G!
A programming language for 2D games
Language Reference Manual

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

COMS W4115: Programming Languages and Translators
Department of Computer Science
Columbia University

October 19, 2006

CONTENTS

1. INTRODUCTION

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.

2 Lexical Conventions

2.1 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.

2.2 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 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 */
```

2.3 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.

2.4 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.

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.)

2.5 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. Operators

G! includes the following 6 categories of operators:

1. ! operator
2. Arithmetic
3. Relational
4. Logical
5. Dot operator
6. Colon Operator
7. Function call operator, Assignment operator , The comma operator and the Array reference Operator

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

3.1 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 The @ Operator

The @ Operator is used in conjunction with the "Coordinate" datatype in G! which represents the x and y coordinates of a sprite on the 2-D playfield. The @ Operator is thus, 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 @

3.3 Arithmetic Operators

3.3.1 Unary Operators

Unary operators are operators that operate on a single operand. G! supports four unary operators: unary + and unary -, unary ++ and unary --. The result of the unary operator + is the value of the operand while the result of the unary – operator is the negative of its operand. The operand must be of the arithmetic type. The unary ++ and unary -- operators are used as prefix or postfix operators i.e. before or after the operand. The ++ operator essentially increments the value of the operand by 1 and the -- decrements the value of the operand by 1. The difference between postfix ++ and prefix ++ is that a prefix ++ increments the value of t he operand and then uses it in the expression whereas the
postfix ++, uses the value of the operand first and increments it later.

```
Examples of unary operators in G! are:
VerticalSpeed = +0.5
```

```
 //Sets vertical speed to 0.5 in the positive read upward
direction

HorizontalSpeed = -1
//Sets horizontal speed to 1 in the negative read left direction

VerticalSpeed2 = 1 + VerticalSpeed
//Results in VerticalSpeed2 increased to 1 + (0.5)) =+2.5,
assuming the above value

HorizontalSpeed2= HorizontalSpeed--  - 1
//Results in HorizontalSpeed2 being set to -2, and HorizontalSpeed
being reduced to -2, assuming the above values.
```

Expressions with unary operators associate from right to left except for the postfix ++ and postfix -- operators which have a left-to–right associativity

3.3.2 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 of multiplicative 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.

3.3.3 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

3.4 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

3.5 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.

3.6 The Dot Operator (.)

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

```
object.property_or_method
instancename.variable
```

object: An element of the game. This parameter is always to the left of the dot (.) operator.

property_or_method: The name of a property or method associated with an object. This parameter is always to the right of the dot (.) operator.

3.7 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
```

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

Operator	Example	Description/Meaning
( )	`F()`	Function call
=	`a = b`	a, after b is assigned to it
*=	`a*=b`	a equals a times b
/=	`a/=b`	a equals a divided by b
+=	`a+=b`	a equals sum of a and b
-=	`a-=b`	a equals difference of a and b
,	`e1,e2`	e2 is evaluated after e1
[]	`a[10]`	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 Postfix increment Postfix decrement
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 **OR**	Logical AND Logical OR
:	Colon
= *= /= **%=** += –=	Assignment Multiplication assignment Division assignment Modulus assignment Addition assignment Subtraction assignment
,	Comma

4 Functions

4.1 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 soudn 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);
```

4.2 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);
```

4.3 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

4.4 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.

5. Data Types

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

5.1 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*.

5.2 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.

6 Statements
6.1 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!

6.2.1 Empty Statement

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

Example:

```
     ;
```

6.2.2 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.

