The option below causes ANTLR to not catch exceptions that are
generated while
      // parsing the grammar. Our unit test framework works in such a
way that
      // it requires the parser to throw exceptions on malformed syntax
      defaultErrorHandler = false;
}

protected
Letter
      :     'A' .. 'Z'
      |     'a' .. 'z'
      ;

protected
Integer
      :     '0' .. '9'
      ;

protected
NonZeroDigit
      :     '1' .. '9'
      ;

protected
DoubleTrailer
      :     '.' (Integer)*
      ;

WS    :     (     ' '
            |     '\t'
            |     '\f'
                  // handle newlines
            |     (     options { generateAmbigWarnings=false; }
                  :     "\r\n"      // DOS / WIN
                  |     '\r'  // Macintosh (but old-school mac, I don't
think we need this)
                  |     '\n'  // Unix (the right way)
                  )
                  { newline(); }
            )+
            { $setType(Token.SKIP); }
```

```
      ;

SL_COMMENT
      :     "//" (~('\n'|'\r'))*
            {$setType(Token.SKIP);}
      // newline is matched by the WS rule
      ;

// multiple-line comments
ML_COMMENT
      :     "/*"
            (     /*    '\r' '\n' can be matched in one alternative or
by matching '\r' in one iteration and '\n' in another. I am trying to
handle any flavor of newline that comes in, but the language that
allows both "\r\n" and "\r" and "\n" to all be valid newline is
ambiguous. Consequently, the resulting grammar must be ambiguous. I'm
shutting this warning off. (Taken from Java1.5 grammar from antlr
website) */
                  options {
                        generateAmbigWarnings=false;
                  }
            :
                  { LA(2)!='/' }? '*'                        // match *
as long as its not followed by /
            |     '\r' '\n'         {newline();}
            |     '\r'              {newline();}
            |     '\n'              {newline();}
            |     ~('*'|'\n'|'\r')
            )*
            "*/"
            {$setType(Token.SKIP);}
      ;

ID    options { testLiterals = true; }
      :     (Letter | '_') (Letter | Integer | '_')*
      ;

// taken from prof's simple language antlr demo
// this allows for a " to be embedded into a quote-surrounded string
// in this fashion: "He said, ""Why Hello"" to his friend"
// --> putting to double quotes together
STRING
      :     '"'! ('"' '"'! | ~('"'))* '"'!
      ;

CHAR
      :     "'"! (Letter)+ "'"!
      ;

NUMBER
      :     '0'
            (     DoubleTrailer { $setType(DOUBLE_NUM); }         //
0.something isn't an int
            |     /* nothing */ { $setType(INT_NUM); }            // 0
is
            )
      |     NonZeroDigit (Integer)*
```

```
            (     DoubleTrailer { $setType(DOUBLE_NUM); }   // 1033729.
isn't an int
            |     /* nothing */ { $setType(INT_NUM); }            //
1033729 is an int
            )
      |     DoubleTrailer { $setType(DOUBLE_NUM); }               //
.873837 is a double
      ;

// Operators
LPAREN: '(';
RPAREN: ')';
LBRACK: '[';
RBRACK: ']';
LCURLY: '{';
RCURLY: '}';
COLON: ':';
COMMA: ',';
PROP: "->";
ASSIGN: '=';
EQUAL: "==";
NEQUAL: "~=";
SLASH: '/';
PLUS: '+';
MINUS: '-';
STAR: '*';
PERCENT: '%';
CARAT: '^';

// AUGMENTING ASSIGNMENT
SLASH_ASSIGN: "/=";
PLUS_ASSIGN: "+=";
INC: "++";
MINUS_ASSIGN: "-=";
DEC: "--";
STAR_ASSIGN: "*=";

// COMPARISON
LEQ: "<=";
LT: '<';
GEQ: ">=";
GT: ">";
BANG: "!"; // !left, !right handled by parser
BTOP: "!top";
BRIGHT: "!right";
BBOTTOM: "!bottom";
BLEFT: "!left";
BL: "#L";
BR: "#R";

// OTHER
SEMI: ';';
AT:   '@';        // @left, @top, etc. get weeded out by parser
```

8.2 Syntactic Rules

```
header {
      // use this area to specify a package for the resulting
parser/lexer
      // and any imported classes
package gbang.antlr;
}

class GBangParser extends Parser;
options {
      k = 2;                                    // two token lookahead
      exportVocab = GBang;          // call the vocab "GBang"
      defaultErrorHandler = false;
      buildAST = true;
}

tokens {
      STATEMENT;
      ARGS; DARGS;
      TYPE; ARRAY_SLICE;
      KWARGS; DKWARGS;
      PROP_OF; PRINT_CALL;
      FUNC_CALL;
      UNARY_MINUS; UNARY_PLUS;
      INDEX_OP;
      POST_INC; POST_DEC;
      EXPR;
      ID;
      BANG_COMP;
      DOUBLE_NUM;
      INT_NUM;
      ARRAY_DECLARATOR;
      VARIABLE_DEF;
      PROG;
      FUNC_DEF;
      BLOCK;
      START_FOR; END_FOR; STEP_FOR; FOR_TYPE; FOR_BODY; FOR; FOR_RANGE;
      IF; ELSE; IFBODY;
      AT_OP;
      KEY_PRESS;
}

program
      :     (statement | func_def)+ EOF!
            { #program = #([PROG,"PROG"], program); }
      ;

statement
      :     expression SEMI!
      |     declaration SEMI!
      |     if_statement
      |     while_statement
      |     for_statement
      |     when_statement
      |     print_statement SEMI!
```

```
      |     control_statement SEMI!
      |     LCURLY! (statement)* RCURLY!
      ;

control_statement
      :     "return"^ (expression)?
      |     "break"^
      |     "continue"^
      ;

for_statement
      :!    "for"^ (t:type)? id:ID ASSIGN! start:expression COLON!
stop:expression
            (COLON! step:expression)? bdy:statement
            {
                  if (#t != null) {
                        #t.setType(FOR_TYPE);
                  }
                  #start.setType(START_FOR);
                  #stop.setType(END_FOR);
                  if (#step != null) {
                        #step.setType(STEP_FOR);
                  }

                  #for_statement = #(#[FOR,"FOR"],t,id,#(#[FOR_RANGE,
"FOR_RANGE"],start,stop,step),#(#[FOR_BODY,"FOR_BODY"],bdy));
            }
      ;

while_statement
      :     "while"^ LPAREN! (expression|key_press) RPAREN! statement
      ;

if_statement
      :!    "if"^ LPAREN! (exp:expression|key:key_press) RPAREN!
bdy:statement
            (     // combat the dangling else -- inspired by the
java1.5 grammar
                  options { warnWhenFollowAmbig = false; greedy = true;
};:
                  el:"else"! els:statement
            )?
       {
            if (#exp == null) #exp = #key;
            #if_statement = #(#[IF,"IF"], exp, #(#[IFBODY,
"IFBODY"],bdy), #(#[ELSE,"ELSE"], els));
       }
      ;

when_statement
      :     "when"^ LPAREN! (exp:expression | key_press) RPAREN!
statement
      ;

key_press
      :     "KeyPress"! LPAREN! c:CHAR { #c.setType(KEY_PRESS); }
RPAREN!
```

```
      ;

////////////////////////////////////////////////////////////////////
// modified from java grammar
declaration!
    :   t:typeSpec[false] v:variableDefinitions[#t]
        {#declaration = #v;}
    ;

// A type specification is a type name with possible brackets afterwards
//   (which would make it an array type).
typeSpec[boolean addImagNode]
    : builtInTypeSpec[addImagNode]
    ;

// A builtin type specification is a builtin type with possible brackets
// afterwards (which would make it an array type).
builtInTypeSpec[boolean addImagNode]
    :   type
        {
            if ( addImagNode ) {
                #builtInTypeSpec = #(#[TYPE,"TYPE"], #builtInTypeSpec);
            }
        }
    ;

variableDefinitions[AST t]
    :   variableDeclarator[getASTFactory().dupTree(t)]
        (   COMMA!
            variableDeclarator[getASTFactory().dupTree(t)]
        )*
    ;

/** Declaration of a variable.  This can be a class/instance variable,
 *   or a local variable in a method
 * It can also include possible initialization.
 */
variableDeclarator![AST t]
    :   id:ID d:declaratorBrackets[t] v:varInitializer
       {#variableDeclarator = #(#[VARIABLE_DEF,"VARIABLE_DEF"],
                                #(#[TYPE,"TYPE"],d), id, v);}
    ;

declaratorBrackets[AST typ]
      :     {#declaratorBrackets=typ;}
            (lb:LBRACK^ {#lb.setType(ARRAY_DECLARATOR);} expression
RBRACK!)*
      ;

varInitializer
      :     ( ASSIGN^ expression )?
      ;



////////////////////////////////////////
```

```
expression
      :     logic_test
            (     augassign logic_test
            |     (ASSIGN^ logic_test)+
            )?
            { #expression = #([EXPR, "EXPR"], expression); }
      ;

augassign
      :     PLUS_ASSIGN
      |     MINUS_ASSIGN
      |     STAR_ASSIGN
      |     SLASH_ASSIGN
      ;

logic_test
      :     and_test ( "or"^ and_test )*
      ;

and_test
      :     not_test ( "and"^ not_test )*
      ;

not_test
      :     "not"^ not_test
      |     comparison
      ;

comparison
      :     arith_expr ( comp_op! { astFactory.makeASTRoot(currentAST,
returnAST); } arith_expr)*
      ;

comp_op
      :     GEQ
      |     LEQ
      |     GT
      |     LT
      |     EQUAL
      |     NEQUAL
      |     BTOP
      |     BRIGHT
      |     BBOTTOM
      |     BLEFT
      |     BANG
//    |     bang_op
      ;

arith_expr
      :     term ( (PLUS^ | MINUS^) term )*
      ;


term
      :     factor ( (STAR^ | SLASH^ | PERCENT^) factor )*
      ;
```

```
factor
      :     power
//    |     MINUS {#MINUS.setType(UNARY_MINUS);} factor
//    |     PLUS {#PLUS.setType(UNARY_PLUS);} factor

      ;

power
//    :     atom! {astFactory.makeASTRoot(currentAST, returnAST); }
(options {greedy=true;}: trailer)*
//    :     atom (options {greedy=true;}: trailer)*
      :     atom
//          (     CARAT^ factor
//          |     at_op! {astFactory.makeASTRoot(currentAST,
returnAST); } // changed
//          )?
            ( CARAT^ factor)?
      ;

atom
//    :     ID
      :     identifier
//    |     NUMBER
      |     INT_NUM
      |     DOUBLE_NUM
      |     STRING
      |     "true"
      |     "false"
      |     "null"
      |     LPAREN! (logic_test)? RPAREN!
      |     MINUS^ INT_NUM
      |     MINUS^ DOUBLE_NUM
//    |     MINUS^ atom
      ;

identifier
      :     ID
            (options {greedy = true;}: lp:LPAREN
{ #lp.setType(FUNC_CALL); } (arg_list)? RPAREN) ?
            (     at:at_op
            |     (array_slice)+
            )?
            (options {greedy=true;}:PROP^ identifier)*
            {
                  if (#at != null) {
                              #identifier = #(#[AT_OP, "AT_OP"],
#identifier);
                  }
            }
      ;

array_slice
      :     lb:LBRACK { #lb.setType(INDEX_OP);} arith_expr RBRACK!
      ;

//trailer
//    :     lp:LPAREN { #lp.setType(FUNC_CALL); } (arg_list)? RPAREN
```

```
//    |     lb:LBRACK { #lb.setType(INDEX_OP);} arith_expr RBRACK!
//    |     pr:PROP^ id:ID // nested accessor
//          { #trailer = #(#(#[PROP, "PROP"], pr), #atom); }
//            #builtInTypeSpec = #(#[TYPE,"TYPE"], #builtInTypeSpec);

//    ;

argument
      :     logic_test (ASSIGN^ logic_test)?
      ;

at_op
      :     AT!
            (     "top"
            |     "right"
            |     "bottom"
            |     "left"
            |     "center"
            )?
//          { #at_op = #(#[AT_OP, "AT_OP"], #at_op); }
      ;

print_statement
      :     "print"^ LPAREN! arg_list RPAREN!
            { #print_statement = #([PRINT_CALL, "print"],
print_statement); }
      ;


// This definition is simple and takes care of args and kwargs
// this doesn't handle the semantics that the arg_list & kwarg_list
// production handle below, but it might be necessary to rever to this
//arg_list
//    :     argument (options {greedy=true;}: COMMA argument)*
//    ;

// remember that required arguments are read down the depth
// of a tree.
arg_list
      :     (ID ASSIGN^) => kwarg_list
      |     expression (COMMA! arg_list)?
            { #arg_list = #([ARGS, "ARGS"], arg_list); }
//    |     /* nothing */
//          { #arg_list = #([ARGS, "ARGS"], arg_list); }
      ;

kwarg_list
      :     ID ASSIGN expression (COMMA! ID ASSIGN expression)*
            { #kwarg_list = #([KWARGS, "KWARGS"], kwarg_list); }
      ;

func_def
      :     "func"! type ID LPAREN! define_arg_list RPAREN! statement
            { #func_def = #([FUNC_DEF, "FUNC_DEF"], func_def); }
      ;
```

```
// Note that every "normal" argument will be a root of a new
// subtree. You will have to dig to the bottom of this tree
// in order to get all of the arguments
define_arg_list
      :     (type ID ASSIGN^) => define_kwarg_list
      |     type ID (COMMA! define_arg_list)?
            { #define_arg_list = #([DARGS, "DARGS"], define_arg_list); }
      |     /* empty */
            { #define_arg_list = #([DARGS, "DARGS"], define_arg_list); }
      ;

define_kwarg_list
      :     type ID ASSIGN expression (COMMA! type ID ASSIGN
expression)*
            { #define_kwarg_list = #([DKWARGS, "DKWARGS"],
define_kwarg_list); }
      ;

//define_kwarg_list
//    :     declaration (COMMA! declaration)*
//          { #define_kwarg_list = #([DKWARGS, "DKWARGS"],
define_kwarg_list); }
//    ;

type
      :     ("SpriteG") => "SpriteGroup"
      |     "Sprite"
      |     "Integer"
      |     "Double"
      |     "String"
      |     "Boolean"
      |     "Sound"
      |     "PlayField"
      |     "Coordinate"
      |     "void"
      ;
```

Chapter 9
9.1 Code Listing

```
=========================================================
src/java/gbang/BaseException.java
=========================================================
/* FILE: BaseException.java
 * $Id$
 *
 * The G! Language
 */
package gbang;

/**
 * Use directly (or sublcass) for GBang Exceptions:
 *
 * Maybe we need differences between SyntaxException and
SemanticException
 * or whatever
 */
public class BaseException extends RuntimeException {

      /**
       * So java doesn't complain ...
       */
      private static final long serialVersionUID = 11905374676112097 72L;

} // class BaseException


=========================================================
src/java/gbang/CLI.java
=========================================================
/* FILE: CLI.java
 * $Id$
 *
 * The G! Language
 */
package gbang;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * helper class to run things from the command line
 */
public class CLI {
      public static void run(String command) {
            try {
                  String ls_str;
                  System.out.println("CLI: " + command);
                  Process ls_proc = Runtime.getRuntime().exec(command);

                  // get its output (your input) stream
```

```
                  BufferedReader out = new BufferedReader(new
InputStreamReader(ls_proc.getInputStream()));
                  BufferedReader errors = new BufferedReader(new
InputStreamReader(ls_proc.getErrorStream()));

                  try {
                        System.out.println("***** RESULT ******");
                        while ((ls_str = out.readLine()) != null) {
                              System.out.println(ls_str);
                        }

                        // report any errors?
                        int errcount = 0;
                        StringBuffer sb = new StringBuffer();
                        while ((ls_str = errors.readLine()) != null) {
                              sb.append(ls_str + "\n");
                              errcount++;
                        }

                        if (errcount > 0) {
                              System.out.println("\n**** ERRORS ( " +
errcount + ") ****\n");
                              System.out.println(sb.toString());

      System.out.println("**********************");
                        }


                        //while ((ls_str = errors.rea))
                  } catch (IOException e) {
                        System.exit(0);
                  }
            } catch (IOException e1) {
                  System.err.println(e1);
                  //System.exit(1);
            }

            //System.exit(0);
      }
}


=======================================================
src/java/gbang/GbFunctionTable.java
=======================================================
/* FILE: GbFunctionTable.java
 * $Id$
 *
 * The G! Language
 */
package gbang;

import java.util.HashMap;

import gbang.statement.Function;
```

```
/**
 *
 */
public class GbFunctionTable {

	protected HashMap<String,Function> table;

	public GbFunctionTable() {
		this.table = new HashMap<String,Function>();
	}

	public void put(String name, Function func) {
		this.table.put(name,func);
	}

	public Function get(String name) {
		return this.table.get(name);
	}

	// sometimes it's handy to be explicit
	public Function getFunction(String name) {
		return this.table.get(name);
	}

	public HashMap<String,Function> getFunctionTable() {
		return this.table;
	}
}


=======================================================
src/java/gbang/GbTranslator.java
=======================================================
/* FILE: GbTranslator.java
 * $Id$
 *
 * The G! Language
 */
package gbang;

import java.util.ArrayList;

import antlr.RecognitionException;
import gbang.datatype.GTGEObject;
import gbang.datatype.GbDataType;
import gbang.datatype.GbIllegalNestingException;
import gbang.datatype.GbPlayField;
import gbang.datatype.GbScopeContainer;
import gbang.datatype.GbSymbolTable;
import gbang.event.GbCollisionManager;
import gbang.statement.Assignment;
import gbang.statement.BangExpression;
import gbang.statement.Declaration;
import gbang.statement.Expression;
import gbang.statement.Function;
import gbang.statement.Statement;
import gbang.statement.When;
```

```
import gbang.template.GlobalTemplate;

/**
 *
 */
public class GbTranslator implements GbScopeContainer {

	protected GbSymbolTable globalScope;
	protected GbFunctionTable functionTable;
	//protected Block statements;
	protected ArrayList<Statement> statements;

	protected GbCollisionManager collisionManager;

	public static GbFunctionTable FUNCTION_TABLE;

	public GbTranslator() {
		this.globalScope = new GbSymbolTable();
		this.functionTable = new GbFunctionTable();

		// bad!
		FUNCTION_TABLE = this.functionTable;

		//this.statements = new Block();
		this.statements = new ArrayList<Statement>();

		this.collisionManager = new GbCollisionManager();

	}

	/**
	 * Each g! file should define only one playfied.
	 * Here's a quick way to get it
	 */
	public GbPlayField getPlayField() {
		GbPlayField obj = null;
		for (String key : this.globalScope.getTable().keySet()) {
			GbDataType test = this.globalScope.get(key);
			if (test instanceof GbPlayField) {
				obj = (GbPlayField) test;
			}
		}
		return obj;
	}

	/**
	 * Iterate over our statements/functions/blocks to ensure
	 * that their semantics are essentially correct
	 */
	public void checkSemantics() throws RecognitionException {
		for (String fname :
this.functionTable.getFunctionTable().keySet()) {
			Function funkDaFunk =
this.functionTable.getFunction(fname);
			funkDaFunk.checkSemantics();
		}
```

```
            for (Statement stmt : this.statements) {
                  stmt.checkSemantics();
            }
      } // method checkSemantics

      //
================================================================================
=====
      //
================================================================================
=====
      // _____________________________________________________________
TRANSLATION LOGIC
      //
================================================================================
=====
      //
================================================================================
=====
      /**
       * Return the GbDataTypes in our global scope.
       * This method is called while we're constructing the
initResources method
       * from the GlobalTemplate.
       *
       * All configuration of target GTGE objects should happen via the
       * properties that are set on each object
       */
      public ArrayList<GTGEObject> getGTGEVars() {
            ArrayList<GTGEObject> vars = new ArrayList<GTGEObject>();
            for (String name : this.globalScope.getTable().keySet()) {
                  GbDataType var = this.globalScope.get(name);
                  if (var instanceof GTGEObject) {
                        vars.add((GTGEObject) var);
                  }
            }
            return vars;
      }

      /**
       * Fetch the statement objects that will need to be written
       * in the initResources method of the target java class
       *
       * This will include everything except func defs (duh) and When
       * statements
       * @return
       */
      public ArrayList<Statement> getInitStatements() {
            ArrayList<Statement> stmts = new ArrayList<Statement>();
            for (Statement stmt : this.statements) {
                  if (!(stmt instanceof When) && !(stmt instanceof
Function) && !(stmt instanceof Declaration)) {
                        stmts.add(stmt);
                  }
            }
            return stmts;
      }
```

```
	public ArrayList<Statement> getUpdateStatements() {
		ArrayList<Statement> lines = new ArrayList<Statement>();

		// harvest the when statements
		for (Statement stmt : this.statements) {
			if (stmt instanceof When) {
				When s = (When) stmt;
				if (s.getTest() instanceof BangExpression) {

				}
				lines.add(stmt);
			}
		}


		return lines;
	} // method getUpdateStatements


	//
_____________________________________________________________________
__
	/**
	 * This is the brains of the operation
	 * @return
	 */
	public String toJava() {
		StringBuffer sb = new StringBuffer();

		// write out imports and class header
		sb.append(GlobalTemplate.header());

		// write out instance variables
		sb.append(GlobalTemplate.instanceVars(this));

		sb.append(GlobalTemplate.initResources(this));

		sb.append(GlobalTemplate.updateMethod(this));

		sb.append(GlobalTemplate.renderMethod(this));
		sb.append(GlobalTemplate.renderFunctions(this));

		// magic here
		sb.append(GlobalTemplate.footer(this));
		return sb.toString();
	}


	// ____________________________________________________________
interface methods

	public void addVar(String name, GbDataType var) {
		this.globalScope.put(name, var);
	}
```

```
	public void addFunction(Function func) {
		this.functionTable.put(func.getName(), func);
	}

	public void addStatement(Statement stmt) throws
GbIllegalNestingException {
//		this.statements.addStatement(stmt);
		if (stmt instanceof Assignment) {
			// something special?
		}
		this.statements.add(stmt);
	}

	public void addExpression(Expression expr) {
//		this.statements.addStatement(expr);
		this.statements.add(expr);
	}
	// _____________________________________________________________
GETTERS / SETTERS
	/**
	 * @return the globalScope
	 */
	public GbSymbolTable getGlobalScope() {
		return this.globalScope;
	}

	/**
	 * @param globalScope the globalScope to set
	 */
	public void setGlobalScope(GbSymbolTable globalScope) {
		this.globalScope = globalScope;
	}

	public GbScopeContainer getParentScope() {
		return this;
	}

	public GbDataType getVar(String name) {
		return this.globalScope.get(name);
	}

	public GbDataType getCurrentVar(String name) {
		return this.globalScope.getCurrentVar(name);
	}

	public void setParentScope(GbScopeContainer parent) {
		// nothing
	}

	public boolean isLegalStatement(Statement stmt) {
		return true;
	}


	/**
	 * @return the functionTable
	 */
```

```
	public GbFunctionTable getFunctionTable() {
		return this.functionTable;
	}


	/**
	 * @param functionTable the functionTable to set
	 */
	public void setFunctionTable(GbFunctionTable functionTable) {
		this.functionTable = functionTable;
	}


	/* (non-Javadoc)
	 * @see gbang.datatype.GbScopeContainer#getSymbolTable()
	 */
	public GbSymbolTable getSymbolTable() {
		return this.globalScope;
	}

	/**
	 * @return the statements
	 */
	public ArrayList<Statement> getStatements() {
		return this.statements;
	}

	/**
	 * @param statements the statements to set
	 */
	public void setStatements(ArrayList<Statement> statements) {
		this.statements = statements;
	}

	/* (non-Javadoc)
	 * @see
gbang.datatype.GbScopeContainer#getFunction(java.lang.String)
	 */
	public Function getFunction(String name) {
		return this.functionTable.get(name);
	}

}


=======================================================
src/java/gbang/JavaCompiler.java
=======================================================
/* FILE: JavaCompiler.java
 * $Id$
 *
 * The G! Language
 */
package gbang;

import java.io.BufferedReader;
import java.io.DataInputStream;
```

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 *
 */
public class JavaCompiler {

	protected GbTranslator translator;

	public JavaCompiler(GbTranslator translator) {
		this.translator = translator;
	}

	/**
	 * Default compile -- just write java output to console
	 */
	public void compile() throws IOException {
		this.compile(false, false);
	}

	public void compile(boolean toFile) throws IOException {
		this.compile(toFile, false);
	}

	public void compile(boolean toFile, boolean clean) throws
IOException {
		if (toFile == false) {
			System.out.println(translator.toJava());
		} else {
			SimpleDateFormat formatter = new SimpleDateFormat
("hh.mm.ss");
			//System.getenv(name)

//			File javaOut = new File("gbang.tmp." +
formatter.format(new Date()) + ".java");
//			FileWriter writer = new FileWriter(javaOut);
			File javaOut = new File("GBangGame.java");
			FileWriter writer = new FileWriter(javaOut);
			writer.write(translator.toJava());
			writer.close();

			CLI.run("javac -cp lib/java/golden_0_2_3.jar " +
javaOut.getAbsolutePath());

			if (clean == true) {
				javaOut.delete();
			}
		}
	}
}
```

==========

src/java/gbang/SymbolTable.java
```
/* FILE: SymbolTable.java
 * $Id$
 *
 * The G! Language
 */
package gbang;

import java.util.HashMap;

import gbang.datatype.GbDataType;

/**
 * The Symbol table goes here.
 */
public class SymbolTable {

	HashMap<String,GbDataType> symbols;

	/**
	 * Default constructor.
	 */
	public SymbolTable() {
		this.symbols = new HashMap<String,GbDataType>();
	}
} // class SymbolTable
```