6.2.3 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";
}
```

6.3 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.

6.4 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.

7. ANTLR GRAMMAR

Note that the latest (and plain text) version of this grammar can be found here:

http://plt.sytes.net:8090/trac/browser/src/java/gbang/antlr/grammar.g

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

// a program is just a series of statements
program
	:	(statement | func_def)+
	;

func_def
	:	"func" type ID LPAREN! define_arg_list RPAREN! block
	;

// arguments/parameters in a function definition need to be able to be given
// default values to enable clean implementation of calling a function with
// keyword arguments once you start giving a variable a default value in the
// definition, the rest of the parameters in the list must have a default value
define_arg_list
	:	(type ID "=") => define_kwarg_list
	|	type var_name (COMMA! type var_name)* (define_kwarg_list)?
	|	/* empty */
	;

define_kwarg_list
	:	type ID "=" rh_assignment (COMMA! type ID "=" rh_assignment)*
	;

// list of arguments (used when calling a function)
// the parser leaves it to something down stream to ensure that
// keyword arguments are defined strictly after required arguments
// although this should ideally be done here!
arg_list
	:	ID (COLON atom)? (COMMA! ID (COLON atom)?)*
	|	/* empty argument list */
	;

expression
	:	assignment
	;

assignment
	:	var_name (ASSIGN | aug_assign_op) rh_assignment
	;

aug_assign_op
	:	PLUS_ASSIGN
```

```
       |     MINUS_ASSIGN
       |     STAR_ASSIGN
       |     DIV_ASSIGN
       ;

aug_op
       : DEC | INC
       ;

rh_assignment
       :     term
       ;

// Note to self:    there is a trailing semi here -- this may cause problems
//                                 if you start changing the "statement" definition
declaration
       :     type var_name (COMMA var_name)* SEMI!
       ;

if_statement
       :     "if"^ LPAREN! test RPAREN! (statement | block)
             (      // combat the dangling else -- inspired by the java1.5 grammar
                    options { warnWhenFollowAmbig = false; }:
                    "else"! (statement | block)
             )?
       ;

when_statement
       :     "when"^ LPAREN! test RPAREN! (statement | block)
       ;

for_statement
       :     "for"^ (type)? var_name ASSIGN NUMBER COLON NUMBER (COLON NUMBER)?
(statement | block)
       ;

while_statement
       :     "while"^ (LPAREN)? test (RPAREN)? (statement | block)
       ;

include_statement
       : "include" (rh_assignment) SEMI!
       ;

break_statement
       :     "break" SEMI!
       ;

continue_statement
       :     "continue" SEMI!
       ;

return_statement
       :     "return" (rh_assignment)? SEMI!
       ;

print_statement
```

```
        :      "print" rh_assignment SEMI!
        ;

statement
        :      declaration
        |      assignment SEMI!
        |      atom SEMI!    // handles normal and chained (name.name.func()) calls
        |      if_statement
        |      when_statement
        |      for_statement
        |      while_statement
        |      include_statement
        |      break_statement
        |      continue_statement
        |      return_statement
        |      print_statement
        ;

// conditional test defined by operation precedence
test: and_test ("OR" and_test)*
        ;

and_test
        : not_test ("AND" not_test)*
        ;

not_test
        : NOT not_test
        | comparison
        ;

// compare with traditional "<,>,<=" operators and
// the BANG (!left, !right) operators
comparison
        :      term (comp_op term)?
        |      var_name (bang_op) var_name
        |      key_press_check
        ;

comp_op
        :      LT
        |      GT
        |      EQUAL
        |      GEQ
        |      LEQ
        |      NOTEQUAL
        ;

// binary (boolean) operator
// LHS and RHS must be sprites
//     Example:     srite1 !left sprite2
bang_op
        :      BANG
               (      "top"         { /* sprite on LHS hits the top of sprite on RHS */ }
               |      "right"       { /* sprite on LHS hits the right of  RHS */ }
               |      "bottom"      { /* sprite on LHS hits the bot. of sprite on RHS */ }
               |      "left"        { /* sprite on LHS hits the left of sprite on RHS */ }
```

```
             |      /* just BANG: sprite on LHS hits any part of sprite on RHS*/
             )
      ;

// Unary operator, LHS must be a sprite, returns a Coordinate object
at_op
      :      AT
             (      "top"         { /* top-middle coords of sprite on LHS */ }
             |      "right"       { /* right-middle coords of sprite on LHS */ }
             |      "bottom"      { /* bottom-middle coords of sprite on LHS */ }
             |      "left"        { /* left-middle coords of sprite on LHS */ }
             |      "center"      { /* center coords coords of sprite on LHS */ }
             )
      ;

// the most unit of all things
term:  factor ((STAR | SLASH | PERCENT) factor)*
      ;

factor
      :      (PLUS | MINUS) factor
      |      power
      ;

power
      :      atom (CARAT factor)?
      ;

atom:  var_name  (trailer)?
      |      truth
      |      INT_NUM
      |      DOUBLE_NUM
      |      STRING
      |      NULL
      ;

trailer
      :      (options {greedy = true;}: DOT var_name)+ (trailer)? // name.name.name ...
      |      LPAREN (arg_list) RPAREN   // embedded function call
      |      (array_index)+
      |      at_op
      |      (DEC | INC)
      ;

array_index
      :      LBRACK INTEGER RBRACK
      ;

var_name
      : ID
      ;

truth: TRUE
      |      FALSE
      ;
```

```
type:  INTEGER_TYPE
       |      DOUBLE_TYPE
       |      STRING_TYPE
       |      BOOLEAN_TYPE
       |      SPRITE_TYPE
       |      SPRITE_GROUP_TYPE
       |      SOUND_TYPE
       |      PLAYFIELD_TYPE
       |      COORD_TYPE
       |      VOID_TYPE
       ;

// a multiline block of code.
block
       :      LCURLY! (statement)* RCURLY!
       ;

key_press_check
       :      "KeyPress" DOT STRING
       ;

// LEXER
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;
}

tokens {
       INT_NUM; DOUBLE_NUM;
}

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

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

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

protected
```