src/java/gbang/antlr/expression_walker.g
```
/*
 * semanticwalker.g
 */
header {
	// use this area to specify a package for the resulting
parser/lexer
	// and any imported classes
package gbang.antlr;

import gbang.*;
import gbang.datatype.*;
import gbang.statement.*;
import java.util.Vector;
import java.util.*;
import antlr.CommonAST;

}

class GBangExpressionWalker extends TreeParser;

options {
	importVocab = GBang;
}

{
```

```
	// Uitility stuffs
	protected GbTranslator _translator = new GbTranslator();
	protected GbScopeContainer _currentScope = _translator;
	protected GBangWalker gbwalker=new GBangWalker();
	public static int _passNumber = 1;
	public static boolean DEBUG = true;
	public GbException gbException=new GbException();
	int errline=0;
	boolean lval = true;    //anything by default..in the assign case
in the expression method, set lval to false while evaluating the rval.

	Vector allsyms = new Vector();
	Vector allrels = new Vector();
	Vector alllogs = new Vector();


//    allrels.add("#L");
//    allrels.add("#R");
	/** Function to write stuff out when debug switch is on */
	protected void debug(String msg) {
		if (DEBUG) {
			System.out.println(msg);
		}
	} // method debug

	/**
	 * Start at node and walk through to end
	 */
	public String walkSiblings(AST node) throws RecognitionException {
		if (node == null) {
			return "";
		}
		StringBuffer sb = new StringBuffer();
		while (node != null) {
			if (node.getNumberOfChildren() != 0) {
				sb.append(expression(node));
			} else {
				sb.append(node.getText());
			}
			node = node.getNextSibling();
		}
		return sb.toString();
	}

	public String printNode(AST node) throws RecognitionException {
		String result;

		return null;
	}

	public boolean isFunctionCall(AST node) {
		boolean is = false;
		AST next = node.getNextSibling();
		if (next != null) {
			if (next.getType() == GBangTokenTypes.FUNC_CALL) {
				is = true;
			}
```

```
            }
            return is;
      }

      // check to see if node is a function first
      public boolean hasArguments(AST node) {
            boolean has = false;
            return (node.getNextSibling().getNextSibling().getType() ==
GBangTokenTypes.ARGS);
      }

      public AST getRhs(AST node) {
            AST rhs;
            if (isFunctionCall(node)) {
                  rhs =
node.getNextSibling().getNextSibling().getNextSibling();
                  if (hasArguments(node)) {
                        // buh-jeeezus
                        rhs = rhs.getNextSibling();
                  }
            } else {
                  rhs = node.getNextSibling();
            }

            return rhs;
      }

      public AST getLhs(AST node) {
            //ASTFactory asf = getASTFactory();
            return node;
      }


      public String writeNode(AST node) throws RecognitionException {
            String result = "???";
            if (!isFunctionCall(node)) {
                  result = node.getText();
            } else {
                  result = node.getText() + "(";
                  if (hasArguments(node)) {
                        result +=
func_args(node.getNextSibling().getNextSibling());
                  } else {
                        result += ")";
                  }
            }

            return result;
      }

      boolean inProp = false;
      boolean inLhs = true;

}

// pass the ARGS root
func_args returns [ String args ]
```

```
{
      args = null;
}
      :     #(ARGS argexp:. (more:ARGS)?) {
                  args = expression(argexp);
                  if (more != null) {
                        args += ", " + func_args(more);
                  }
      }
      ;

expression returns [ String result ]
{
      result = null;
      String adtype, bdtype;
      String l,r;
      AST LHS, RHS;
}

      :     #(EXPR child:.) {
                  result = expression(#child);
      }

      |     #(PLUS a1:. b1:.) { // 1
                  LHS = getLhs(a1);
                  RHS = getRhs(a1);
                  result = "(" + expression(LHS) + ") + (" +
expression(RHS) + ")";
      }

    | #(MINUS a2:. (b2:.)?) { // 2
            if (b2 == null) {
                  LHS = getLhs(a2);
                  result = "-" + expression(LHS);
            } else {
                        LHS = getLhs(a2);
                        RHS = getRhs(a2);
                        result = "(" + expression(LHS) + ") - (" +
expression(RHS) + ")";
            }
    }

    | #(STAR a3:. b3:.) { // 3
                  LHS = getLhs(a3);
                  RHS = getRhs(a3);
                  result = "(" + expression(LHS) + ") * (" +
expression(RHS) + ")";
      }

    | #(SLASH a4:. b4:.) { // 4
                  LHS = getLhs(a4);
                  RHS = getRhs(a4);
                  result = "(" + expression(LHS) + ") / (" +
expression(RHS) + ")";
    }

    | #(POWER a5:. b5:.) { // 5
```

```
                LHS = getLhs(a5);
                RHS = getRhs(a5);
                result = "(" + expression(LHS) + ")^(" +
expression(RHS) + ")";
    }

    | #(PERCENT a6:. b6:.) { // 6
                LHS = getLhs(a6);
                RHS = getRhs(a6);
                result = "(" + expression(LHS) + ") % (" +
expression(RHS) + ")";
    }

    | #(ASSIGN a7:. b7:.) { // 7
                LHS = getLhs(a7);
                RHS = getRhs(a7);
                result = expression(LHS) + " = " + expression(RHS);
    }

      |     #(GT a8:. b8:.) { // 8
                LHS = getLhs(a8);
                RHS = getRhs(a8);
                result = "(" + expression(LHS) + ") > (" +
expression(RHS) + ")";
      }

      |     #(LT a9:. b9:.) { // 9
                LHS = getLhs(a9);
                RHS = getRhs(a9);
                result = "(" + expression(LHS) + ") < (" +
expression(RHS) + ")";
      }

      |     #(GEQ a10:. b10:.) { // 10
                LHS = getLhs(a10);
                RHS = getRhs(a10);
                result = "(" + expression(LHS) + ") >= (" +
expression(RHS) + ")";
      }

      |     #(LEQ a11:. b11:.) { // 11
                LHS = getLhs(a11);
                RHS = getRhs(a11);
                result = "(" + expression(LHS) + ") <= (" +
expression(RHS) + ")";
      }

      |     #(EQUAL a12:. b12:.) { // 12
                LHS = getLhs(a12);
                RHS = getRhs(a12);
                result = "(" + expression(LHS) + ") == (" +
expression(RHS) + ")";
      }

      |     #(NEQUAL a13:. b13:.) { // 13
                LHS = getLhs(a13);
                RHS = getRhs(a13);
```

```
                  result = "(" + expression(LHS) + ") != (" +
expression(RHS) + ")";
      }

      |     #("and" a19:. b19:.) { // 14
                  LHS = getLhs(a19);
                  RHS = getRhs(a19);
                  result = "(" + expression(LHS) + ") && (" +
expression(RHS) + ")";
      }

      |     #("or" a20:. b20:.) { // 15
                  LHS = getLhs(a20);
                  RHS = getRhs(a20);
                  result = "(" + expression(LHS) + ") || (" +
expression(RHS) + ")";
      }

      |     #(PROP a21:. b21:.) { // 16
            LHS = getLhs(a21);
            RHS = getRhs(a21);
            result =  expression(LHS) + ".";
            inLhs = false;
            inProp = true;
            result += expression(RHS);
            inLhs = true;
            inProp = false;
      }

      |     #("not" a22:.) {
            result = "!" + "(" + expression(getLhs(a22))  + ")";
      }

      |     id:. { // 17
                  // result = #id.getText();
                  if (inProp) {
                        if (inLhs == false) {
                              // convert to get*
                              if (isFunctionCall(id)) {
                                    result = writeNode(id);
                              } else {
                                    result =
GTGEObject.javaGetter(#id.getText());
                              }
                        }
                  } else {
                        result = writeNode(#id);
                  }

      }

      ;



==========
src/java/gbang/antlr/Main.java
```

```
=========================================================
/*
 * Simple front-end for an ANTLR lexer/parser that dumps the AST
 * textually and displays it graphically.  Good for debugging AST
 * construction rules.
 *
 * Behrooz Badii, Miguel Maldonado, and Stephen A. Edwards
 */
package gbang.antlr;
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

public class Main {
  public static void main(String args[]) {
    try {
      FileInputStream input = new FileInputStream(new
File("c:\\Documents and Settings\\Amortya
Ray\\workspace\\PLT\\test\\test_programs\\test.txt"));

      // Create the lexer and parser and feed them the input
      GBangLexer lexer = new GBangLexer(input);
      GBangParser parser = new GBangParser(lexer);
      parser.program(); // "program" is the main rule in the parser

      // Get the AST from the parser
      CommonAST parseTree = (CommonAST)parser.getAST();

      // Print the AST in a human-readable format
      System.out.println(parseTree.toStringList());

      // Open a window in which the AST is displayed graphically
      ASTFrame frame = new ASTFrame("AST from the Gbang parser",
parseTree);
      frame.setVisible(true);

    } catch(Exception e) { System.err.println("Exception: "+e); }
  }
}


=========================================================
src/java/gbang/antlr/semanticwalker2.g
=========================================================
/*hey
 * semanticwalker.g
 */
header {
      // use this area to specify a package for the resulting
parser/lexer
      // and any imported classes
package gbang.antlr;

import gbang.*;
import gbang.datatype.*;
import gbang.statement.*;
```

```
import java.util.Vector;
import java.util.*;
import antlr.CommonAST;

}

class GBangSemanticWalker2 extends TreeParser;

options {
      importVocab = GBang;
}

{
      // Uitility stuffs
      public GbTranslator _translator = new GbTranslator();
      protected GbScopeContainer _currentScope = _translator;
      protected GBangWalker gbwalker=new GBangWalker();
      public static int _passNumber = 1;
      public static boolean DEBUG = true;
      public GbException gbException=new GbException();
      int errline=0;
      boolean lval = true;    //anything by default..in the assign case
in the expression method, set lval to false while evaluating the rval.

      Vector allsyms = new Vector<String>();
      Vector allrels = new Vector<String>();
      Vector alllogs = new Vector<String>();

      public GbScopeContainer getCurrentScope() { return _currentScope;
}
      public void setCurrentScope(GbScopeContainer scope)
{ _currentScope = scope; }


//    allrels.add("#L");
//    allrels.add("#R");
      /** Function to write stuff out when debug switch is on */
      protected void debug(String msg) {
            if (DEBUG) {
                  System.out.println(msg);
            }
      } // method debug


      protected String getNodeDataType(AST cast) {
      String mytype="Default";
      int temp;
      if(cast.getNextSibling() !=null &&
cast.getNextSibling().getText().equals("(")) {
            mytype = "funky";
      } else if(cast.getType()==ID) {     //Get the type from the ID
//          if( null ==
_translator.getGlobalScope().get(cast.getText())) {
            if( null == _currentScope.getVar(cast.getText())) {
                  gbException.addException("Variable " + cast + "
undefined "); // at line: " + errline);
                  mytype="Undefined";
```

```
                return mytype;
            }
            //mytype =
_translator.getGlobalScope().get(cast.getText()).getType();
            mytype = _currentScope.getVar(cast.getText()).getType();
      } else {    //get the type from the token
            temp = cast.getType();
            switch (temp)  {
                  case (INT_NUM):
                        mytype = "Integer"; break;
                  case (DOUBLE_NUM):
                        mytype = "Double"; break;
                  case (NUMBER):
                        mytype = "Number"; break;
                  case (LITERAL_Boolean):
                        mytype = "Boolean"; break;
                  case (LITERAL_SpriteG):
                  case (LITERAL_SpriteGroup):
                        mytype = "SpriteGroup"; break;
                  case (LITERAL_Sprite):
                        mytype = "Sprite"; break;
                  case (FUNC_CALL):
                        mytype = "Function"; break;
                  case (STRING):
                        mytype = "String"; break;
                  case (LITERAL_PlayField):
                        mytype = "PlayField"; break;
                  case (LITERAL_Coordinate):
                        mytype = "Coordinate"; break;
                  case (LITERAL_void):
                        mytype = "void"; break;
                  case (KEY_PRESS):
                        mytype = "KeyPress"; break;
                  default:
                        mytype = "Default"; break;
            }
      }
      return mytype;

      }



      public String getFuncReturnType(Function f) {
            String type=f.getReturnType().toString();

            if(type.equals("GbInteger"))
                  type = "Integer";
            else if (type.equals("GbString"))
                  type = "String";
            else if (type.equals("GbDouble"))
                  type = "Double";
            else
                  type = type;


            return type;
```

```
	}

}

// semantics should deal w/ boolean expression vs normal ones
//boolean_expression returns [ GbDataType x ]
//{
//    x = expression(#);
//}
//;

expression returns [ String x ] {
	x = new String();
	String adtype, bdtype;


	GbDataType tempname;
//    errline=count;
	if (this.allsyms.size() == 0) {
		// we don't have to initialize this all the time, do we?
		allsyms.add("+");
		allsyms.add("-");
		allsyms.add("*");
		allsyms.add("/");
		allsyms.add("%");
		allsyms.add("^");

		allrels.add(">=");
		allrels.add("<=");
		allrels.add(">");
		allrels.add("<");
		allrels.add("==");
		allrels.add("~=");
		allrels.add("!top");
		allrels.add("!bottom");
		allrels.add("!left");
		allrels.add("!right");
		allrels.add("!");

		alllogs.add(">=");
		alllogs.add("<=");
		alllogs.add(">");
		alllogs.add("<");
		alllogs.add("==");
		alllogs.add("~=");
		alllogs.add("!top");
		alllogs.add("!bottom");
		alllogs.add("!left");
		alllogs.add("!right");
		alllogs.add("!");
		alllogs.add("and");
		alllogs.add("not");
		alllogs.add("or");
	}

}
```

```
/*    :
      (#(PLUS a=expression b=expression { str =  "(" + a +" + "+ b +
")"; })
      | #(MINUS a=expression b=expression { str =  "(" + a +" - "+ b +
")"; })
      | #(STAR a=expression b=expression { str =  "(" + a +" * "+ b +
")"; })
      | #(SLASH a=expression b=expression { str =  "(" + a +" / "+ b +
")"; })
      | #(PERCENT a=expression b=expression { str =  "(" + a +" % "+ b
+ ")"; })
      | #(POWER a=expression b=expression { str =  "(" + a +" ^ "+ b +
")"; })
      | i:ATOM {
            if (!(i instanceof GbNumber))
                  gbException.addException(str + "is not a valid
arithmetic expression");
            }

);*/
 :
   (
   #(PLUS a1:. b1:. {
            debug("a1 " + a1.getText());
            debug("b1 " + b1.getText());
            adtype = getNodeDataType(a1);
            if(adtype.equals("funky")) {
                  function(a1);
                  adtype = "Integer";           //a1 doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
-rachit
                  while(!b1.getText().equals(")")) {
                        b1=b1.getNextSibling();
                  }
                  b1 = b1.getNextSibling();
            }
            bdtype = getNodeDataType(b1);
            if(bdtype.equals("funky")) {
                  function(b1);
                  bdtype="Integer"; //b1 doesn't need to be evaluated
further...just setting it to Integer to avoid redundant code
            }

            debug("a1 is type: " + adtype);
            debug("b1 is type: " + bdtype);
       if((allsyms.contains(a1.getText())) &&
(allsyms.contains(b1.getText()))){
           expression(#a1);
           expression(#b1);
       } else if ((adtype.equals("Integer") ||
adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           debug(a1+"+"+b1);
       } else
if((allsyms.contains(a1.getText()))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
```

```
          // debug("00101");
           expression(#a1);
       } else
if((allsyms.contains(b1.getText()))&&(adtype.equals("Integer") ||
adtype.equals("Double"))) {
           debug("Inside expression");
           expression(#b1);
       }else {
           //gbException.addException("(" + a1 + "+" + b1 + "): Invalid
arithmetic expression "); // at line: " + errline);
           gbException.addException("(" + a1 + " + " + b1 + "): Invalid
arithmetic expression "); // at line: " + errline);
       }

       if(adtype.equals("Double") || bdtype.equals("Double")){
                 x="Double";
         }
         else{
                 x="Integer";
         }

    }
    )
    |
    #(MINUS a2:. b2:. {
      adtype = getNodeDataType(a2);
      bdtype = "blah";  //just another stupid hack to check for unary
minus (negative numbers)
      if(adtype.equals("funky")) {
                 function(a2);
                 adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                 while(!b2.getText().equals(")")) {
                       b2=b2.getNextSibling();
                 }
                 b2 = b2.getNextSibling();
           }
           bdtype = getNodeDataType(b2);
           if(bdtype.equals("funky")) {
                 function(b2);
                 bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
           }
      if(b2!=null && bdtype.equals("blah"))     //if bdtype is
blah...its not a function but a negative number
                 bdtype = getNodeDataType(b2);
           else
           bdtype = "NULL";

       if((allsyms.contains(a2.getText())) &&
(allsyms.contains(b2.getText()))){
           expression(#a2);
           expression(#b2);
       } else if ((adtype.equals("Integer") ||
adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
```

```
            debug(a2+"-"+b2);
        } else if
((allsyms.contains(a2.getText()))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
            expression(#a2);
        } else if
((allsyms.contains(b2.getText()))&&(adtype.equals("Integer") ||
adtype.equals("Double"))) {
            expression(#b2);
        } else if (bdtype == null) {
                    if(adtype.equals("Integer") ||
adtype.equals("Double")) {
                          debug("negative number");
                    } else {
                          gbException.addException("-" + a2.getText() + "
Invalid use of unary operation "); // at line: " + errline);
                    }
        } else {
            gbException.addException("(" + a2+ " - " + b2 + "): Invalid
arithmetic expression "); // at line: " + errline);
        }

        if(adtype.equals("Double") || bdtype.equals("Double")){
                    x="Double";
          }
          else{
                    x="Integer";
          }
     })
     |
     #(STAR a3:. b3:. {
       adtype = getNodeDataType(a3);
       if(adtype.equals("funky")) {
                    function(a3);
                    adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                    while(!b3.getText().equals(")")) {
                          b3=b3.getNextSibling();
                    }
                    b3 = b3.getNextSibling();
             }
             bdtype = getNodeDataType(b3);
             if(bdtype.equals("funky")) {
                    function(b3);
                    bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
             }

          debug("a3 datatype = " + adtype);
          debug("b3 datatype = " + bdtype);
        if((allsyms.contains(a3.getText())) &&
(allsyms.contains(b3.getText()))){
            expression(#a3);
            expression(#b3);
```

```
       } else if ((adtype.equals("Integer") ||
adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           debug(a3+"*"+b3);
       } else if
((allsyms.contains(a3.getText()))&&(b3.getType()==INT_NUM ||
b3.getType() == DOUBLE_NUM)) {
           expression(#a3);
       } else if
((allsyms.contains(b3.getText()))&&(adtype.equals("Integer") ||
adtype.equals("Double"))) {
           expression(#b3);
       }else {
           gbException.addException("(" + a3 + " * " + b3 + "): Invalid
arithmetic expression  "); // at line: " + errline);
       }

       if(adtype.equals("Double") || bdtype.equals("Double")){
                  x="Double";
         }
         else{
                  x="Integer";
         }
    })
    |
    #(SLASH a4:. b4:. {
      adtype = getNodeDataType(a4);
      if(adtype.equals("funky")) {
                  function(a4);
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b4.getText().equals(")")) {
                        b4=b4.getNextSibling();
                  }
                  b4 = b4.getNextSibling();
            }
            bdtype = getNodeDataType(b4);
            if(bdtype.equals("funky")) {
                  function(b4);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

       if((allsyms.contains(a4.getText())) &&
(allsyms.contains(b4.getText()))){
           expression(#a4);
           expression(#b4);
       } else
        if ((adtype.equals("Integer") ||
adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           debug(a4+"/"+b4);
       } else if
((allsyms.contains(a4.getText()))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           expression(#a4);
```

```
       } else if
((allsyms.contains(b4.getText()))&&(adtype.equals("Integer") ||
adtype.equals("Double"))) {
           expression(#b4);
       }else {
           gbException.addException("(" + a4 + " / " + b4 + "): Invalid
arithmetic expression  "); // at line: " + errline);
       }
       if(adtype.equals("Double") || bdtype.equals("Double")){
                  x="Double";
         }
         else{
                  x="Integer";
         }
    })
    |
    #(POWER a5:. b5:. {
      adtype = getNodeDataType(a5);
      if(adtype.equals("funky")) {
                  function(a5);
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b5.getText().equals(")")) {
                        b5=b5.getNextSibling();
                  }
                  b5 = b5.getNextSibling();
            }
            bdtype = getNodeDataType(b5);
            if(bdtype.equals("funky")) {
                  function(b5);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

       if((allsyms.contains(a5.getText())) &&
(allsyms.contains(b5.getText()))){
           expression(#a5);
           expression(#b5);
       } else if ((adtype.equals("Integer") ||
adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           debug(a5+"^"+b5);
       } else if
((allsyms.contains(a5.getText()))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           expression(#a5);
       } else if
((allsyms.contains(b5.getText()))&&(adtype.equals("Integer") ||
adtype.equals("Double"))) {
           expression(#b5);
       }else {
           gbException.addException("(" + a5 + " ^ " + b5 + "): Invalid
arithmetic expression  "); // at line: " + errline);
       }
       if(adtype.equals("Double") || bdtype.equals("Double")){
                  x="Double";
```

```
        }
        else{
                x="Integer";
        }
    })
    |
    #(PERCENT a6:. b6:. {
      adtype = getNodeDataType(a6);
      if(adtype.equals("funky")) {
                function(a6);
                adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                while(!b6.getText().equals(")")) {
                      b6=b6.getNextSibling();
                }
                b6 = b6.getNextSibling();
           }
           bdtype = getNodeDataType(b6);
           if(bdtype.equals("funky")) {
                function(b6);
                bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
           }


       if((allsyms.contains(a6.getText())) &&
(allsyms.contains(b6.getText()))){
           expression(#a6);
           expression(#b6);
       } else if ((adtype.equals("Integer") ||
adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           debug(a6+"%"+b6);
       } else if
((allsyms.contains(a6.getText()))&&(bdtype.equals("Integer") ||
bdtype.equals("Double"))) {
           expression(#a6);
       } else if
((allsyms.contains(b6.getText()))&&(adtype.equals("Integer") ||
adtype.equals("Double"))) {
           expression(#b6);
       }else {
           gbException.addException("(" + a6 + " % " + b6 + "): Invalid
arithmetic expression");
       }
         if(adtype.equals("Double") || bdtype.equals("Double")){
                x="Double";
         }
         else{
                x="Integer";
         }
    })
    |
    #(ASSIGN a7:. b7:. {
      if(a7.getText().equals("->")) {
           adtype = "prop";
```

```
            expression(#a7);
      } else {
            adtype = getNodeDataType(a7);
      }
      if(b7.getText().equals("->")) {
            bdtype = "prop";
            expression(#b7);
      } else {
            bdtype = getNodeDataType(b7);
      }

      if((b7.getNextSibling() != null) && (FUNC_CALL ==
b7.getNextSibling().getType())) {
                  Function func_name =
_currentScope.getFunction(b7.getText());
            debug("b7.getFirstChild()="  + func_name.getReturnType());
                  String temp = getFuncReturnType(func_name);
                  if(adtype.equals(getFuncReturnType(func_name))) {
                        function(b7);
                  } else {
                        gbException.addException("Type Mismatch "); //
at line: " + errline);
                  }
      } else {

                  bdtype = getNodeDataType(b7);
              debug("Equalto");
              if(allsyms.contains(a7.getText()) ||
allrels.contains(a7.getText()) || alllogs.contains(a7.getText())) {
                  gbException.addException("Invalid lvalue operation: "
+ a6 + " / " + b6 + " "); // at line: " + errline);
              }

              if(allsyms.contains(b7.getText())) {
                  debug("RHS datatype = " + bdtype);
                        if((!adtype.equals("prop")) &&
(!(adtype.equals("Integer") || adtype.equals("Double") ||
adtype.equals("Number")))) {
                        gbException.addException ("lvalue does not match
rvalue ");
                  }
                  expression(#b7);
              } else if(allrels.contains(b7.getText())) {


//if(!_translator.getGlobalScope().get(a7.getText()).getType().equals("
Boolean"))  {


if(!_currentScope.getVar(a7.getText()).getType().equals("Boolean"))
      {
                        gbException.addException ("lvalue does not match
rvalue");
                  }
                  expression(#b7);
              }
```

```
      }
    })
        |
       #(GT a8:. b8:. {
            adtype = getNodeDataType(a8);
      if(adtype.equals("funky")) {
                  function(a8);
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b8.getText().equals(")")) {
                        b8=b8.getNextSibling();
                  }
                  b8 = b8.getNextSibling();
            }
            bdtype = getNodeDataType(b8);
            if(bdtype.equals("funky")) {
                  function(b8);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

            if((allrels.contains(a8.getText()))  ||
(allrels.contains(b8.getText()))) {
                  gbException.addException("( " + a8 + " > " + b8 + " )
: Invalid boolean statement "); // at line: " + errline);
            }
            if((allsyms.contains(a8.getText())) &&
(allsyms.contains(b8.getText()))){
                  expression(#a8);
                  expression(#b8);
            } else if ((adtype.equals("Integer") ||
(adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double")))) {
                  debug(a8+">"+b8);
            } else if
((allsyms.contains(a8.getText()))&&((bdtype.equals("Integer") ||
(bdtype.equals("Double"))))) {
                  expression(#a8);
            } else if
((allsyms.contains(b8.getText()))&&((adtype.equals("Integer") ||
(adtype.equals("Double"))))) {
                  expression(#b8);
            }else {
                  gbException.addException("(" + a8 + " > " + b8 + "):
Invalid relational expression");
            }

            x = "Boolean";
       })
        |
       #(LT a9:. b9:. {
            adtype = getNodeDataType(a9);
      if(adtype.equals("funky")) {
                  function(a9);
```

```
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b9.getText().equals(")")) {
                        b9=b9.getNextSibling();
                  }
                  b9 = b9.getNextSibling();
            }
            bdtype = getNodeDataType(b9);
            if(bdtype.equals("funky")) {
                  function(b9);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

            if((allrels.contains(a9.getText()))  ||
(allrels.contains(b9.getText()))) {
                  gbException.addException("( " + a9 + " < " + b9 + " )
: Invalid boolean statement "); // at line: " + errline);
            }
            if((allsyms.contains(a9.getText())) &&
(allsyms.contains(b9.getText()))){
                  expression(#a9);
                  expression(#b9);
            } else if ((adtype.equals("Integer") ||
(adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double")))) {
                  debug(a9+"<"+b9);
            } else if
((allsyms.contains(a9.getText()))&&((bdtype.equals("Integer") ||
(bdtype.equals("Double"))))) {
                  expression(#a9);
            } else if
((allsyms.contains(b9.getText()))&&((adtype.equals("Integer") ||
(adtype.equals("Double"))))) {
                  expression(#b9);
            }else {
                  gbException.addException("(" + a9 + " < " + b9 + "):
Invalid relational expression");
            }
            x = "Boolean";
       })
      |
       #(GEQ a10:. b10:. {
            adtype = getNodeDataType(a10);
      if(adtype.equals("funky")) {
                  function(a10);
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b10.getText().equals(")")) {
                        b10=b10.getNextSibling();
                  }
                  b10 = b10.getNextSibling();
            }
            bdtype = getNodeDataType(b10);
            if(bdtype.equals("funky")) {
```

```
                  function(b10);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

            if((allrels.contains(a10.getText()))  ||
(allrels.contains(b10.getText()))) {
                  gbException.addException("( " + a10 + " == " + b10 +
" ): Invalid boolean statement "); // at line: " + errline);
            }
            if((allsyms.contains(a10.getText())) &&
(allsyms.contains(b10.getText()))){
                  expression(#a10);
                  expression(#b10);
            } else if ((adtype.equals("Integer") ||
(adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double")))) {
                  debug(a10+">="+b10);
            } else if
((allsyms.contains(a10.getText()))&&((bdtype.equals("Integer") ||
(bdtype.equals("Double"))))) {
                  expression(#a10);
            } else if
((allsyms.contains(b10.getText()))&&((adtype.equals("Integer") ||
(adtype.equals("Double"))))) {
                  expression(#b10);
            }else {
                  gbException.addException("(" + a10 + " >= " + b10 +
"): Invalid relational expression");
            }

      x = "Boolean";
       })
       |
       #(LEQ a11:. b11:. {
            adtype = getNodeDataType(a11);
      if(adtype.equals("funky")) {
                  function(a11);
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b11.getText().equals(")")) {
                        b11=b11.getNextSibling();
                  }
                  b11 = b11.getNextSibling();
            }
            bdtype = getNodeDataType(b11);
            if(bdtype.equals("funky")) {
                  function(b11);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

            if((allrels.contains(a11.getText()))  ||
(allrels.contains(b11.getText()))) {
                  gbException.addException("( " + a11 + " == " + b11 +
" ): Invalid boolean statement "); // at line: " + errline);
```

```
            }
            if((allsyms.contains(a11.getText())) &&
(allsyms.contains(b11.getText()))){
                  expression(#a11);
                  expression(#b11);
            } else if ((adtype.equals("Integer") ||
(adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double")))) {
                  debug(a11+"<="+b11);
            } else if
((allsyms.contains(a11.getText()))&&((bdtype.equals("Integer") ||
(bdtype.equals("Double"))))) {
                  expression(#a11);
            } else if
((allsyms.contains(b11.getText()))&&((adtype.equals("Integer") ||
(adtype.equals("Double"))))) {
                  expression(#b11);
            }else {
                  gbException.addException("(" + a11 + " <= " + b11 +
"): Invalid relational expression");
            }
            x = "Boolean";
       })
      |
       #(EQUAL a12:. b12:. {
            adtype = getNodeDataType(a12);
      if(adtype.equals("funky")) {
                  function(a12);
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b12.getText().equals(")")) {
                        b12=b12.getNextSibling();
                  }
                  b12 = b12.getNextSibling();
            }
            bdtype = getNodeDataType(b12);
            if(bdtype.equals("funky")) {
                  function(b12);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

            if((allrels.contains(a12.getText()))  ||
(allrels.contains(b12.getText()))) {
                  gbException.addException("( " + a12 + " == " + b12 +
" ): Invalid boolean statement "); // at line: " + errline);
            }
            if((allsyms.contains(a12.getText())) &&
(allsyms.contains(b12.getText()))){
                  expression(#a12);
                  expression(#b12);
            } else if ((adtype.equals("Integer") ||
(adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double")))) {
                  debug(a12+"=="+b12);
```

```
            } else if
((allsyms.contains(a12.getText()))&&((bdtype.equals("Integer") ||
(bdtype.equals("Double"))))) {
                  expression(#a12);
            } else if
((allsyms.contains(b12.getText()))&&((adtype.equals("Integer") ||
(adtype.equals("Double"))))) {
                  expression(#b12);
            }else {
                  gbException.addException("(" + a12 + " == " + b12 +
"): Invalid relational expression");
            }
            x = "Boolean";
       })
      |
       #(NEQUAL a13:. b13:. {
            adtype = getNodeDataType(a13);
      if(adtype.equals("funky")) {
                  function(a13);
                  adtype = "Integer";           //adtype doesn't need
to be evaluated further...just setting it to Integer to avoid redundant
code
                  while(!b13.getText().equals(")")) {
                        b13=b13.getNextSibling();
                  }
                  b13 = b13.getNextSibling();
            }
            bdtype = getNodeDataType(b13);
            if(bdtype.equals("funky")) {
                  function(b13);
                  bdtype="Integer"; //bdtype doesn't need to be
evaluated further...just setting it to Integer to avoid redundant code
            }

            if((allrels.contains(a13.getText()))  ||
(allrels.contains(b13.getText()))) {
                  gbException.addException("( " + a13 + " ~= " + b13 +
" ): Invalid boolean statement "); // at line: " + errline);
            }
            if((allsyms.contains(a13.getText())) &&
(allsyms.contains(b13.getText()))){
                  expression(#a13);
                  expression(#b13);
            } else if ((adtype.equals("Integer") ||
(adtype.equals("Double"))&&(bdtype.equals("Integer") ||
bdtype.equals("Double")))) {
                  debug(a13+"~="+b13);
            } else if
((allsyms.contains(a13.getText()))&&((bdtype.equals("Integer") ||
(bdtype.equals("Double"))))) {
                  expression(#a13);
            } else if
((allsyms.contains(b13.getText()))&&((adtype.equals("Integer") ||
(adtype.equals("Double"))))) {
                  expression(#b13);
            }else {
```

```
                  gbException.addException("(" + a13 + " ~= " + b13 +
"): Invalid relational expression");
            }
      x = "Boolean";
       })
      |
       #(BANG a14:. b14:. {
            adtype = getNodeDataType(a14);
      if(adtype.equals("funky")) {
                  function(a14);
                  adtype = "Sprite";            //adtype doesn't need
to be evaluated further...just setting it to Sprite to avoid redundant
code
                  while(!b14.getText().equals(")")) {
                        b14=b14.getNextSibling();
                  }
                  b14 = b14.getNextSibling();
            }
            bdtype = getNodeDataType(b14);
            if(bdtype.equals("funky")) {
                  function(b14);
                  bdtype="Sprite";  //bdtype doesn't need to be
evaluated further...just setting it to Sprite to avoid redundant code
            }

            if((allrels.contains(a14.getText()))  ||
(allrels.contains(b14.getText()))) {
                  gbException.addException("( " + a14 + " ! " + b14 + "
): Invalid boolean statement "); // at line: " + errline);
            }
            if ((adtype.equals("Sprite") ||
(adtype.equals("SpriteGroup"))&&(bdtype.equals("Sprite") ||
bdtype.equals("SpriteGroup")))) {
                  debug(a14+"!"+b14);
            } else {
                  gbException.addException("(" + a14 + " ! " + b14 +
"): Invalid bang expression");
            }
      x = "Boolean";
       })
      |
       #(BTOP a15:. b15:. {
            adtype = getNodeDataType(a15);
      if(adtype.equals("funky")) {
                  function(a15);
                  adtype = "Sprite";            //adtype doesn't need
to be evaluated further...just setting it to Sprite to avoid redundant
code
                  while(!b15.getText().equals(")")) {
                        b15=b15.getNextSibling();
                  }
                  b15 = b15.getNextSibling();
            }
            bdtype = getNodeDataType(b15);
            if(bdtype.equals("funky")) {
                  function(b15);
```

```
                  bdtype="Sprite";  //bdtype doesn't need to be
evaluated further...just setting it to Sprite to avoid redundant code
            }

            if((allrels.contains(a15.getText()))  ||
(allrels.contains(b15.getText()))) {
                  gbException.addException("( " + a15 + " !top " + b15
+ " ): Invalid boolean statement "); // at line: " + errline);
            }
             if ((adtype.equals("Sprite") ||
(adtype.equals("SpriteGroup"))&&(bdtype.equals("Sprite") ||
bdtype.equals("SpriteGroup")))) {
                  debug(a15+"!top"+b15);
            } else {
                  gbException.addException("(" + a15 + " !top " + b15 +
"): Invalid bang expression");
            }
      x = "Boolean";
       })
      |
       #(BBOTTOM a16:. b16:. {
            adtype = getNodeDataType(a16);
      if(adtype.equals("funky")) {
                  function(a16);
                  adtype = "Sprite";            //adtype doesn't need
to be evaluated further...just setting it to Sprite to avoid redundant
code
                  while(!b16.getText().equals(")")) {
                        b16=b16.getNextSibling();
                  }
                  b16 = b16.getNextSibling();
            }
            bdtype = getNodeDataType(b16);
            if(bdtype.equals("funky")) {
                  function(b16);
                  bdtype="Sprite";  //bdtype doesn't need to be
evaluated further...just setting it to Sprite to avoid redundant code
            }
            if((allrels.contains(a16.getText()))  ||
(allrels.contains(b16.getText()))) {
                  gbException.addException("( " + a16 + " !bottom " +
b16 + " ): Invalid boolean statement "); // at line: " + errline);
            }
            if ((adtype.equals("Sprite") ||
(adtype.equals("SpriteGroup"))&&(bdtype.equals("Sprite") ||
bdtype.equals("SpriteGroup")))) {
                  debug(a16+"!bottom"+b16);
            } else {
                  gbException.addException("(" + a16 + " !bottom " +
b16 + "): Invalid bang expression");
            }
            x = "Boolean";

       })

      |
       #(BLEFT a17:. b17:. {
```

```
            adtype = getNodeDataType(a17);
      if(adtype.equals("funky")) {
                  function(a17);
                  adtype = "Sprite";            //adtype doesn't need
to be evaluated further...just setting it to Sprite to avoid redundant
code
                  while(!b17.getText().equals(")")) {
                        b17=b17.getNextSibling();
                  }
                  b17 = b17.getNextSibling();
            }
            bdtype = getNodeDataType(b17);
            if(bdtype.equals("funky")) {
                  function(b17);
                  bdtype="Sprite";  //bdtype doesn't need to be
evaluated further...just setting it to Sprite to avoid redundant code
            }
            if((allrels.contains(a17.getText()))  ||
(allrels.contains(b17.getText()))) {
                  gbException.addException("( " + a17 + " !left " + b17
+ " ): Invalid boolean statement "); // at line: " + errline);
            }
            if ((adtype.equals("Sprite") ||
(adtype.equals("SpriteGroup"))&&(bdtype.equals("Sprite") ||
bdtype.equals("SpriteGroup")))) {
                  debug(a17+" !left "+b17);
            } else {
                  gbException.addException("(" + a17 + " !left "  + b17
+ "): Invalid bang expression");
            }
            x = "Boolean";
       })
      |
       #(BRIGHT a18:. b18:. {
            adtype = getNodeDataType(a18);
      if(adtype.equals("funky")) {
                  function(a18);
                  adtype = "Sprite";            //adtype doesn't need
to be evaluated further...just setting it to Sprite to avoid redundant
code
                  while(!b18.getText().equals(")")) {
                        b18=b18.getNextSibling();
                  }
                  b18 = b18.getNextSibling();
            }
            bdtype = getNodeDataType(b18);
            if(bdtype.equals("funky")) {
                  function(b18);
                  bdtype="Sprite";  //bdtype doesn't need to be
evaluated further...just setting it to Sprite to avoid redundant code
            }
            if((allrels.contains(a18.getText()))  ||
(allrels.contains(b18.getText()))) {
                  gbException.addException("( " + a18 + " !right " +
b18 + " ): Invalid boolean statement "); // at line: " + errline);
            }
```

```
            if ((adtype.equals("Sprite") ||
(adtype.equals("SpriteGroup"))&&(bdtype.equals("Sprite") ||
bdtype.equals("SpriteGroup")))) {
                  debug(a18+"!right"+b18);
            } else {
                  gbException.addException("(" + a18 + " !right " + b18
+ "): Invalid bang expression");
            }
            x = "Boolean";
       })
        |
       #("and" a19:. b19:. {
            adtype = getNodeDataType(a19);
      if(adtype.equals("funky")) {
                  function(a19);
                  adtype = "Boolean";           //adtype doesn't need
to be evaluated further...just setting it to Boolean to avoid redundant
code
                  while(!b19.getText().equals(")")) {
                        b19=b19.getNextSibling();
                  }
                  b19 = b19.getNextSibling();
            }
            bdtype = getNodeDataType(b19);
            if(bdtype.equals("funky")) {
                  function(b19);
                  bdtype="Boolean"; //bdtype doesn't need to be
evaluated further...just setting it to Boolean to avoid redundant code
            }
            if(alllogs.contains(a19.getText()) &&
(alllogs.contains(b19.getText()))) {
                  debug(a19+" and "+b19);
                  expression(a19);
                  expression(b19);
            } else if ((adtype.equals("Boolean") &&
(bdtype.equals("Boolean")))) {
                  debug(a19+" and "+b19);
            } else if ((adtype.equals("Boolean")) &&
(alllogs.contains(b19.getText()))) {
                  debug(a19+" and "+b19);
                  expression(b19);
            } else if ((bdtype.equals("Boolean")) &&
(alllogs.contains(a19.getText()))) {
                  debug(a19+" and "+b19);
                  expression(a19);
            } else {
                  gbException.addException("( " + a19 + " and " + b19 +
" ): Invalid logical expression");
            }
            x = "Boolean";
       })
      )

      | #("or" a20:. b20:. {
            adtype = getNodeDataType(a20);
      if(adtype.equals("funky")) {
                  function(a20);
```

```
                  adtype = "Boolean";           //adtype doesn't need
to be evaluated further...just setting it to Boolean to avoid redundant
code
                  while(!b20.getText().equals(")")) {
                        b20=b20.getNextSibling();
                  }
                  b20 = b20.getNextSibling();
            }
            bdtype = getNodeDataType(b20);
            if(bdtype.equals("funky")) {
                  function(b20);
                  bdtype="Boolean"; //bdtype doesn't need to be
evaluated further...just setting it to Boolean to avoid redundant code
            }
            if(alllogs.contains(a20.getText()) &&
(alllogs.contains(b20.getText()))) {
                  debug(a20+" and "+b20);
                  expression(a20);
                  expression(b20);
            } else if ((adtype.equals("Boolean") &&
(bdtype.equals("Boolean")))) {
                  debug(a20+" and "+b20);
            } else if ((adtype.equals("Boolean")) &&
(alllogs.contains(b20.getText()))) {
                  debug(a20+" and "+b20);
                  expression(b19);
            } else if ((bdtype.equals("Boolean")) &&
(alllogs.contains(a20.getText()))) {
                  debug(a20+" and "+b20);
                  expression(a20);
            } else {
                  gbException.addException("( " + a20 + " and " + b20 +
" ): Invalid logical expression");
            }
            x = "Boolean";
       })

       | #("not" a22:. {
            adtype = getNodeDataType(a22);
            if(adtype.equals("funky")) {
                  function(a22);
                  adtype="Boolean"; //adtype doesn't need to be
evaluated further...just setting it to Boolean to avoid redundant code
            }
            if (alllogs.contains(a22.getText())){
                  debug("not "+a22);
                  expression(a22);
            }else if (adtype.equals("Boolean")){
                  debug("not "+a22);
            }
            else{
                  gbException.addException("( not "+a22+" ): Invalid
logical expression");
            }
            x="Boolean";
       }
       )
```

```
      |
       #(PROP a21:. b21:. {
            debug("Walking prop operator");
            adtype = getNodeDataType(a21);
//          bdtype = getNodeDataType(b21);      //DON'T DO THIS! - we
don't need to check the datatype for the RHS of ->
            if(adtype.equals("funky") || (b21.getNextSibling()!=null &&
b21.getNextSibling().getText().equals("("))) {
                  gbException.addException("Function calls not allowed
through \'->\' "); // at line: " + errline);
            }
            if(a21.getText().equals("->")) {
                  expression(a21);
            }
            if(b21.getText().equals("->")) {
                  expression(b21);
            }
            if (allrels.contains(a21.getText()) ||
alllogs.contains(a21.getText()) || allsyms.contains(a21.getText())){
                  gbException.addException("Lvalue cannot be an
expression"); // at line: " + errline);
            }
            if (allrels.contains(b21.getText()) ||
alllogs.contains(b21.getText()) || allsyms.contains(b21.getText())){
                  gbException.addException("Rvalue cannot be an
expression"); // at line: "+errline);
            }
       }
      )
;

//------------------
/*program returns [ GbTranslator trans ]
{
      boolean success;
      trans = _translator;

}
      :     #(PROG something:.
                  {
                        if (_passNumber == 1) {
                              debug(".. first pass of tree ..");
                              // do something special on first pass (?)
                        }
                  }
                  (statement { statement(#something); })*
            )
    ;
*/

identifier returns [ String name ]
{
      name = "UNDEFINED";
}
      :     n:ID  { name = n.getText(); }
```

```
      ;

returntype returns [ GbDataType ret ] {
      ret = null;
}
      :     r:.   {
            if(r.getText().equals("EXPR")) {
                  r = r.getFirstChild();
            }
            if (r.getNextSibling()!=null &&
r.getNextSibling().getType()==FUNC_CALL){
                  function(#r);
                  Function f = _currentScope.getFunction(r.getText());
                  ret = f.getReturnType();
            } else if(allrels.contains(r.getText()) ||
alllogs.contains(r.getText()) || allsyms.contains(r.getText())) {
                  debug("r.getText()= " + r.getText());
                  expression(#r);
                  if(allrels.contains(r.getText())) {
                        ret = new GbInteger();
                  } else {
                        ret = new GbBoolean();
                  }
            } else {
                  ret = GbDataType.getInstance(getNodeDataType(r));
            }
      }
      ;

function {
      ArrayList<GbDataType> realArgs, callArgs;
      }
      : fname:. {
            debug("Walking arguments of function: " + fname.getText());
            //AST argstree=fname.getFirstChild(); //Retrieves LPAREN
            AST argstree=fname.getNextSibling(); //Retrieves LPAREN
            argstree=argstree.getNextSibling(); //Retrieves ARGS (or
RPAREN)
            //Function
function=_translator.getFunctionTable().get(fname.getText());
            Function function =
_currentScope.getFunction(fname.getText());
            if (function == null) {
                  gbException.addException("Function '" +
fname.getText() + "' undefined");
                  return;
            }
            realArgs=function.getArgs();
            callArgs=new ArrayList<GbDataType>();