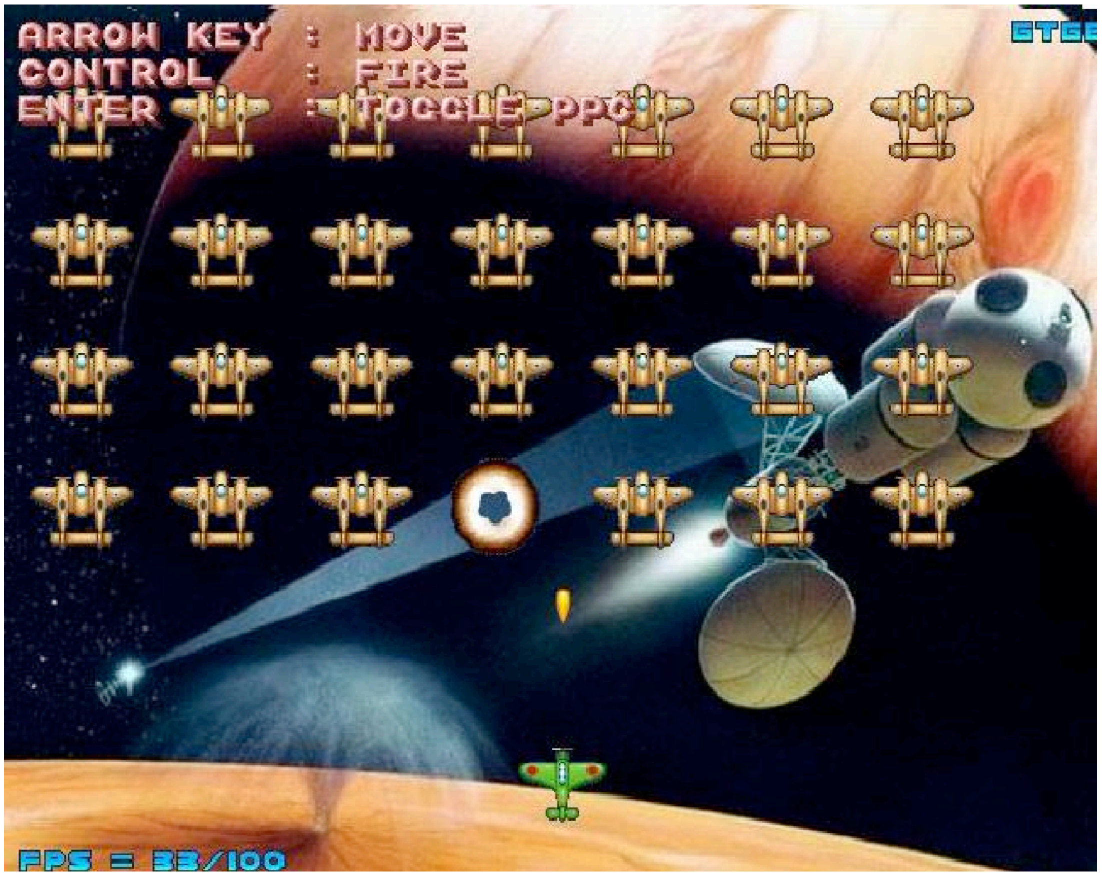
```
DoubleTrailer
        :       '.' (Integer)*
        ;


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
       :      '"'! ('"' '"'! | ~('"'))* '"'!
       ;


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
       ;

// types
INTEGER_TYPE: "Integer";
DOUBLE_TYPE: "Double";
STRING_TYPE: "String";
BOOLEAN_TYPE: "Boolean";
SPRITE_TYPE: "Sprite";
SPRITE_GROUP_TYPE:"SpriteGroup";
SOUND_TYPE: "Sound";
PLAYFIELD_TYPE: "Playfield";
COORD_TYPE: "Coordinate";
VOID_TYPE: "void";

TRUE: "true";
FALSE: "false";

// Operators
NOT: "NOT";
LPAREN: '(';
RPAREN: ')';
LBRACK: '[';
RBRACK: ']';
LCURLY: '{';
RCURLY: '}';
COLON: ':';
COMMA: ',';
DOT: '.';
ASSIGN: '=';
EQUAL: "==";
SLASH: '/';
PLUS: '+';
MINUS: '-';
STAR: '*';
PERCENT: '%';
CARAT: '^';

// AUGMENTING ASSIGNMENT
DIV_ASSIGN: "/=";
PLUS_ASSIGN: "+=";
```

```
INC: "++";
MINUS_ASSIGN: "-=";
DEC: "--";
STAR_ASSIGN: "*=";

// COMPARISON
LEQ: "<=";
LT: '<';
GEQ: ">=";
GT: ">";
BANG: '!'; // !left, !right handled by parser

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

9. Sample G! Program