            AST argstree1;
            GbDataType dtype;
            if (argstree.getType() == GBangTokenTypes.ARGS) {
                  do{//Iterates over each argument
                        argstree1=argstree;
                        if(argstree1.getText().equals("ARGS") &&
argstree1.getFirstChild() != null) {
                              argstree1=argstree1.getFirstChild();
```

```
                        }
                        do{//Iterates into each argument
                              debug(argstree.getText());
                              if (argstree.getText().equals("EXPR")){
                                    if
(allsyms.contains(argstree.getFirstChild().getText())){
                                          dtype =
GbDataType.getInstance("Integer");
                                          dtype.setDataType("Integer");
                                          callArgs.add(dtype);

      expression(argstree.getFirstChild());
                                    }
                                    else if
(alllogs.contains(argstree.getFirstChild().getText())){
                                          dtype =
GbDataType.getInstance("Boolean");
                                          dtype.setDataType("Boolean");
                                          callArgs.add(dtype);

      expression(argstree.getFirstChild());
                                    }
                                    else if
(allrels.contains(argstree.getFirstChild().getText())){
                                          dtype =
GbDataType.getInstance("Boolean");
                                          dtype.setDataType("Boolean");
                                          callArgs.add(dtype);

      expression(argstree.getFirstChild());
                                    }
                                    //check if its a function
                                    else
if(argstree.getFirstChild().getFirstChild()!=null &&
argstree.getFirstChild().getFirstChild().getText().equals("(")) {

      function(argstree.getFirstChild());
                                                dtype =
_currentScope.getFunction(fname.getText()).getReturnType();


dtype.setDataType(_translator.getFunctionTable().get(fname.getText()).g
etReturnType().toString());
                                                callArgs.add(dtype);
                                    }
                                    //check if its an identifier
                                     else  {
                                          //if( null ==
_translator.getGlobalScope().get(argstree.getFirstChild().getText())) {
                                          //
      gbException.addException("Variable " +
argstree.getFirstChild().getText() + " undefined "); // at line: " +
errline);
                                          //} else {
                                                dtype =
GbDataType.getInstance(getNodeDataType(argstree.getFirstChild()));
```

```
      dtype.setDataType(getNodeDataType(argstree.getFirstChild()));
                                                callArgs.add(dtype);
                                          //}

                                    }
                              }
                        } while
((argstree=argstree.getFirstChild())!=null);
                  } while((argstree=argstree1.getNextSibling())!=null);
            }

            if (realArgs.size() != callArgs.size()) {
                  gbException.addException("Number of arguments don't
match number in function definition");
            } else {
                  debug(realArgs.toString());
                  debug(callArgs.toString());
                  if (!realArgs.toString().equals(callArgs.toString())){
                        gbException.addException("Mismatched
arguments"); // at line: " + errline);
                  }
            }


      }
;

statement [ int count ]
{
      //String name;
      GbDataType tempname;
      errline=count;
if (this.allsyms.size() == 0) {
            // no need to keep redlacring this, makes debugging painful
      allsyms.add("+");
      allsyms.add("-");
      allsyms.add("*");
      allsyms.add("/");
      allsyms.add("%");
      allsyms.add("^");

      allrels.add(">=");
      allrels.add("<=");
      allrels.add(">");
      allrels.add("<");
      allrels.add("==");
      allrels.add("~=");
      allrels.add("!top");
      allrels.add("!bottom");
      allrels.add("!left");
      allrels.add("!right");
      allrels.add("!");

      alllogs.add(">=");
      alllogs.add("<=");
      alllogs.add(">");
```

```
      alllogs.add("<");
      alllogs.add("==");
      alllogs.add("~=");
      alllogs.add("!top");
      alllogs.add("!bottom");
      alllogs.add("!left");
      alllogs.add("!right");
      alllogs.add("!");
      alllogs.add("and");
      alllogs.add("not");
      alllogs.add("or");
}

}
// variable declarations are handled in first pass

      :     #(VARIABLE_DEF {  //this is taken care of in the first
pass...
            /*AST typeTree = a.getFirstChild();
            GbDataType dtype =
GbDataType.getInstance(typeTree.getText());
            tempname = _currentScope.getCurrentVar(name.getText());
            if(name.getType() == ID) {
                  if(tempname == null) {
                        //_currentScope.addVar(name.getText(), dtype);
                        debug(dtype.getType() + " " + name + " added to
symbol table "); // at line: " + errline);
                  } else {
                        gbException.addException("Duplicate definition
of " + name + " found "); // at line: " + errline);

                  }
            } else {
                  gbException.addException("Invalid identifier " + name
+ " found in variable definition "); // at line: " + errline);
            }*/
      })

      | #(EXPR express:. {
            debug("Inside expression");
            if (express.getNextSibling()!=null &&
express.getNextSibling().getType()==FUNC_CALL){
                  debug("FUNC:" + express.getText());
                  function(#express);
            } else if(express.getText().equals("=") ||
allrels.contains(express.getText()) ||
alllogs.contains(express.getText()) ||
allsyms.contains(express.getText())) {
                        debug("express.getText()= " +
express.getText());
                        expression(#express);
            } else {
                        gbException.addException("Invalid expression "
+ express.getText() + " "); // at line: " + errline);
            }
```

```
      }
      )
      |     #(IF ifexpr:. {
            debug("Walking an If-loop!");
            debug(ifexpr.getText());
            AST temp = ifexpr; //EXPR
            AST temp1;
            if(temp.getText().equals("EXPR")) {
                  temp1=temp.getFirstChild();

                  //Type checking for If condition
                  debug(temp1.getText());
                  if(temp1.getText().equals("true") ||
temp1.getText().equals("false")){
                        debug("Welcome to New York, bitch!");
                  }else if (!getNodeDataType(temp1).equals("Boolean")){
                        if(temp1.getFirstChild() == null) {
                              gbException.addException(temp1 + "
Invalid boolean expression in if statement "); // at line: " + errline);
                        } else {
                              expression(temp1);
                        }
                  }
            } else if (!getNodeDataType(temp).equals("KeyPress")) {
                  gbException.addException(temp + " Invalid boolean
expression in if statement "); // at line: " + errline);
            }

            //If body
            temp=temp.getNextSibling();//IF_BODY
            temp1=temp.getFirstChild();//CHild of IF BODY
            if(temp1!=null){
                  statement(temp1, count);
            }
            //Else Body
            temp=temp.getNextSibling();
            temp1=temp.getFirstChild();//CHild of ELSE BODY
            if(temp1!=null){
                  statement(temp1, count);
            }

      }
      )
      |     #("while" whileexpr:. {
            debug("Walking a while-loop");
            AST temp  = whileexpr;
            if(whileexpr.getText().equals("EXPR")) {
                  temp=whileexpr.getFirstChild();
                  if(temp.getText().equals("true") ||
temp.getText().equals("false")){
                        debug("WELLEED");
                  }
                  if (!getNodeDataType(temp).equals("Boolean")){
                        if(temp.getFirstChild() == null) {
                              gbException.addException(temp + " Invalid
boolean expression in while loop "); // at line: " + errline);
                        } else {
```

```
                              expression(temp);
                        }
                  }
            } else if(!getNodeDataType(temp).equals("KeyPress")) {
                  gbException.addException(temp + " Invalid boolean
expression in while loop "); // at line: " + errline);
            }
      }
      )
      |     #("when" whenexpr:.{
            debug("Walking a when-loop");
            debug(whenexpr.getText());
            AST temp = whenexpr;//.getFirstChild();
            if(temp.getText().equals("EXPR")) {
                  temp = temp.getFirstChild();
                  if (!getNodeDataType(temp).equals("Boolean")){
                        if(temp.getFirstChild() == null) {
                              gbException.addException(temp + " Invalid
boolean expression in when statement "); // at line: " + errline);
                        } else {
                              expression(temp);
                        }
                  }
            } else if(!getNodeDataType(temp).equals("KeyPress")) {
                  gbException.addException(temp + " Invalid boolean
expression in when statement "); // at line: " + errline);
            }
            while((temp=temp.getNextSibling())!=null) {
                  statement(temp, errline);
            }
      }
      )
      |     #(FOR forexpr:.){
            debug("Walking a for-loop");
            AST temp = forexpr;           //initial value (variable)
            //Type checking inital condition
            if (!getNodeDataType(temp).equals("Integer")){
                  gbException.addException(temp + " Invalid integer
expression "); // at line: " + errline);
            }

            temp = forexpr.getNextSibling();          //FOR_RANGE


            //Type checking initial value
            temp = temp.getFirstChild();              //EXPR
            AST temp1=temp.getFirstChild();           //initial value
            if (!getNodeDataType(temp1).equals("Integer")){
                  if(temp1.getFirstChild() == null) {
                        gbException.addException(temp + " Invalid
integer expression "); // at line: " + errline);
                  } else {
                        expression(temp1);
                  }
            }

            //Type checking final condition
```

```
            temp=temp.getNextSibling();   //EXPR
            temp1=temp.getFirstChild();   //final condition
            if (!getNodeDataType(temp1).equals("Integer")){
                  if(temp1.getFirstChild() == null) {
                        gbException.addException(temp + " Invalid
integer expression "); // at line: " + errline);
                  } else {
                        expression(temp1);
                  }
            }

            //Type checking increment/ decrement
            if(temp.getNextSibling() != null) {
                  if(temp.getNextSibling().getText().equals("EXPR")) {

                        temp = temp.getNextSibling(); //EXPR
                        temp1=temp.getFirstChild();         //increment
value
                              if
(!getNodeDataType(temp1).equals("Integer")){
                                    if(temp1.getFirstChild() == null ||
allrels.contains(temp1.getText()) || alllogs.contains(temp1.getText()))
{

      gbException.addException(temp1 + " Integer required "); // at
line: " + errline);
                                    } else {
/*                                        if(allrels.contains(temp1)) {

      System.out.println("ting tong");
                                        }
                                        debug("here");
      */
                                        expression(temp1);
                                    }
                              }
                  } else {
                        gbException.addException(temp + " Invalid
expression in for loop "); // at line: " + errline);
                  }
            }
      }
      |     #(FUNC_DEF ret_type:.{
            debug("Walking a function");
            AST fun_name = ret_type.getNextSibling();
            //Function func_name =
_translator.getFunctionTable().get(fun_name.getText());
            Function func_name =
_currentScope.getFunction(fun_name.getText());
            //if(func_name != null) {
            //    gbException.addException("A function with name " +
func_name + " already exists. Error "); // at line: " + errline);
            //} else{
                  AST temp = fun_name.getNextSibling();
                  Function func = new Function(ret_type, fun_name,
temp, temp.getNextSibling(), gbwalker);
                  _translator.addFunction(func);
```

```
                  //temp = temp.getNextSibling();
                  //debug("Hola " + temp.getText());
                  while((temp=temp.getNextSibling())!=null) {
                        if(temp.getText().equals("return")) {


if(!func_name.getReturnType().toString().equals(returntype(temp.getFirs
tChild()).toString())){
                                    gbException.addException("return
value and return type of function " + fun_name.getText() + " do not
match "); // at line: " + errline);
                              }
                              break;
                        }
                        statement(temp, count);
                  }
            //}
      }
      )
      | #("break" {
      })
      | #("continue" {
      })
      ;
```

```
============================================================
src/java/gbang/antlr/walker.g
============================================================
/*
 * walker.g
 */
 header {
      // use this area to specify a package for the resulting
parser/lexer
      // and any imported classes
package gbang.antlr;

import java.util.*;

import gbang.*;
import gbang.datatype.*;
import gbang.statement.*;


}

class GBangWalker extends TreeParser;

options {
      importVocab = GBang;

}
```

```
{
      // what the prop does:
      public static final int PROP_FUNC_CALL = 0;
      public static final int PROP_ASSIGNMENT = 1;
      public static final int PROP_ACCESSOR = 2;
      public static final int PROP_COORDS = 3;
      public static final int PROP_PROPERTIES = 4;

      // Uitility stuffs
      public static GbTranslator _translator = new GbTranslator();
      public static GbFunctionTable _functionTable =
_translator.getFunctionTable();

      protected GbScopeContainer _currentScope = _translator;

      public GbException gbException=new GbException();

      public static int _passNumber = 1;
      public static boolean DEBUG = true;

      /** Function to write stuff out when debug switch is on */
      protected void debug(String msg) {
            if (DEBUG) {
                  System.out.println(msg);
            }
      } // method debug

      public GbScopeContainer getCurrentScope() { return _currentScope;
}
      public void setCurrentScope(GbScopeContainer scope)
{ _currentScope = scope; }

      public GbTranslator getTranslator() { return this._translator; }
}

program returns [ GbTranslator trans ]
{
      boolean success;
      trans = _translator;

}
      :     #(PROG something:.
                  {
                        if (_passNumber == 1) {
                              debug("... first pass of tree ...");
                              // do something special on first pass (?)
                        }
                  }
                  (statement { statement(#something); })*
            )
    ;


identifier returns [ String name ]
{
      name = "UNDEFINED";
}
```

```
      :     n:ID  { name = n.getText(); }
      ;



statement
{
      String name;
}
      :     #(VARIABLE_DEF a:TYPE name=identifier (assignmnt:.)?) {
            // check to see that variable isn't declared already
            if (_currentScope.getCurrentVar(name) != null) {
                  gbException.addException("Variable " + name + " is
already declared "
                        + "on line " + a.getLine());
            }
            AST typeTree = a.getFirstChild();
            debug(typeTree.getText());
            GbDataType dtype =
GbDataType.getInstance(typeTree.getText(), name);
            dtype.setVarName(name);
            _currentScope.addVar(name, dtype);
            _currentScope.addStatement(new Declaration(dtype, name));

            // dealing with inline assignment with declarations
            if (assignmnt != null) {
                  //Expression assn =
Expression.getInstance(a.getNextSibling());
                  Assignment assign =
Assignment.fromDeclaration(a.getNextSibling(), assignmnt,
_currentScope);
                  //assign.setScope(_currentScope);
                  _currentScope.addStatement(assign);
            }
      }

      |     #(FUNC_DEF ftype:. fname:. fargs:. body:.) {
            // notice that we are missing the "body" argument
            // up top -- ANTLR only needs to match the first few
            Function func = new Function(ftype, fname, fargs, body,
this);
            func.setTree(#FUNC_DEF);
            Function func_name =
_currentScope.getFunction(fname.getText());
            if(func_name != null) {
                  gbException.addException("A function with name " +
func_name + " already exists.");
            } else{
                  _translator.addFunction(func);
            }
      }

      |     #(EXPR expr:.) {
            //
            // FIX HERE FOR NEW PROP STYLE GRAMMAR
            //
            // check to see if this is a properties call.
```

```
            if (expr.getType() == GBangTokenTypes.PROP) {
                  // -> is root, has 2 children (left: x, right: y, in
x->y)
                  AST left = expr.getFirstChild();
                  AST right = left.getNextSibling();
                  GTGEObject thing=null;

                  try {
                        Object it =
_currentScope.getVar(left.getText());
                        if (it == null) {
                              debug("playing with an object that isn't
declared!");
                              gbException.addException("Undeclared
variable " + left.getText());
                        } else {
                              thing = (GTGEObject) it;
                        }
                  } catch (ClassCastException e) {
                        gbException.addException("Can't use -> access
on a non GTGE object");
                  }

                  switch (prop_role(expr)) {
                        case PROP_FUNC_CALL:
                              // x->something();
                              debug("prop function call x->do()");
                              break;

                        case PROP_ASSIGNMENT:
                              // x->y = something;
                              // handle like a normal assignent
                              debug("prop x->y = 10");
                              Expression expression =
Expression.getInstance(expr,this);
                  //          PropAssignment lhs = new
PropAssignmnet(expr);
                              _currentScope.addExpression(expression);
                              break;

                        case PROP_ACCESSOR:
                              // something = x->y;
                              debug("prop x->y accessor");
                              break;

                        case PROP_COORDS:
                              // playfield->someSprite@right
                              debug("prop coord call");
                              break;

                         case PROP_PROPERTIES:
                              // sprite->properties( ... )
                              debug("prop properties: sprite-
>properties(...)");
                              AST kwargs =
right.getNextSibling().getNextSibling();
```

```
                              if (kwargs.getType() ==
GBangTokenTypes.RPAREN) {
                                    debug("No properties to set,
skipping");
                              } else {
                                    thing.setProperties(kwargs);
                              }
                              break;

                        default:
                              gbException.addException("Can not
disambiguate PROP (->) function");
                        }
            } else {
                  Expression expression = Expression.getInstance(expr,
this);
                  _currentScope.addExpression(expression);
            }
      }

      |     #(IF iftest:. ifbody:IFBODY ifel:ELSE ) {
            // if this isn't awesome, I don't know what is.
            // each node here will have no children if there's no
"meat" in there
            // e.g. testing for null ifbody's was a pain until now
            If iff = new If(iftest, ifbody, ifel, this);
            iff.setTree(#IF);
            _currentScope.addStatement(iff);
      }

      |     #(w:"when" whentest:.) {
            // deal with empty when block
            AST dowhen = whentest.getNextSibling();
            if (dowhen != null) {
                  When when = new When(whentest, dowhen, this);
                  when.setTree(#w);
                  _currentScope.addStatement(when);
            }
      }

      |     #(wh:"while" whiletest:.) {
            // deal with empty when block
            AST dowhile = whiletest.getNextSibling();
            if (dowhile != null) {
                  While dwhile = new While(whiletest, dowhile, this);
                  dwhile.setTree(#wh);
                  _currentScope.addStatement(dwhile);
            }
      }

      |     #("return" (ret:EXPR)?) {
            // TODO: may need to handle "return;" (with no return value)
            Return retVal = new Return(ret, this);
            _currentScope.addStatement(retVal);
      }

      |     #(FOR for_first:.) {
```

```
            try {
                  For f = new For(for_first,this);
                  f.setTree(#FOR);
                  _currentScope.addStatement(f);
            } catch (GbEmptyBodyException e) {
                  // don't use an empty body in a for loop
                  gbException.addException("Use of empty \'for\' loop
not allowed");
            }
      }
      ;

prop_role returns [ int role ]
{
      role = -1;
}
      :     #(PROP left:. right:.) {
            if (right.getType() == GBangTokenTypes.PROP) {
                  role = prop_role(right);
            } else {
                  if (right.getNextSibling() == null) {
                        role = PROP_ACCESSOR;
                  } else      if (right.getType() ==
GBangTokenTypes.AT_OP) {
                        role = PROP_COORDS;
                  } else if (right.getText().equals("properties")) {
                        role = PROP_PROPERTIES;
                  } else if (right.getNextSibling().getType() ==
GBangTokenTypes.FUNC_CALL) {
                        role = PROP_FUNC_CALL;
                  } else if (right.getNextSibling().getType() ==
GBangTokenTypes.ASSIGN) {
                        role = PROP_ASSIGNMENT;
                  }
            }
      }
      ;

=====================================================
src/java/gbang/datatype/GbArgumentTable.java
=====================================================
/* FILE: GbArgumentTable.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.util.HashMap;

import gbang.statement.Expression;

/**
 * Should we use this to deal with arguments that are
 * passed/defined in function calls (?)
 */
public class GbArgumentTable {
```

```
	HashMap<String,Expression> args;
	HashMap<String,Expression> kwargs;

	public GbArgumentTable() {
		this.args = new HashMap<String,Expression>();
		this.kwargs = new HashMap<String,Expression>();
	}

} // class GbArgumentTable
```

```
=======================================================
src/java/gbang/datatype/GbArray.java
=======================================================
/* FILE: MxArray.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.util.ArrayList;

import gbang.statement.Expression;

/**
 *
 */
public class GbArray extends GbDataType {

	protected ArrayList<GbDataType> vals;
	protected Expression length;

	public GbArray() {
		this.vals = new ArrayList<GbDataType>();
	}

	public GbArray(GbDataType[] vals) {
		this.vals = new ArrayList<GbDataType>(vals.length);
		for (GbDataType val : vals) {
			this.vals.add(val);
		}
	}

	public GbArray(Expression exp) {
		this.vals = new ArrayList<GbDataType>();
		this.length = exp;
	}

	public GbArray(ArrayList<GbDataType> vals) {
		this.vals = vals;
	}
}
```

```
=======================================================
```

src/java/gbang/datatype/GbBoolean.java
```
/* FILE: GbBoolean.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 * Wrapper class for boolean data types
 * @author Amortya Ray
 * @version 1.0
 */

public class GbBoolean extends GbDataType {

	protected boolean val;

	public GbBoolean(boolean b) {
		this.val = b;
	}

	public GbBoolean() {
		this(false);
	}

	public GbBoolean copy(){
		return new GbBoolean(val);
	}

	public String getType(){
		return "Boolean";
	}


	@Override
	public String javaType() {
		return "Boolean";
	}

}
```

src/java/gbang/datatype/GbCoordinate.java
```
/* FILE: GbCoordinate.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.awt.Point;

/**
```

```
 * Wrapper class for Coordinate datatype
 * @author Amortya Ray
 * @version 1.0
 */

public class GbCoordinate extends GbDataType {
      Point p;

      //Default constructor- pretty much useless
      public GbCoordinate(){
      }

      //Constructor to initialize the x and y coordinates of the point
      public GbCoordinate (int x, int y){
            this.p=new Point(x,y);
      }

      //Method to return the x-coordinate of the point
      public int getX(){
            return p.x;
      }

      //Method to return the y-coordinate of the point
      public int getY(){
            return p.y;
      }

      //Copy method
      public GbDataType copy(){
            return new GbCoordinate(p.x, p.y);
      }

      //Method to return the type of the object
      public String getType(){
            return "Coordinate";
      }

}


=====================================================
src/java/gbang/datatype/GbDataType.java
=====================================================
package gbang.datatype;

import antlr.collections.AST;

/**
 * All data types should extend this class
 * This class also generates the error messages
 * Prefix all datatype subclasses w/ Gb* to avoid collision with Java
types:
 *          Like: GbInteger vs. Integer
 *
 * @author Amortya Ray
 * @version 1.0
 */
```

```
public class GbDataType {

	/** Holds onto the name of the datatyp as a String */
	String name;

	/**
	 * This is the name of the variable that is this type
	 */
	protected String varName;

	public static GbDataType getInstance(String type) {
		return getInstance(type, null);
	}

	public static GbDataType getInstance(String type, String name) {
		GbDataType ret = null;

		if ("Integer".equals(type)) {
			ret = new GbInteger();
		} else if ("PlayField".equals(type)) {
			ret = new GbPlayField();
		} else if ("Sprite".equals(type)) {
			ret = new GbSprite();
			GbSprite spr = new GbSprite();
			if (name != null) {
				GbSpriteGroup sister = new GbSpriteGroup();
				sister.setVarName(name.toUpperCase() +
"_GROUP");
				spr.setSister(sister);
			}
			ret = spr;
		} else if ("SpriteGroup".equals(type)) {
			ret = new GbSpriteGroup();
		} else if ("Double".equals(type)) {
			ret = new GbDouble();
		} else if ("Sound".equals(type)) {
			ret = new GbSound();
		} else if ("String".equals(type)) {
			ret = new GbString();
		} else if ("Boolean".equals(type)) {
			ret = new GbBoolean();
		} else if ("Coordinate".equals(type)) {
			ret = new GbCoordinate();
		} else if ("[".equals(type)) {
			// TODO: Get size of array!
			ret = new GbArray();
		} else if ("void".equals(type)) {
			ret = new GbVoid();
		} else {
			throw new GbException("Undefined Type: " + type);
		}

		ret.name = type;
		return ret;
	}
```

```
	public static GbDataType getInstance(AST typeNode, String name) {
		String type = typeNode.getText();

		GbDataType ret = null;

		if ("Integer".equals(type)) {
			ret = new GbInteger();
		} else if ("PlayField".equals(type)) {
			ret = new GbPlayField();
		} else if ("Sprite".equals(type)) {
			GbSprite spr = new GbSprite();
			GbSpriteGroup sister = new GbSpriteGroup();
			sister.setVarName(name.toUpperCase() + "_GROUP");
			spr.setSister(sister);
			ret = spr;
		} else if ("SpriteGroup".equals(type)) {
			ret = new GbSpriteGroup();
		} else if ("Double".equals(type)) {
			ret = new GbDouble();
		} else if ("Sound".equals(type)) {
			ret = new GbSound();
		} else if ("String".equals(type)) {
			ret = new GbString();
		} else if ("Boolean".equals(type)) {
			ret = new GbBoolean();
		} else if ("Coordinate".equals(type)) {
			ret = new GbCoordinate();
		} else if ("[".equals(type)) {
			String arrType = typeNode.getNextSibling().getText();
		} else if ("void".equals(type)) {
			ret = null;
		} else {
			throw new GbException("Undefined Type: " + type);
		}

		ret.name = name;
		return ret;
	}


	public GbDataType(){
		name = null;
	}



    // ____________________________________________________ TRANSLATING TO
JAVA METHODS

	public String javaDeclare() {
		return this.javaType() + " " + this.varName + ";\n";
	}

	/**
	 * Returns the string that should be written when the object
	 *
```

```
       * is finally constructed (and initialized)
       * @return
       */
      public String javaInstantiate() {
            StringBuffer sb = new StringBuffer();
            sb.append(this.varName + " = new " + this.javaType() +
"();\n");
            return sb.toString();
      }

    public String javaType() {
      return "Undefined Type";
    }

    //
_____________________________________________________________________________
___


      public void setDataType(String type) {
            name = type;
      }

    public GbDataType error( String msg ) {
        throw new GbException( "Illegal operation: " + msg
                               + "( <" + getType() + "> "
                               + ( name != null ? name : "<?>" )
                               + " )" );
    }

    public GbDataType error( GbDataType b, String msg ) {
        if ( null == b )
            return error( msg );
        throw new GbException(
            "illegal operation: " + msg
            + "( <" + getType() + "> "
            + ( name != null ? name : "<?>" )
            + " and "
            + "<" + getType() + "> "
            + ( name != null ? name : "<?>" )
            + " )" );
    }

    public String getType(){
      return "Unknown";
    }

    public String toString() {
      return name;
    }

      /**
       * @return the name
       */
      public String getName() {
            return this.name;
      }
```

```
	/**
	 * @param name the name to set
	 */
	public void setName(String name) {
		this.name = name;
	}

	/**
	 * @return the varName
	 */
	public String getVarName() {
		return this.varName;
	}

	/**
	 * @param varName the varName to set
	 */
	public void setVarName(String varName) {
		this.varName = varName;
	}

} // GbDataType


=========================================================
src/java/gbang/datatype/GbDouble.java
=========================================================
/* FILE: GbDouble.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 * Wrapper class for all GBang variables of type double
 * @author Amortya Ray
 * @version 1.0
 */
public class GbDouble extends GbNumber {
	protected double val;

	//Constructor for GbDouble class
	public GbDouble(double d) {
		this.val = d;
	}

	public GbDouble() {
		this(0.0);
	}

	//Method that returns the datatype of the object
	public String getType(){
		return "Double";
	}
```

```
	//Method that returns a new instance of the Double object
	//Inspired by MX
	public GbNumber copy(){
		return new GbDouble(val);
	}

	//Method to return the value of the Integer as a string
	public String getValAsString(){
		return Double.toString(val);
	}


	@Override
	public String javaType() {
		return "Double";
	}


} // class GbDouble


=======================================================
src/java/gbang/datatype/GbEmptyBodyException.java
=======================================================
/* FILE: GbEmptyBodyException.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 *
 */
public class GbEmptyBodyException extends GbException {

	/**
	 *
	 */
	private static final long serialVersionUID = 1817554122077132558L;

	public GbEmptyBodyException(String msg) {
		super(msg);
	}

}


=======================================================
src/java/gbang/datatype/GbException.java
=======================================================
package gbang.datatype;

public class GbException extends RuntimeException {

	/**
	 * @author Amortya Ray and Rachit Parikh
```

```
       * @version 1.0
       */

      private static final long serialVersionUID = 6321754718344178264L;
      public StringBuffer errbuff=new StringBuffer();
      public boolean isSuccess=true;
      public int nErrors = 0;

      public GbException(){}

      public GbException(String msg) {
            System.err.println("Error: "+msg);
            isSuccess=false;
    }

      public void printException(String msg) {
            System.err.println( "Error: " + msg);
      }

      public void addException(String message) {
            errbuff.append("Error: " + message+"\n");
            this.nErrors++;
            isSuccess=false;
      }

      public String getException()  {
            if(errbuff.length()==0) {
                  isSuccess=true;
                  return "Success!";
            } else {
                  return errbuff.toString();
            }
      }

      public int nErrors() { return this.nErrors; }
}


=====================================================
src/java/gbang/datatype/GbExpression.java
=====================================================
/* FILE: GbExpression.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

public class GbExpression {

      //this constructor required in class GbIdentifier
      public GbExpression(String id, GbDataType type) {

      }
}
```

```
=========================================================
src/java/gbang/datatype/GbIdentifier.java
=========================================================
/* FILE: GbIdentifier.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

public class GbIdentifier extends GbExpression {
      public int offset;

      public GbIdentifier(String id, GbDataType type, int i) {
            super(id, type);
            offset = i;
      }
}


=========================================================
src/java/gbang/datatype/GbIllegalNestingException.java
=========================================================
/* FILE: GbIllegalNestingException.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 *
 */
public class GbIllegalNestingException extends GbException {

      /**
       *
       */
      private static final long serialVersionUID =
-2258144049150742513L;

      public GbIllegalNestingException(String msg) {
            super(msg);
      }
}


=========================================================
src/java/gbang/datatype/GbInteger.java
=========================================================
/* FILE: GbInteger.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;
```

```
/**
 * Wrapper class for all Integer data types in GBang
 * @author Amortya Ray
 * @version 1.0
 */

public class GbInteger extends GbNumber {
      int val;

      //Constructor for GbInteger class
      //Initializes an object of type Integer
      public GbInteger(int i) {
            this.val = i;
      }

      public GbInteger() {
            this(0);
      }

      //Method to return the type of the object
      public String getType(){
            return "Integer";
      }

      //Method to return a copy of the Integer value of the object
      //I'm not sure if we can really use it in G!
      //I included it from the MX language
      public GbNumber copy(){
            return new GbInteger(val);
      }

      //Method to return the value of the Integer as a string
      public String getValAsString(){
            return Integer.toString(val);
      }


      @Override
      public String javaType() {
            return "Integer";
      }

} // GbInteger


=======================================================
src/java/gbang/datatype/GbMismatchPropertyTypeException.java
=======================================================
/* FILE: GbMismatchPropertyTypeException.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 *
```

```
 */
public class GbMismatchPropertyTypeException extends GbException {

	/**
	 *
	 */
	private static final long serialVersionUID =
-6164095948901416402L;

	public GbMismatchPropertyTypeException(String msg) {
		super(msg);
	}
}


=======================================================
src/java/gbang/datatype/GbNoSuchPropertyException.java
=======================================================
/* FILE: GbNoSuchPropertyException.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 *
 */
public class GbNoSuchPropertyException extends GbException {

	/**
	 *
	 */
	private static final long serialVersionUID = 1130705723460030030L;

	public GbNoSuchPropertyException(String msg) {
		super(msg);
	}
}


=======================================================
src/java/gbang/datatype/GbNumber.java
=======================================================
/* FILE: GbNumber.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 * Parent class for Float and Integer numbers.
 */
public abstract class GbNumber extends GbDataType {

}
```

```
=====================================================
src/java/gbang/datatype/GbPlayField.java
=====================================================
/* FILE: GbPlayField.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.util.HashMap;

import gbang.statement.Expression;

/**
 *
 */
public class GbPlayField extends GTGEObject {

	public static HashMap<String,GbDataType> ATTRIBS;

	@Override
	public String javaType() {
		return "PlayField";
	}


	/* (non-Javadoc)
	 * @see gbang.datatype.GTGEObject#initAttributes()
	 */
	@Override
	public void initAttributes() {
		if (ATTRIBS == null) {
			ATTRIBS = new HashMap<String,GbDataType>();
			ATTRIBS.put("background", new GbString()); // path to
background
			ATTRIBS.put("bgwidth", new GbInteger());
			ATTRIBS.put("bgheight", new GbInteger());

			ATTRIBS.put("height", new GbInteger()); // bound the
height of the playfield (the image can be bigger!
			ATTRIBS.put("width", new GbInteger()); // bound the
width of the playfield

			// if sprite you want to set a sprite to be the
center of the board, set
			// center to that sprite object. This is useful e.g.
when you have a wide
			// image for a playfield and want the board to scroll
as w/ you @ the center
			// as you move across the screen.
			ATTRIBS.put("center", new GbSprite());
		}

	}
```

```
      // ____________________________________________________ TRANSLATOR
METHODS

      public String javaDeclare() {
//          String decl = super.javaDeclare();
            String decl = this.javaType() + " " + this.varName + " =
new PlayField();\n";

            decl += "Background background;\n";
            return decl;
      }

      // written during initVar / instantiation
      public String javaInstantiate() {
            StringBuffer sb = new StringBuffer();
            sb.append(super.javaInstantiate());
            //sb.append(this.varName + " = new " + this.javaType() +
"();\n");
            // add initialization stuff

            // deal with the associated SPRITE_GROUP
            sb.append("background = new ImageBackground(getImage("
                        + this.getPropertyValue("background",
"\"bgroung.png\"")
                        + "), "
                        + this.getPropertyValue("bgwidth", "640") + ", "
                        + this.getPropertyValue("bgheight", "480")
                        + ");\n");

      sb.append(String.format("%s.setBackground(background);\n\n",
this.varName));
            return sb.toString();
      }



      public void setProperty(String name, GbDataType type, Expression
value) {
            super.setProperty(name, type, value, ATTRIBS);
      }


      /**
       * Return the value <tt>GbDataType</tt> for the property set to
       * <tt>name</tt>.
       *
       * Note that by the time we're using this, everything should be
semantically checked,
       * and we should be able to monkey-rig the <tt>AST</tt> that's
stored in the
       * <tt>Expression</tt>-half of this <tt>properties</tt> hashmap
to just return the
       * String in there (the <tt>GbDataType</tt> might not do so
easliy)
       *
```

```
       * @param name    the name of the property you want to get.
       */
//    public GbDataType getProperty(String name) {
//          return null;
//    }
}


=====================================================
src/java/gbang/datatype/GbScopeContainer.java
=====================================================
/* FILE: GbScopeContainer.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import gbang.GbFunctionTable;
import gbang.statement.Expression;
import gbang.statement.Function;
import gbang.statement.Statement;

/**
 *
 */
public interface GbScopeContainer {

      public void addVar(String name, GbDataType var);
      public GbDataType getVar(String name);
      public GbDataType getCurrentVar(String name);

      public void addStatement(Statement stmt) throws
GbIllegalNestingException;
      public void addExpression(Expression exp) throws
GbIllegalNestingException;

      public boolean isLegalStatement(Statement stmt);

      public void setParentScope(GbScopeContainer parent);
      public GbScopeContainer getParentScope();

      public GbSymbolTable getSymbolTable();

      /**
       * Fetch the function table.
       * Note that this allows us to nest functions into specific
       * scopes, but we're always just passing around the global one
       * for now
       *
       * @return  the <tt>GbFunctionTable</tt>
       */
      public GbFunctionTable getFunctionTable();

      /**
       * Fetch the function identified by <tt>name</tt> from the
       * local <tt>GbFunctionTable</tt>
```

```
	 *
	 * @param name    the name of the function you're looking for
	 * @return        the function object identified by <tt>name</tt>
	 */
	public Function getFunction(String name);

}
```

```
=======================================================
src/java/gbang/datatype/GbSound.java
=======================================================
/* FILE: GbSound.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.util.HashMap;

import gbang.statement.Expression;

/**
 *
 */
public class GbSound extends GTGEObject {

	public static HashMap<String,GbDataType> ATTRIBS;



	@Override
	public String javaType() {
		return "Sound";
	}

	/* (non-Javadoc)
	 * @see gbang.datatype.GTGEObject#initAttributes()
	 */
	@Override
	public void initAttributes() {
		// TODO Auto-generated method stub

	}

	public void setProperty(String name, GbDataType type, Expression
value) {
		super.setProperty(name, type, value, ATTRIBS);
	}

}
```

```
=======================================================
src/java/gbang/datatype/GbSprite.java
=======================================================
```

```
/* FILE: Sprite.java
 * $Id$
 *
 * The G! Language
 */

/**
 * Wrapper class for Sprite objects in GBang
 * @author Amortya Ray
 * @version 1.0
 */
package gbang.datatype;

import java.util.HashMap;

import gbang.statement.Expression;

public class GbSprite extends GTGEObject {
      public static HashMap<String,GbDataType> ATTRIBS;
      public static HashMap<String,String> TRANSLATE;

      // we stuff every Sprite into a SpriteGroup
      protected GbSpriteGroup sister;

      String path;
      int rows;
      int cols;
      boolean loop;


      //Default constructor
      public GbSprite() {
            super();
      }

      //Parameterized constructor
      public GbSprite(String p, int rows, int cols, boolean loop){
            this();
            this.path=p;
            this.rows=rows;
            this.cols=cols;
            this.loop=loop;
      }

      public void initAttributes() {
            if (ATTRIBS == null) {
                  ATTRIBS = new HashMap<String,GbDataType>();
                  ATTRIBS.put("animate", new GbBoolean());
                  ATTRIBS.put("loop_animate", new GbBoolean());
                  ATTRIBS.put("x", new GbDouble());
                  ATTRIBS.put("y", new GbDouble());
                  ATTRIBS.put("location", new GbCoordinate()); // yeah?
                  ATTRIBS.put("active", new GbBoolean());
//                ATTRIBS.put("start_animation_frame", GbInteger.class);
//                ATTRIBS.put("stop_animation_frame", GbInteger.class);
                  ATTRIBS.put("horizontal_speed", new GbDouble());
                  ATTRIBS.put("vertical_speed", new GbDouble());
```

```
                  ATTRIBS.put("move_x", new GbDouble());
                  ATTRIBS.put("move_y", new GbDouble());
                  ATTRIBS.put("image", new GbString()); // path to
sprite
                  ATTRIBS.put("nrows", new GbInteger()); // number of
rows in image
                  ATTRIBS.put("ncols", new GbInteger()); // number of
columns in image
            }

            if (TRANSLATE == null) {
                  TRANSLATE = new HashMap<String,String>();
                  TRANSLATE.put("animate", "Animate");
                  TRANSLATE.put("loop_animate", "LoopAnim");
                  TRANSLATE.put("x", "X");
                  TRANSLATE.put("y", "Y");
                  TRANSLATE.put("coordinate", "XXXX"); // not direct
                  TRANSLATE.put("active", "Active");
//                ATTRIBS.put("start_animation_frame",
"animationFrame");
//                ATTRIBS.put("stop_animation_frame", GbInteger.class);
                  TRANSLATE.put("horizontal_speed", "HorizontalSpeed");
                  TRANSLATE.put("vertical_speed", "VerticalSpeed");
                  TRANSLATE.put("image", "Image"); // path to sprite
                  TRANSLATE.put("nrows", "XXXX"); // number of rows in
image
                  TRANSLATE.put("ncols", "XXXX"); // number of columns
in image
                  TRANSLATE.put("move_x", "moveY"); // number of
columns in image
                  TRANSLATE.put("move_y", "moveX"); // number of
columns in image

            }
      }

      //
_____________________________________________________________________
TRANSLATION METHODS

      @Override
      public String javaDeclare() {
            String decl = super.javaDeclare();
            // Sprites have a sister SPRITE_GROUP we stuff them into
            decl += this.sister.javaDeclare();
            return decl;
      }

      // written during initVar / instantiation
      public String javaInstantiate() {
            StringBuffer sb = new StringBuffer();
            sb.append(super.javaInstantiate());
            sb.append(this.javaInitializeFromProps());

            // set start position
//          sb.append(String.format("%s.setX(%f);\n", this.varName,
this.getX()));
```

```
//          sb.append(String.format("%s.setY(%f);\n", this.varName,
this.getY()));

            // deal with the associated SPRITE_GROUP
            sb.append(sister.varName + " = new " + sister.javaType() +
"(\"" + this.varName + "\");\n");
            sb.append(String.format("%s.add(%s);\n\n",
this.sister.varName, this.varName));
            return sb.toString();
      }

      public String javaInitializeFromProps() {
            StringBuffer sb = new StringBuffer();
            // load images into sprite
            if (this.isMultiImage()) {

      sb.append(String.format("%s.setImages(getImages(%s,%d,%d));\n",
                              this.varName, this.getImagePath(),
nCols(), nRows())
                              );
            } else {
//
      sb.append(String.format("%s.setImage(getImage(%s));\n",
//                            this.varName, this.getImagePath())
//                            );

      sb.append(String.format("%s.setImages(getImages(%s,%d,%d));\n",
                              this.varName, this.getImagePath(), 1, 1)
                              );


            }

            for (String key : this.properties.keySet()) {
                  if (!(key.equals("background") || key.equals("nrows")
                              || key.equals("ncols")
                              || key.equals("image")
                              )) {
                        sb.append(this.varName + "." +
                                    GTGEObject.javaSetter(key,
this.properties.get(key))
                                    + ";\n");
                  }
            }
            return sb.toString();
      }

      public boolean isMultiImage() {
            return ((this.properties.get("nrows") != null)
                        && (this.properties.get("ncols") != null));
      }

      public String getImagePath() {
            String path = "\"" + this.getPropertyValue("image") + "\"";
            if (path == null) {
                  path = "DEFUALT_PATH";
            }
```

```
            return path;
      }

      public int nRows() {
            String n = this.getPropertyValue("nrows");
            if (n == null) {
                  n = "1";
            }
            return Integer.parseInt(n);
      }

      public int nCols() {
            String n = this.getPropertyValue("ncols");
            if (n == null) {
                  n = "1";
            }
            return Integer.parseInt(n);
      }

      public double getX() {
            return Double.parseDouble(this.getPropertyValue("x",
"0.0"));
      }

      public double getY() {
            return Double.parseDouble(this.getPropertyValue("y",
"0.0"));
      }
      //
_____________________________________________________________________
_____________________

      public void setProperty(String name, GbDataType type, Expression
value) {
            super.setProperty(name, type, value, ATTRIBS);
      }

      //Method to return type
      public String getType(){
            return "Sprite";
      }

      @Override
      public String javaType() {
            return "AnimatedSprite";
      }

      //Copy method
      public GbSprite copy(){
            return new GbSprite(this.path, this.rows, this.cols,
this.loop);
      }
      public static void main(String[] args) {
            GbSprite hero = new GbSprite();

      }
```

```
	/**
	 * @return the sister
	 */
	public GbSpriteGroup getSister() {
		return this.sister;
	}

	/**
	 * @param sister the sister to set
	 */
	public void setSister(GbSpriteGroup sister) {
		this.sister = sister;
	}
}


=======================================================
src/java/gbang/datatype/GbSpriteGroup.java
=======================================================
/* FILE: SpriteGroup.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.util.HashMap;

import gbang.statement.Expression;

/**
 *
 */
public class GbSpriteGroup extends GbSprite {

	public static HashMap<String,GbDataType> ATTRIBS;

	// _____________________________________________________________
TRANSLATION
	@Override
	public String javaType() {
		return "SpriteGroup";
	}

	public String javaDeclare() {
		return this.javaType() + " " + this.varName + ";\n";
	}

	public String javaInstantiate() {
		return this.varName + " = new " + this.javaType() + "();\n";
	}


	/* (non-Javadoc)
	 * @see gbang.datatype.GTGEObject#initAttributes()
	 */
	@Override
```

```
	public void initAttributes() {
		if (ATTRIBS == null) {
			ATTRIBS = new HashMap<String,GbDataType>();
		}
	}

	public void setProperty(String name, GbDataType type, Expression
value) {
		super.setProperty(name, type, value, ATTRIBS);
	}

}
```

```
=======================================================
src/java/gbang/datatype/GbString.java
=======================================================
/* FILE: GbString.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 * Wrapper class for all objects of type String
 * @author Amortya Ray
 * @version 1.0
 */
public class GbString extends GbDataType {
	protected String val;

	public GbString(String str) {
		this.val = str;
	}

	public GbString() {
		this("");
	}

	//Method that returns the data type of the object
	public String getType(){
		return "String";
	}

	//Copy method
	public GbDataType copy(){
		return new GbString(val);
	}

	public GbString concat(GbString s){
		if (s instanceof GbString){
			val.concat(s.getString());
			return this;
		}
		return  (GbString)error("Incompatible Types");
	}
```

```
	//Method to return the String value of a GbString object
	private String getString(){
		return val;
	}

	@Override
	public String javaType() {
		return "String";
	}
} // class GbString


=======================================================
src/java/gbang/datatype/GbSymbolTable.java
=======================================================
/* FILE: GbSymbolTable.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.util.HashMap;
import java.util.Hashtable;

public class GbSymbolTable {

	private static final long serialVersionUID =
-4231364031295827427L;

	//private Hashtable<String, GbIdentifier> table;
	protected HashMap<String, GbDataType> table;

	protected GbSymbolTable outer;

	public GbSymbolTable(GbSymbolTable st) {
		//table = new Hashtable<String, GbIdentifier>();
		this.table = new HashMap<String, GbDataType>();
		outer = st;
	}

	public GbSymbolTable() {
		this(null);
	}

	public void put(String name, GbDataType object) {
		this.table.put(name, object);
	}

	/**
	 * Looks for <tt>name</tt> in the current scope.
	 * If the name isn't found, it asks the parent scope to get
<tt>name</tt>
	 * If there is no parent scope and the name isn't found, returns
null.
	 *
```

```
	 * Note: recursion is happening here.
	 *
	 * @param name    name of the variable you are after
	 * @return        the <tt>GbDataType</tt> assoicated to
<tt>name</tt>
	 *                      isn't found in current our parent scopes,
return <tt>null</tt>.
	 */
	public GbDataType get(String name) {
		if (this.table.containsKey(name)) {
			return this.table.get(name);
		} else {
			if (this.outer == null) {
				return null;
			} else {
				return this.outer.get(name);
			}
		}
	} // method get

	public GbDataType getCurrentVar(String name) {
		if (this.table.containsKey(name)) {
			return this.table.get(name);
		} else {
			return null;
		}
	}

	/**
	 * @return the outer
	 */
	public GbSymbolTable getOuter() {
		return this.outer;
	}

	/**
	 * @param outer the outer to set
	 */
	public void setOuter(GbSymbolTable outer) {
		this.outer = outer;
	}

	/**
	 * @return the table
	 */
	public HashMap<String, GbDataType> getTable() {
		return this.table;
	}

	/**
	 * @param table the table to set
	 */
	public void setTable(HashMap<String, GbDataType> table) {
		this.table = table;
	}
```

```
}


=======================================================
src/java/gbang/datatype/GbVoid.java
=======================================================
/* FILE: GbVoid.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 *
 */
public class GbVoid extends GbDataType {

      public String javaType() { return "void"; }
}


=======================================================
src/java/gbang/datatype/GbWalker.java
=======================================================
/* FILE: GbWalker.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

public class GbWalker {

}


=======================================================
src/java/gbang/datatype/GbWalkerTokenTypes.java
=======================================================
/* FILE: GbWalkerTokenTypes.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

/**
 *
 */
public class GbWalkerTokenTypes {

}


=======================================================
src/java/gbang/datatype/GTGEObject.java
```

```
=====================================================
/* FILE: GTGEObject.java
 * $Id$
 *
 * The G! Language
 */
package gbang.datatype;

import java.util.HashMap;

import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.antlr.GBangExpressionWalker;
import gbang.statement.Expression;

/**
 * GTGE objects should subclass this
 *          -> Sprite / SpriteGroup / etc.
 */
public abstract class GTGEObject extends GbDataType {
      public static HashMap<String,String> GLOBAL_TRANSLATE = new
HashMap<String,String>();
      static {
            GLOBAL_TRANSLATE = new HashMap<String,String>();
            GLOBAL_TRANSLATE.put("animate", "Animate");
            GLOBAL_TRANSLATE.put("loop_animate", "LoopAnim");
            GLOBAL_TRANSLATE.put("x", "X");
            GLOBAL_TRANSLATE.put("y", "Y");
            GLOBAL_TRANSLATE.put("coordinate", "XXXX"); // not direct
            GLOBAL_TRANSLATE.put("active", "Active");
//          ATTRIBS.put("start_animation_frame", "animationFrame");
//          ATTRIBS.put("stop_animation_frame", GbInteger.class);
            GLOBAL_TRANSLATE.put("horizontal_speed", "HorizontalSpeed");
            GLOBAL_TRANSLATE.put("vertical_speed", "VerticalSpeed");
            GLOBAL_TRANSLATE.put("image", "Image"); // path to sprite
            GLOBAL_TRANSLATE.put("nrows", "XXXX"); // number of rows in
image
            GLOBAL_TRANSLATE.put("ncols", "XXXX"); // number of columns
in image
      }



      /** Holds the name, GbDataType pairs that each GTGE object can
use */
      protected static HashMap<String, GbDataType> ATTRIBS;
      static {
            ATTRIBS = new HashMap<String,GbDataType>();

            // sprite
            ATTRIBS.put("background", new GbString()); // path to
background
            ATTRIBS.put("bgwidth", new GbInteger());
            ATTRIBS.put("bgheight", new GbInteger());

            ATTRIBS.put("height", new GbInteger()); // bound the height
of the playfield (the image can be bigger!
```

```
            ATTRIBS.put("width", new GbInteger()); // bound the width
of the playfield

            // if sprite you want to set a sprite to be the center of
the board, set
            // center to that sprite object. This is useful e.g. when
you have a wide
            // image for a playfield and want the board to scroll as w/
you @ the center
            // as you move across the screen.
            ATTRIBS.put("center", new GbSprite());

      }


      /**
       * These hold the properties that were set during program
compilation
       * for each particular object
       */
      protected HashMap<String, Expression> properties;


      public GTGEObject() {
            this.properties = new HashMap<String,Expression>();
            this.initAttributes();
      }

      /**
       * Override this method to initialize atributes for each GTGE
object
       * on creation.
       */
      public abstract void initAttributes();

      /**
       * Might be useufl for semanting checking ... GTGE
       * objects should be initialized before a run
       *
       * @return true/false if object is initialized
       */
      public boolean isInitialized() {
            return (this.properties.size() != 0);
      }


      /**
       * Returns the string that should be written when the object
       * is finally constructed (and initialized)
       * @return
       */
      public String javaInstantiate() {
            StringBuffer sb = new StringBuffer();
            sb.append(super.javaInstantiate());
            //sb.append(this.varName + " = new " + this.name + "();\n");
            // add initialization stuff
            return sb.toString();
```

```
	}


	/**
	 * Parses keyword arguments from a <tt>KWARGS</tt> tree and save
them in
	 * their respecitve slot in <tt>this.properties</tt>
	 *
	 * Note that we're only saving the child of the EXPR tree here.
	 *
	 * Two ways to set the x property of a sprite:
	 *
	 * sprite->x = something
	 * sprite->properties(x=something)
	 *
	 * @param props   the <tt>KWARGS</tt> root of the AST for the
properties
	 */
	public void setProperties(AST props) {
		int counter = 1;
		String name = null;
		Expression exp = null;
		if (props.getNumberOfChildren() > 0) {
			props = props.getFirstChild();
			do {
				int position = counter % 3;
				switch (position) {
				case 1:
					name = props.getText();
					break;
				case 2:
					// this is an equals
					break;
				case 0:
					exp = new
Expression(props.getFirstChild());
					//exp = new Expression(props);
					this.properties.put(name, exp);
					break;
				}
				counter++;
			} while((props = props.getNextSibling()) != null);
		}
	}

	// ____________________________________________________________ JAVA
TRANSLATION

	// TODO: check semantics of property?
	public String createJavaSetter(String property, AST expression) {
		String result = null;

		// special cases
		result = this.varName + ".";
		// check to see if first letter is upper case -- we won't
put a set and Caps it
```

```
            if
(property.substring(0,1).equals(property.substring(0,1).toLowerCase()))
{
                  result += "set";
            }

            result += this.translateProperty(property)
                  + "(" + Expression.toJava(expression) + ");";
            return result;
      }

      public String createJavaGetter(String property) {
            return this.varName + "." + "get" +
translateProperty(property);
      }

      public String translateProperty(String name) {
            // TODO: work on translating g! property_names to jave
propertyNames
            //return name;
            return GLOBAL_TRANSLATE.get(name);
      }

      public static String javaGetter(String prop) {
            return "get" + GLOBAL_TRANSLATE.get(prop) + "()";
      }

      public static String javaSetter(String prop, Expression expr) {
            GBangExpressionWalker ew = new GBangExpressionWalker();
            try {
                  return "set" + GLOBAL_TRANSLATE.get(prop) + "(" +
ew.expression(expr.getTree()) + ")";
            } catch (RecognitionException e) {
                  // TODO Auto-generated catch block
                  return "set"  + GLOBAL_TRANSLATE.get(prop) + "(XXX)";
            }
      }


      public String getPropertyValue(String name, String dfault) {
            //Expression expr = this.getPropertyValue(name);
            String expr = this.getPropertyValue(name);
            if (expr == null) {
                  return dfault;
            }
            return expr;
      }

      public String getPropertyValue(String name) {
            Expression expr = this.properties.get(name);
            String ret = null;
            if (expr == null) {
                  System.out.println("[" + name + "] -- it hasn't been
initalized");
                  //ret = " // [" + name +"] is not a valid property";
            } else {
                  ret = Expression.toJava(this.properties.get(name));
```

```
                  if (ATTRIBS.get(name) instanceof GbString) {
                        ret = "\"" + ret + "\"";
                  }
            }
            return ret;
      }

      //
```

```
      /**
       * Ensures that the given name and value pair occur in the attribs
       * HashMap, throws an error if not.
       *
       * The child classes call this -- the functionality is here for
convenience
       *
       * @param name          the name of the variable tha'ts being set
       * @param val           the value of the parameter
       * @param attribs the child's attribute info
       *
       * @throws  GbNoSuchPropertyException if <tt>name</tt> is not a
valid
       *                variable as defined in <tt>attribs</tt>
       * @throws  GbMismatchPropertyTypeException if the datatype
doesn't
       *                match with the name
       */
      public void setProperty(String name, GbDataType type, Expression
value,
                  HashMap<String,GbDataType> attribs)
                  throws GbNoSuchPropertyException,
GbMismatchPropertyTypeException {
            GbDataType target = attribs.get(name);
            if (name == null) {
                  throw new GbNoSuchPropertyException("No such property
'" + name
                              + "' on variable type +" +
this.getClass());
            }

            if (!(type.getClass() != target.getClass())) {
                  throw new GbMismatchPropertyTypeException("Property
'" + name
                              + "' is not of class " + type.getClass());
            }

            // seems kosher
            this.properties.put(name, value);
      } // method setProperty

}
```