```
//This code creates a game where an airplane shoots bullets
// at a block of 28 enemies. A screen shot is attached after the code
// so you get n idea of what is happenning here.

Playfield playfield;

Sprite airplane, bullet, enemy[28];
SpriteGroup badguys;

playfield.properties(background:"bgimage_path", height: 500, width: 500);

// height and width of the airplane sprite are automatically detected
// by G!, but they can be overidden to shrink/grow the image
airplane.properties(image: "image_path", height: 300, width: 150, posx: 235,
posy: 150, group: "hero", animate: true, loopanim: true, <parent:playfield>);

bullet.properties(image: "image_path", group: "projectile", animate: true;
loopanim: false);

// make a grid of enemies and add them to the badguy group
for i=0:3 {
      for j=0: 6 {
            enemy[(2*i)+j].properties(image: "image_path", group: badguys,
posx:i*80, posy:j*70);
      }
 }

// a little asynchronisity!

when(bullet !bottom enemy) { shot_enemy(with: bullet, which: enemy); }
when(KeyPress."SPACE") { shoot(from: airplane@top, what: bullet); }
when(KeyPress."LEFT") { move_left(airplane); }
when(KeyPress."RIGHT") {move_right(airplane); }

// now some function definitions
func void shot_enemy(Sprite with, Sprite enemy) {
    Sprite explosion;
    // make the explosion an animated sprite that only plays once
    // it will appear at the center of the enemy
    explosion.properties(image: "image_path", coords: enemy@center, timeout:
100, animate: true, loopanim: false);
    enemy.properties(active:false); // makes sprite disappear
    explosion.play();
}

func void move_left(Sprite airplane) {
    if (airplane@left < playfield@left) {
        // airplane is at edge of board, can't move left
        airplane.setHorizontalSpeed(0);
    } else {
        Coordinate position;
        position = airplane.coords;
        position.x -= 5
        airplane.coords = position;
    }
}

func void move_right(Sprite airplane) {
    if(airplane@right < playfield@right) {
        airplane.coords.x += 5;
    } else {
        airplane.setHorizontalSpeed(0);
    }
}

func void shoot(Coordinate from, Sprite what) {
    // sets the start coordinates of the bullet and an initialization
    // vertical speed. This sprite will continue to move up until
    // it collides with an enemy, or it flies off of the screen
    what.properties(coords: from, vspeed: 1);
    what.play();
}
```

The above is a the code that would create a game that looks like the image attached.


ARROW KEY : MOVE
CONTROL : FIRE
ENTER : TOGGLE PPC
FPS = 33/100