==========

src/java/gbang/event/GbCollision.java

```
=======================================================
/* FILE: GbCollision.java
 * $Id$
 *
 * The G! Language
 */
package gbang.event;


/**
 * This will be used for something, I don't know what just yet.
 */
public class GbCollision extends GbEvent {

}


=======================================================
src/java/gbang/event/GbCollisionManager.java
=======================================================
/* FILE: GbCollisionManager.java
 * $Id$
 *
 * The G! Language
 */
package gbang.event;

import java.util.HashMap;

import gbang.datatype.GbSprite;

/**
 * Keeps track of all who is coliding into what, when, where, and how
 */
public class GbCollisionManager {

	/** From g! to Java */
	public static HashMap<String,String> BANG_TYPES = new
HashMap<String,String>();
	static {
		// initialize BANG_MAP
		BANG_TYPES.put("!left", "RIGHT_LEFT_COLLISION");
		BANG_TYPES.put("!right", "LEFT_RIGHT_COLLISION");
		BANG_TYPES.put("!top", "BOTTOM_TOP_COLLISION");
		BANG_TYPES.put("!bottom", "TOP_BOTTOM_COLLISION");
	}

	// _____________________________________ INSTANCE VARS

	protected HashMap<GbSprite,GbSprite> collisionTracker;

	public GbCollisionManager() {
		this.collisionTracker = new HashMap<GbSprite,GbSprite>();
	}
}
```

```
=========================================================
src/java/gbang/event/GbEvent.java
=========================================================
/* FILE: Event.java
 * $Id$
 *
 * The G! Language
 */
package gbang.event;

/**
 *
 */
public class GbEvent {

}


=========================================================
src/java/gbang/event/GbKeyPress.java
=========================================================
/* FILE: KeyPress.java
 * $Id$
 *
 * The G! Language
 */
package gbang.event;

/**
 *
 */
public class GbKeyPress extends GbEvent {

}


=========================================================
src/java/gbang/statement/ArithmeticExpression.java
=========================================================
/* FILE: ArithmeticExpression.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

/**
 *
 */
public class ArithmeticExpression extends Expression {

      public ArithmeticExpression() {

      }
}
```

```
=========================================================
src/java/gbang/statement/Assignment.java
=========================================================
/* FILE: Assignmnet.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.CommonAST;
import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.antlr.GBangExpressionWalker;
import gbang.antlr.GBangTokenTypes;
import gbang.datatype.GbScopeContainer;

/**
 *
 */
public class Assignment extends Expression {
      protected Expression lhs;
      //protected ArrayList<Expression> rhs;
      protected Expression rhs;
      //protected boolean fromDeclaration;

      public Assignment() {}

      public Assignment(AST node) {
            this.lhs = new Expression(node);
            this.rhs = new Expression(node.getNextSibling());
            this.tree = node;
            //this.fromDeclaration = false;

            //this.rhs = new ArrayList<Expression>();
//          AST right = node.getNextSibling();
//
//          if (right.getNumberOfChildren() == 0) {
//                // stop
//          }
      }

      /**
       * Construct an assignment statement when the assignment happens
       * at the same time as a variable declaration.
       *          Integer something = 7;
       *
       * @param lhs     lefthand side of assignment
       * @param rhs     righthand side of assignment
       * @return        new Assignment object
       */
      public static Assignment fromDeclaration(AST lhs, AST rhs,
GbScopeContainer scope) {
            Assignment assign = new Assignment();
            assign.lhs = new Expression(lhs);
            assign.rhs = new
Expression(rhs.getFirstChild().getFirstChild());
```

```
            assign.scope = scope;

            // the tree structure of this assignment is different than
a normal
            // assignment, due to parser doing some nifty on-the-fly
stuff to the
            // ADT when we do an inling assignment (eg. Integer a = 10;)
            // see the varDeclarator --> varInitializer song and dance
in the grammar
            System.out.println("Printing decls tree:");

            rhs=rhs.getFirstChild();rhs=rhs.getFirstChild();

            String leftText=lhs.getText();
            int leftType=lhs.getType();

            AST tempTree=new CommonAST();
            tempTree.setType(GBangTokenTypes.ASSIGN);
            tempTree.setText("=");
            //System.out.println(tempTree.toStringTree());
            AST firstChild=new CommonAST();
            firstChild.setType(leftType);
            firstChild.setText(leftText);
            tempTree.addChild(firstChild);
            tempTree.addChild(rhs);
            //System.out.println("New Tree="+tempTree.toStringTree());
            assign.tree=tempTree;
            return assign;
      }

      public String toJava() {
            String string;
            GBangExpressionWalker ew = new GBangExpressionWalker();
            try {
                  string = ew.expression(this.tree) + ";\n";
            } catch (RecognitionException e) {
                  string = "EXPRESSION PARSE ERROR";
            }
            return string;

      }

/*
      public void checkSemantics() {
            GBangSemanticWalker2 walker = new GBangSemanticWalker2();
            walker.setCurrentScope(scope);

            if (this.fromDeclaration) {
                  // need to make a new AST just like a normal
assignment would look:
                  // does look like:
                  //    lhs
                  //    =
                  //          EXPR
                  //                rhs
                  // should look like:
                  // =
```

```
                  //          lhs
                  //          rhs
            } else {
                  try {
                        walker.expression(this.tree);
                        if (walker.gbException.nErrors() > 0) {
                              System.out.println("*** SEMANTICS ***" +
walker.gbException.getException());
                        }
                  } catch (RecognitionException e) {
                        throw new GbException("Semantic error: " +
e.getMessage());
                  }
            }
      }

      */

}


=======================================================
src/java/gbang/statement/BangExpression.java
=======================================================
/* FILE: BangExpression.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.collections.AST;
import gbang.datatype.GbSprite;

/**
 *
 */
public class BangExpression extends BooleanExpression {

      public BangExpression(AST t) {
            super(t);
      }

      public AST getLHS() {
            return this.tree.getFirstChild();
      }

      public AST getRHS() {
            return this.tree.getFirstChild().getNextSibling();
      }

      public GbSprite getRightObject() {
            return (GbSprite)
this.scope.getVar(this.getRHS().getText());
      }
```

```
	public GbSprite getLeftObject() {
		return (GbSprite)
this.scope.getVar(this.getLHS().getText());
	}
}


=======================================================
src/java/gbang/statement/Block.java
=======================================================
/* FILE: Block.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import java.util.ArrayList;

import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.GbFunctionTable;
import gbang.antlr.GBangSemanticWalker2;
import gbang.antlr.GBangWalker;
import gbang.datatype.GbDataType;
import gbang.datatype.GbException;
import gbang.datatype.GbIllegalNestingException;
import gbang.datatype.GbScopeContainer;
import gbang.datatype.GbSymbolTable;

/**
 * A block of statements
 */
public class Block extends Statement implements GbScopeContainer {
	protected AST tree;

	protected GbSymbolTable symbolTable;
	protected GbScopeContainer parentScope;
	protected GbFunctionTable functionTable;

	protected ArrayList<Statement> statements;

	public Block() {
		this.symbolTable = new GbSymbolTable();
		this.functionTable = new GbFunctionTable();
		this.statements = new ArrayList<Statement>();
	}

	public Block(GbScopeContainer parent) {
		this();
		this.parentScope = parent;
	}

	/* (non-Javadoc)
	 * @see gbang.statement.Statement#toJava()
	 */
	@Override
```

```
	public String toJava() {
		StringBuilder sb = new StringBuilder();
		for (Statement s : this.statements) {
			sb.append(s.toJava());
		}
		return sb.toString();
	}

	public void buildBody(AST body, GBangWalker walker) {
		walker.setCurrentScope(this);
		try {
			do {
				walker.statement(body);
				body = body.getNextSibling();
			} while(body != null);
		} catch (RecognitionException e) {
			throw new GbException("Error while building body of
block");
		}
		walker.setCurrentScope(this.parentScope);
	}

	public void addExpression(Expression exp) throws
GbIllegalNestingException {
		this.statements.add(exp);
	}

	public void addStatement(Statement stmt) throws
GbIllegalNestingException {
		this.statements.add(stmt);
	}

	public void addVar(String name, GbDataType var) {
		this.symbolTable.put(name, var);
	}

	public GbDataType getVar(String name) {
		return this.symbolTable.get(name);
	}

	public GbDataType getCurrentVar(String name) {
		return this.symbolTable.getCurrentVar(name);
	}

	public GbScopeContainer getParentScope() {
		return this.parentScope;
	}

	public void setParentScope(GbScopeContainer parent) {
		this.parentScope = parent;
		this.symbolTable.setOuter(parent.getSymbolTable());
		this.functionTable = parent.getFunctionTable();
	}

	public boolean isLegalStatement(Statement stmt) {
		return true;
	}
```

```
	/* (non-Javadoc)
	 * @see gbang.datatype.GbScopeContainer#getSymbolTable()
	 */
	public GbSymbolTable getSymbolTable() {
		return this.symbolTable;
	}

	/**
	 * @return the functionTable
	 */
	public GbFunctionTable getFunctionTable() {
		return this.functionTable;
	}

	/**
	 * @param functionTable the functionTable to set
	 */
	public void setFunctionTable(GbFunctionTable functionTable) {
		this.functionTable = functionTable;
	}

	/**
	 * @see
gbang.datatype.GbScopeContainer#getFunction(java.lang.String)
	 */
	public Function getFunction(String name) {
		return this.functionTable.get(name);
	}

	/**
	 * @param symbolTable the symbolTable to set
	 */
	public void setSymbolTable(GbSymbolTable symbolTable) {
		this.symbolTable = symbolTable;
	}

	/**
	 * Delegate to each statement in this block to ensure
	 * their semantics are correct
	 * @throws RecognitionException
	 * @see gbang.statement.Statement#checkSemantics()
	 */
	@Override
	public void checkSemantics() throws RecognitionException {
		walker.setCurrentScope(this);
		walker.statement(this.tree, 0);

//		for (Statement stmt : this.statements) {
//			stmt.checkSemantics();
//		}
	}


	/**
	 * @return the tree
	 */
```

```
	public AST getTree() {
		return this.tree;
	}

	/**
	 * @param tree the tree to set
	 */
	public void setTree(AST tree) {
		this.tree = tree;
	}



} // class Block


============================================================
src/java/gbang/statement/BooleanExpression.java
============================================================
/* FILE: BooleanExpression.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.collections.AST;

/**
 *
 */
public class BooleanExpression extends Expression {

	public BooleanExpression(AST t) {
		super(t);
	}
}


============================================================
src/java/gbang/statement/Declaration.java
============================================================
/* FILE: Declaration.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import gbang.datatype.GbDataType;

/**
 *
 */
public class Declaration extends Statement {
	protected GbDataType var;
	protected String name;
```

```
public Declaration(GbDataType var, String name) {
	this.var = var;
	this.name = name;
}

/* (non-Javadoc)
 * @see gbang.statement.Statement#toJava()
 */
@Override
public String toJava() {
	String ret = this.var.javaType() + " " + this.name + ";\n";
	return ret;
}




/**
 * @return the name
 */
public String getName() {
	return this.name;
}

/**
 * @param name the name to set
 */
public void setName(String name) {
	this.name = name;
}

/**
 * @return the var
 */
public GbDataType getVar() {
	return this.var;
}

/**
 * @param var the var to set
 */
public void setVar(GbDataType var) {
	this.var = var;
}

/**
 * The semantics of declarations are checked when they
 * are instantiated by the first pass of the walker.
 *
 * This method will just ignore the request to check its own
 * semantics, because if we've gotten to this point by now, they
 * must be correct.
 */
@Override
public void checkSemantics() {
```

```
	}

}


=======================================================
src/java/gbang/statement/Expression.java
=======================================================
/* FILE: Expression.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.antlr.GBangExpressionWalker;
import gbang.antlr.GBangSemanticWalker2;
import gbang.antlr.GBangTokenTypes;
import gbang.antlr.GBangWalker;
import gbang.datatype.GbException;
import gbang.datatype.GbScopeContainer;

/**
 *
 */
public class Expression extends Statement {
	protected AST tree;
	protected GbScopeContainer scope;
	public Expression() {}

	/**
	 * Some logic to walk an expression and change it to java code
	 * @param expr
	 */
	public static String toJava(AST expr) {
		GBangExpressionWalker ew = new GBangExpressionWalker();

		try {
			return ew.expression(expr);// + ";\n";
		} catch (RecognitionException e) {
			return "javaEXPR";
		}
	}

	public static String toJava(Expression expr) {
		return Expression.toJava(expr.getTree());
	}

	public static Expression getInstance(AST t, GBangWalker walker) {
		Expression expr = null;

		////////////////////////////////////////////
		// what type of expression is it?
		// func_call -> first child is of type FUNC_CALL?
		boolean bangOp = false;
```

```
            switch (t.getType()) {
            case GBangTokenTypes.BANG:
            case GBangTokenTypes.BRIGHT:
            case GBangTokenTypes.BLEFT:
            case GBangTokenTypes.BTOP:
            case GBangTokenTypes.BBOTTOM:
                  bangOp = true;
                  expr = new BangExpression(t);
                  break;
            }

            if (!bangOp) {
                  if (t.getType() == GBangTokenTypes.ASSIGN) {
                        if (t.getFirstChild().getType() ==
GBangTokenTypes.PROP) {
                              expr = new
PropAssignment(t.getFirstChild(), walker.getCurrentScope());
                        } else {
                              expr = new Assignment(t.getFirstChild());
                        }
                  } else {
                        AST next = t.getNextSibling();
                        if (next != null) {
                              if (next.getType() ==
GBangTokenTypes.FUNC_CALL) {
                                    // ( is set to FUNC_CALL by parser
                                    expr = new FunctionCall(t);
                              }
                        }
                  }
            }

            // unhandled?
            if (expr == null) {
                  expr = new Expression(t);
            }

            expr.tree = t;

            expr.scope = walker.getCurrentScope();
            return expr;
      }

      @Override
      public void checkSemantics() {
            GBangSemanticWalker2 walker = new GBangSemanticWalker2();
            walker.setCurrentScope(scope);

            try {
                  walker.expression(this.tree);
                  if (walker.gbException.nErrors() > 0) {
                        System.out.println("*** SEMANTICS ***" +
walker.gbException.getException());
                  }
            } catch (RecognitionException e) {
                  throw new GbException("Semantic error: " +
e.getMessage());
```

```
	}
}

public Expression(AST t) {
	this.tree = t;
}


/* (non-Javadoc)
 * @see gbang.statement.Statement#toJava()
 */
@Override
public String toJava() {
	String string = null;
	GBangExpressionWalker ew = new GBangExpressionWalker();
	try {
		string = ew.expression(this.tree);
	} catch (RecognitionException e) {
		string = "EXPRESSION PARSE ERROR";
	}
	return string;
}


public String parse() {
	return parse(tree);
}

/**
 * Why am I doing this manuallY?
 * Run the in order traversal
 * @param tree	the tree to parse
 * @return
 */
public static String parse(AST tree) {

	return null;
}

/**
 * @return the scope
 */
public GbScopeContainer getScope() {
	return this.scope;
}

/**
 * @param scope the scope to set
 */
public void setScope(GbScopeContainer scope) {
	this.scope = scope;
}

/**
 * @return the tree
 */
public AST getTree() {
```

```
            return this.tree;
      }

      /**
       * @param tree the tree to set
       */
      public void setTree(AST tree) {
            this.tree = tree;
      }


}
```

```
=======================================================
src/java/gbang/statement/For.java
=======================================================
/* FILE: For.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.antlr.GBangTokenTypes;
import gbang.antlr.GBangWalker;
import gbang.datatype.GbDataType;
import gbang.datatype.GbEmptyBodyException;
import gbang.datatype.GbIllegalNestingException;

/**
 *
 */
public class For extends Block {
      protected GbDataType varType;
      protected String var;
      protected Expression start;
      protected Expression stop;
      protected Expression step;

      public For(AST start, AST stop, AST rest, GBangWalker walker) {


      }

      public String toJava() {
            int incr=1;
            String rel="<=";
            if (step != null) {
                  incr = Integer.parseInt(step.toJava());
            }
            if(incr<0) rel=">=";
```

```
            String forbuff = "for (" + var + "=" + start.toJava() + ";
" + var + rel + stop.toJava() + "; " + var + "=" + var + "+(" + incr +
")) {";
            forbuff+= "\n";
            forbuff+= super.toJava();
            forbuff+= "\n";
            forbuff+= "}";
            return forbuff;
      }
      public For(AST tree, GBangWalker walker) {
            AST forVar, range, body;

            if (tree.getType() == GBangTokenTypes.FOR_TYPE) {
                  varType = GbDataType.getInstance(tree.getText());
                  forVar = tree.getNextSibling();
                  // put into symbol table
                  this.symbolTable.put(forVar.getText(), this.varType);
            } else {
                  forVar = tree;
            }
            range = forVar.getNextSibling();
            body = range.getNextSibling();

            if (body.getNumberOfChildren() == 0) {
                  throw new GbEmptyBodyException("Don't put an empty
body in the for loop");
            }

            this.setParentScope(walker.getCurrentScope());

            this.var = forVar.getText();

            this.start = new
Expression(range.getFirstChild().getFirstChild());
            this.start.setScope(this);
            this.stop = new
Expression(range.getFirstChild().getNextSibling().getFirstChild());
            this.stop.setScope(this);
            if (range.getNumberOfChildren() == 3) {
                  // there's a step
                  this.step = new
Expression(range.getFirstChild().getNextSibling().getNextSibling().getF
irstChild());
                  this.step.setScope(this);
            }


            this.buildBody(body.getFirstChild(), walker);
      }

      public boolean isLegalStatement(Statement stmt) throws
GbIllegalNestingException {
            boolean legal = true;

            if (stmt instanceof When) {
                  throw new GbIllegalNestingException("Can't use a When
statement in a For loop");
```

```
            }

            if (stmt instanceof Function) {
                  throw new GbIllegalNestingException("Can't define a
Function in a for loop.");
            }

            legal = this.parentScope.isLegalStatement(stmt);

            return legal;
      }

      @Override
      public void checkSemantics() throws RecognitionException {
            super.checkSemantics();
//          this.start.checkSemantics(); // check for arithmetic
expression
//          this.stop.checkSemantics();
//          this.step.checkSemantics();
//
//          super.checkSemantics();       // check the body
      }


      public void addStatement(Statement stmt) {
            this.isLegalStatement(stmt);

            this.statements.add(stmt);
      }

}


=======================================================
src/java/gbang/statement/ForRange.java
=======================================================
/* FILE: ForRange.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

/**
 *
 */
public class ForRange {

      protected Expression start;
      protected Expression end;
      protected Expression increment;



}
```

```
=====================================================
src/java/gbang/statement/Function.java
=====================================================
/* FILE: Function.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import java.util.ArrayList;
import java.util.HashMap;

import antlr.collections.AST;
import gbang.antlr.GBangWalker;
import gbang.datatype.GbDataType;
import gbang.datatype.GbIllegalNestingException;

/**
 *
 */
public class Function extends Block {

	protected String name;
	protected GbDataType returnType;

	protected HashMap<String,GbDataType> args;
	protected ArrayList<GbDataType> arguments;
	protected String[] argPositions;


	public Function() {
		super();
		this.returnType = null;
		this.args = new HashMap<String,GbDataType>();
	}

	public Function(AST type, AST name, AST args, AST body,
GBangWalker walker) {
		this();

		this.returnType = GbDataType.getInstance(type.getText());
		this.returnType.setDataType(type.getText());
		this.name = name.getText();
		this.returnType.setDataType(type.getText());
		this.parseArguments(args, walker);
		this.setParentScope(walker.getCurrentScope());

		// build body
		this.buildBody(body, walker);

	}

	public void parseArguments(AST node, GBangWalker walker) {
		this.arguments = new ArrayList<GbDataType>();
		if (node.getNumberOfChildren() > 0) {
```

```
                  node = node.getFirstChild();
                  do {
                        GbDataType dtype =
GbDataType.getInstance(node.getText());
                        dtype.setDataType(node.getText());
                        node = node.getNextSibling();
                        String name = node.getText();
                        this.arguments.add(dtype);
                        this.symbolTable.put(name, dtype);
                        node = node.getNextSibling();

                        if (node !=  null) {
                              node = node.getFirstChild();
                        } else {
                              break;
                        }
                  } while (true);
            }
      }


      @Override
      public String toJava() {
            StringBuilder sb = new StringBuilder();
            sb.append("\n\n");

            sb.append("public " + this.returnType.javaType() + " " +
this.name + "(");
            for (GbDataType t : this.arguments) {
                  sb.append(t.javaType() + " " + t.getVarName() + ", ");
            }
            if (this.arguments.size() > 1)
{ sb.deleteCharAt(sb.length()-1); }

            sb.append(")");
            sb.append(" { \n");
            sb.append(super.toJava());
            sb.append("}");
            sb.append("\n\n");
            return sb.toString();
      }

      public boolean isLegalStatement(Statement stmt) throws
GbIllegalNestingException {
            boolean legal = true;
            if (stmt instanceof When) {
                  throw new GbIllegalNestingException("Can't use a When
statement in a function call");
            }

            legal = this.parentScope.isLegalStatement(stmt);

            return legal;
      }

      public void addStatement(Statement stmt) {
            this.isLegalStatement(stmt);
```

```
            this.statements.add(stmt);
      }

      // _____________________________________________ GETTERS / SETTERS
      /**
       * @return the args
       */
      public ArrayList<GbDataType> getArgs() {
            return this.arguments;
      }

      /**
       * @param args the args to set
       */
      public void setArgs(HashMap<String,GbDataType> args) {
            this.args = args;
      }

      /**
       * @return the returnType
       */
      public GbDataType getReturnType() {
            return this.returnType;
      }

      /**
       * @param returnType the returnType to set
       */
      public void setReturnType(GbDataType returnType) {
            this.returnType = returnType;
      }

      /**
       * @return the name
       */
      public String getName() {
            return this.name;
      }

      /**
       * @param name the name to set
       */
      public void setName(String name) {
            this.name = name;
      }


}


=======================================================
src/java/gbang/statement/FunctionCall.java
=======================================================
/* FILE: FunctionCall.java
 * $Id$
```

```
 *
 * The G! Language
 */
package gbang.statement;

import java.util.ArrayList;

import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.antlr.GBangExpressionWalker;
import gbang.antlr.GBangTokenTypes;

/**
 *
 */
public class FunctionCall extends Expression {

	//protected HashMap<String,Expression> args;
	protected ArrayList<Expression> args;
	protected Function callee;
	protected String functionCall;


	public FunctionCall(AST tree) {
		this.functionCall = tree.getText();
		this.args = new ArrayList<Expression>();
		this.tree = tree;

		//tree = tree.getFirstChild().getNextSibling();
		tree = tree.getNextSibling().getNextSibling(); // skip the (

		// are there arguments?
		if (tree.getType() == GBangTokenTypes.ARGS) {
			do {
				tree = tree.getFirstChild(); // THIS IS ARGS
				Expression exp = new
Expression(tree.getFirstChild());
				args.add(exp);
			} while ((tree = tree.getNextSibling()) != null);
		}
		// parse the arguments
	}

//	public FunctionCall(Function call) {
//		this.args = new HashMap<String,Expression>();
//	}

	/* (non-Javadoc)
	 * @see gbang.statement.Statement#toJava()
	 */
	@Override
	public String toJava() {
		GBangExpressionWalker ew = new GBangExpressionWalker();
		String result = null;
		try {
			result = ew.expression(this.tree);
		} catch (RecognitionException e) {
```

```
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
            return result + ";\n";
      }

}


=======================================================
src/java/gbang/statement/If.java
=======================================================
/* FILE: If.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.antlr.GBangSemanticWalker2;
import gbang.antlr.GBangTokenTypes;
import gbang.antlr.GBangWalker;
import gbang.datatype.GbEmptyBodyException;
import gbang.datatype.GbException;
import gbang.datatype.GbIllegalNestingException;

/**
 *
 */
public class If extends Block {

      protected Expression test;
      protected Statement elseClause;

      protected boolean bodyBuilt;        // help with chaining if's
(why is our life so hard?)

      public If() {
            super();
            this.bodyBuilt = false;
      }

      public If(AST test, AST body, AST ifelse, GBangWalker walker)
throws RecognitionException {
            this();

            if (body.getNumberOfChildren() == 0) {
                  throw new GbEmptyBodyException("Don't put an empty
body in the IF");
            }

            this.setParentScope(walker.getCurrentScope());

            if (test.getType() == GBangTokenTypes.KEY_PRESS) {
                  this.test = new KeyPressExpression(test.getText());
```

```
            } else {
                  this.test = new Expression(test.getFirstChild());
            }

            this.test.setScope(this);

            this.buildBody(body.getFirstChild(), walker);
            bodyBuilt = true;

            if (ifelse.getNumberOfChildren() != 0) {
                  if (ifelse.getFirstChild().getType() ==
GBangTokenTypes.IF) {
                        walker.setCurrentScope(this);
                        walker.statement(ifelse.getFirstChild());
                        walker.setCurrentScope(this.parentScope);
                  } else {
                        // naked else
                        Block block = new Block();
                        walker.setCurrentScope(block);
                        block.buildBody(ifelse.getFirstChild(), walker);
                        this.elseClause = block;
                        walker.setCurrentScope(this.parentScope);
                  }
            }

            // did we add an else?
      }


      public boolean isLegalStatement(Statement stmt){
            boolean legal = true;
            if (stmt instanceof Function) {
                  throw new GbIllegalNestingException("Can't define a
Function within an If clause");
            }

            this.parentScope.isLegalStatement(stmt);

            return legal;
      }

      @Override
      public void checkSemantics() throws RecognitionException {
//          this.test.checkSemantics(); // check the if clause
//          super.checkSemantics();       // check the body
//
//          if (elseClause != null) {
//                elseClause.checkSemantics();
//          }
            super.checkSemantics();
      }


      public void addStatement(Block block) {
            this.elseClause = block;
      }
```

```
	public void addStatement(Statement stmt) {
		this.isLegalStatement(stmt);

		if (this.bodyBuilt == true) {
			this.elseClause = stmt;
		} else {
			super.addStatement(stmt);
		}
	}

	/* (non-Javadoc)
	 * @see gbang.statement.Statement#toJava()
	 */
	@Override
	public String toJava() {
		StringBuffer sb = new StringBuffer();
		sb.append("if (" + this.test.toJava() + ") {\n");
//		sb.append(this.statements.toJava());
		sb.append("}\n");
		return sb.toString();
	}

	/**
	 * @return the elseClause
	 */
	public Statement getElseClause() {
		return this.elseClause;
	}

	/**
	 * @param elseClause the elseClause to set
	 */
	public void setElseClause(Block elseClause) {
		this.elseClause = elseClause;
	}

	/**
	 * @return the test
	 */
	public Expression getTest() {
		return this.test;
	}

	/**
	 * @param test the test to set
	 */
	public void setTest(Expression test) {
		this.test = test;
	}

}


=======================================================
src/java/gbang/statement/KeyPressExpression.java
=======================================================
package gbang.statement;
```

```
public class KeyPressExpression extends Expression {

	protected String key;

	// TODO: Check the key to make sure its valid char sequence
	public KeyPressExpression(String key) {
		this.key = key;
	}

	public String getKey() {
		return key;
	}

	public void setKey(String key) {
		this.key = key;
	}

	public String toJava() {
		return "keyPressed(KeyEvent.VK_" + this.key.toUpperCase() +
")";
	}


}
```

```
=======================================================
src/java/gbang/statement/PropAssignment.java
=======================================================
/* FILE: PropAssignment.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.collections.AST;
import gbang.datatype.GTGEObject;
import gbang.datatype.GbScopeContainer;

/**
 *
 */
public class PropAssignment extends Assignment {

	protected GTGEObject leftVar;
	protected String propName;


	public PropAssignment(AST node, GbScopeContainer scope) {
		super(node);
//		this.lhs = new Expression(node);
//		this.rhs = new Expression(node.getNextSibling());
//		this.tree = node;

		this.scope = scope;
```

```
            // XXX: restricted to one->nesting of a property in
PropAssignment
            AST object = this.lhs.getTree().getFirstChild();
            this.leftVar = (GTGEObject)
this.scope.getVar(object.getText());
            this.propName = object.getNextSibling().getText();
            // TODO: Check semantics of sett call to GTGE object
      }

      public void checkSemantics() {
            System.out.println("prop assign");
      }

      public String toJava() {
            return leftVar.createJavaSetter(this.propName,
this.rhs.getTree()) + "\n";
      }
}


=======================================================
src/java/gbang/statement/Return.java
=======================================================
/* FILE: Return.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import gbang.antlr.GBangWalker;
import antlr.collections.AST;

/**
 *
 */
public class Return extends Statement {

      protected Statement retValue;

      public Return(AST node, GBangWalker w) {
            if (node != null) {
                  this.retValue =
Expression.getInstance(node.getFirstChild(), w);
            }
      }

      /* (non-Javadoc)
       * @see gbang.statement.Statement#toJava()
       */
      @Override
      public String toJava() {
            // TODO Auto-generated method stub
            return null;
      }

      /* (non-Javadoc)
```

```
	 * @see gbang.statement.Statement#checkSemantics()
	 */
	@Override
	public void checkSemantics() {
		// TODO Auto-generated method stub

	}

}
```

```
=======================================================
src/java/gbang/statement/Statement.java
=======================================================
/* FILE: Statement.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import gbang.antlr.GBangSemanticWalker2;
import antlr.RecognitionException;

/**
 *
 */
public abstract class Statement {

	/**
	 * Flag to indicate whether or not this statement has been
transformed to Java syntax
	 * by the Translator object
	 */
	protected boolean xformed = false;

	public String toJava() {
		return "// Some Statement";
	}
	public static GBangSemanticWalker2 walker = new
GBangSemanticWalker2();

	public void setXformed(boolean r) {
		this.xformed = r;
	}


	// ____________________________________________ JAVA TRANSLATION METHODS

	// ____________________________________________________________________

	/**
	 * Returns the string that
	 */
	/**
	 * After the g! objects are built, this function
	 * is called to ensure that the semantics are kosher
```

```
	 */
	public abstract void checkSemantics() throws RecognitionException;


}


=======================================================
src/java/gbang/statement/When.java
=======================================================
/* FILE: When.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import gbang.antlr.GBangTokenTypes;
import gbang.antlr.GBangWalker;
import antlr.RecognitionException;
import antlr.collections.AST;

/**
 *
 */
public class When extends Block {

	protected Expression test;
	//protected Block statements;

	public When() {
		super();
	}

	public When(AST test, AST body, GBangWalker walker) {
		this();
		if (test.getType() == GBangTokenTypes.KEY_PRESS) {
			this.test = new KeyPressExpression(test.getText());
		} else {
			this.test = new Expression(test.getFirstChild());
		}

		this.test.setScope(this);

		this.setParentScope(walker.getCurrentScope());
		this.buildBody(body, walker);
	}

	@Override
	public void checkSemantics() throws RecognitionException {
		//this.test.checkSemantics(); // check the if clause
		super.checkSemantics();       // check the body
	}


	@Override
	public String toJava() {
```

```
            StringBuffer sb = new StringBuffer();
            sb.append("if (" + this.test.toJava() + ") {\n");
            //sb.append(this.statements.toJava());
            sb.append(super.toJava());
            sb.append("}\n");
            return sb.toString();
      }

      /**
       * @return the test
       */
      public Expression getTest() {
            return this.test;
      }

      /**
       * @param test the test to set
       */
      public void setTest(Expression test) {
            this.test = test;
      }

}


============================================================
src/java/gbang/statement/While.java
============================================================
/* FILE: While.java
 * $Id$
 *
 * The G! Language
 */
package gbang.statement;

import antlr.RecognitionException;
import antlr.collections.AST;
import gbang.antlr.GBangTokenTypes;
import gbang.antlr.GBangWalker;

/**
 *
 */
public class While extends Block {

      protected Expression test;
      //protected Block statements;

      public While() {
            super();
      }

      public While(AST test, AST body, GBangWalker walker) {
            this();
            if (test.getType() == GBangTokenTypes.KEY_PRESS) {
                  this.test = new KeyPressExpression(test.getText());
            } else {
```

```
                  this.test = new Expression(test.getFirstChild());
            }
            this.test.setScope(this);

            this.setParentScope(walker.getCurrentScope());
            this.buildBody(body, walker);
      }


      @Override
      public void checkSemantics() throws RecognitionException {
            super.checkSemantics();
//          this.test.checkSemantics(); // check continuation clause
for boolean
//          super.checkSemantics();       // check the body
      }

      @Override
      public String toJava() {
            StringBuffer sb = new StringBuffer();
            sb.append("while (" + this.test.toJava() + ") {\n");
            //sb.append(this.statements.toJava());
            sb.append("}\n");
            return sb.toString();
      }

}


=======================================================
src/java/gbang/template/GlobalTemplate.java
=======================================================
/* FILE: GlobalTemplate.java
 * $Id$
 *
 * The G! Language
 */
package gbang.template;

import java.util.ArrayList;
import java.util.HashMap;

import gbang.GbTranslator;
import gbang.datatype.GTGEObject;
import gbang.datatype.GbDataType;
import gbang.datatype.GbPlayField;
import gbang.datatype.GbSprite;
import gbang.statement.BangExpression;
import gbang.statement.Declaration;
import gbang.statement.Function;
import gbang.statement.Statement;
import gbang.statement.When;

/**
 *
 */
public class GlobalTemplate {
```

```
	/**
	 * Create string for imports and class header
	 * @return
	 */
	public static String header() {
		StringBuffer sb = new StringBuffer();
		// imports
		sb.append("import java.awt.*;\n"
				+ "import java.awt.event.*;\n"
				+ "import java.awt.image.*;\n"
				// GTGE
				+ "import com.golden.gamedev.*;\n"
				+ "import com.golden.gamedev.object.*;\n"
				+ "import
com.golden.gamedev.object.background.*;\n"
				+ "import com.golden.gamedev.object.sprite.*;\n"
				+ "import
com.golden.gamedev.object.collision.*;\n"
				+ "\n"
				);

		// class header
		sb.append("\n\n //
_____________________________________ Game Class\n");
		sb.append("public class GBangGame extends Game {\n");
		return sb.toString();
	}

	/**
	 * Work over the translator object, pick off its instance vars
and declare them
	 * for the output.
	 * @param trans
	 */
	public static String instanceVars(GbTranslator trans) {
		StringBuffer sb = new StringBuffer();
		sb.append("\n\n // _____________________________________
START: INSTANCE VARS\n");
		for (Statement stmt : trans.getStatements()) {
			if (stmt instanceof Declaration) {
				Declaration d = (Declaration) stmt;
				sb.append(d.getVar().javaDeclare());
				d.setXformed(true);
			}
		}

		// Include some default objects we may always use
		HashMap<String,String> defaults = new
HashMap<String,String>();
//		defaults.put("Background", "background");
		defaults.put("Timer", "timer");
		defaults.put("Timer", "printTimer");
		defaults.put("GameFont", "messageFont");

		sb.append("\n // _________________________________ default
instance vars\n");
```

```
            for (String key : defaults.keySet()) {
                  sb.append(key + "\t" + defaults.get(key) + ";\n");
            }

            sb.append("\n // _______________________________ collision
managers\n");
            int i=0;
            for (Statement stmt : trans.getStatements()) {
                  i++;
                  if (stmt instanceof When) {
                        When s = (When) stmt;
                        if (s.getTest() instanceof BangExpression) {
                              String rsprite =  ((BangExpression)
s.getTest()).getRightObject().getSister().getVarName();
                              String lsprite =  ((BangExpression)
s.getTest()).getLeftObject().getSister().getVarName();

                              String temp =
rsprite.toUpperCase()+lsprite.toUpperCase()+i + " " +
rsprite.toLowerCase()+lsprite.toLowerCase()+i + ";";
                              temp+="\n";
                              sb.append(temp);
                        }
                  }
            }



            sb.append("\n\n // ________________________________________
END: INSTANCE VARS\n");
            return sb.toString();
      }

      public static String footer(GbTranslator translator) {
            StringBuffer sb = new StringBuffer();
            GbPlayField pf = translator.getPlayField();

            sb.append(
                        "\n\n//
_____________________________________________ MAIN\n"
                        + "public static void main (String[] args) {\n"
                        + "GameLoader game = new GameLoader();\n"
                        + "game.setup(new GBangGame(), new Dimension("
                        + pf.getPropertyValue("height", "480") + ", "
                        + pf.getPropertyValue("width", "640") + "),
false);\n"
                        + "game.start();\n"
                        + "} // method main\n\n\n"
                        + "} // class GBangGame\n");
            return sb.toString();
      }

      /**
       * @param translator
       * @return
       */
      public static String initResources(GbTranslator translator) {
```

```
            StringBuffer sb = new StringBuffer();
            sb.append("// _____________________________________
INITIALIZE RESOURCES\n");

            sb.append("public void initResources() { \n");
            String pfname = translator.getPlayField().getVarName();
            // initialize instance vars and set them accordingly
            ArrayList<Statement> stmts = translator.getInitStatements();

            try {
                  for (GTGEObject var : translator.getGTGEVars()) {
                        sb.append(var.javaInstantiate());
                        // add the sprite's sprite group to the
playfield
                        if (var instanceof GbSprite) {

      sb.append(String.format("%s.addGroup(%s);\n",
                                          pfname, ((GbSprite)
var).getSister().getVarName()));
                        }
                  }

                  for (Statement stmt : stmts) {
                        sb.append(stmt.toJava());
                  }
            } catch (Exception e) {
                  System.out.println("wtf");
            }


      ////////////////////////////////////////////////////////
            // SETUP FOR PRINT METHOD
            // WE ALWAYS USE THE SAME FONT
            // -> any call to print will only last for 2.5 seconds
            // setup the font thing for writing
            sb.append("\n// _____________________ FOR PRINTING
_____________\n");
            sb.append("messageFont =
fontManager.getFont(getImage(\"resources/BitmapFont.png\"));\n");
            sb.append("printTimer = new Timer(2500);\n");


            sb.append("\n// _____________________ COLLISIONS
_____________\n");
            //
            //
            int i=0;
            for (Statement stmt : stmts) {
                  i++;
                  if (stmt instanceof When) {
                        When s = (When) stmt;
                        if (s.getTest() instanceof BangExpression) {
                              String rsprite =  ((BangExpression)
s.getTest()).getRightObject().getSister().getVarName();
                              String lsprite =  ((BangExpression)
s.getTest()).getLeftObject().getSister().getVarName();
```

```
                              String temp =
rsprite.toLowerCase()+lsprite.toLowerCase()+i + " = new " +
rsprite.toUpperCase()+lsprite.toUpperCase()+i + "(this);";
                              temp+="\n";


temp+=translator.getPlayField().getVarName()+".addCollisionGroup(" +
rsprite + ", " + lsprite + ", " +
rsprite.toLowerCase()+lsprite.toLowerCase()+i +");";
                              sb.append(temp);
                        }
                  }
            }

            //ending method
            sb.append("} // method initResources \n\n");
            return sb.toString();
      }

      /**
       * @param translator
       * @return
       */
      public static String updateMethod(GbTranslator translator) {
            StringBuffer sb = new StringBuffer();
            sb.append("// _____ UPDATE
METHOD\n");

            sb.append("public void update(long elapsedTime) { \n");
            sb.append(translator.getPlayField().getVarName() +
".update(elapsedTime);\n");

            for (String var :
translator.getGlobalScope().getTable().keySet()) {
                  GbDataType obj = translator.getVar(var);
                  if (obj instanceof GbSprite) {
                        sb.append(obj.getVarName() +
".update(elapsedTime);\n");

      sb.append(((GbSprite)obj).getSister().getVarName() +
".update(elapsedTime);\n");
                  }
            }

            // get the when statements from the translator here
            for (Statement stmt : translator.getUpdateStatements()) {
                  sb.append(stmt.toJava() + "\n");
            }

            sb.append("}\n\n");
            return sb.toString();
      }

      /**
       * @param translator
       * @return
       */
```

```
	public static String renderMethod(GbTranslator translator) {
		StringBuffer sb = new StringBuffer();
		String pfname = translator.getPlayField().getVarName();
		sb.append("// _____________________________________ RENDER
METHOD\n");

		sb.append("public void render(Graphics2D g) { \n");

		sb.append(String.format("%s.render(g);\n", pfname));


		for (String var :
translator.getGlobalScope().getTable().keySet()) {
			GbDataType obj = translator.getVar(var);
			if (obj instanceof GbSprite) {
				sb.append(obj.getVarName() + ".render(g);\n");

	sb.append(((GbSprite)obj).getSister().getVarName() +
".render(g);\n");
			}
		}


		sb.append("} // method update\n\n");

		return sb.toString();
	}

	public static String renderFunctions(GbTranslator translator) {
		StringBuffer sb = new StringBuffer();
		String pfname = translator.getPlayField().getVarName();

		sb.append("\n// _____________________________________
FUNCTIONS\n");

		for (String key :
translator.getFunctionTable().getFunctionTable().keySet()) {
			Function fnc = translator.getFunction(key);
			sb.append(fnc.toJava());
		}
		return sb.toString();
	}

}
```