MaRVN

Multi-threading and Recursive Variable Notation

Language Specification

Version 0.42

John Dunham

1. Introduction

An utterly dreadful amalgamation of C# and Erlang created by some Computer Science Student who thinks he's far wittier than he actually is MaRVN (pronounced Marvin) is an imperative language to lapse into endless recursion bemoaning the fact that such a gloriously powerful language is used to interface with humans (of all things).MaRVN stands for Multi-threading and Recursive Variable Notation. Although MaRVN is predominately based on C# it does differ from the standard procedure a bit in a few major ways, particularly revolving around the syntax and how the language handles recursion and threading.

1. All Strings are lists why people have been using anything else just boggles MaRVN's compiler. For those who can't wrap their minds around list based Strings the data type StringArray exists, although the MaRVN compiler will throw a snarky warning in anything but an improbability field.

2. The only allowed form of recursion is tail end; if you want to do any other form of recursion it must be done in a improbability field block.

3. C#'s unsafe code blocks are now called Improbability fields as some wild stuff can happen in them.

4. All statements **should** be worded in the form of complaints or be incredibly sarcastic, recursion in particular which, being tail end, is laid out as iteration with every complaint slavishly executed by the compiler. This, however, isn't enforced but strongly recommended by the compiler through incredibly sarcastic warnings.

5. Programs are called superRants and subprograms are called rants.

6. Fields are declared with field blocks as show in the section on fields. They make for easier more organized code.

1.1 Hello world

```
Using brainSizePlanet;
//I know this may not seem like a complaint but it's certainly sarcastic
Class HelloWorld{
        static fruitlessRant beginRant(){
                console.sigh("Hello World");
                endRant;
        }
}
```

1.2 Program structure

As the compiler is wickedly Sardonic MaRVN is a little harder to do any code in, but if the compilers somewhat arbitrary rule are met the language can be incredibly powerful.

1. Entry occurs at beginRant in every superRant (although with .dlls it is possible to not use a beginRant).

2. Program blocks and statements are delimited by curly braces {} to indicate scoping.

3. To end a rant one must use an endRant. If the return type of the rant is fruitlessRant endRant; will suffice, for programs with a return type endRant appropriateVariable; is the standard.

4. endRant may also be used in block statements to return the values to an ordinal set of arguments to an array as show in later examples.

```
using brainSizePlanet;

class OhAnotherExample{
	static fruitlessRant beginRant(){
		int x = 7;

		//sighln is MaRVN's println
		console.sighln(square(x) – 7);
		fruitlessRant();
		endRant;
	}
	public int square(int x){
		endRant x *x;
	}

	public fruitlessRant leftSideDiodeCheck(){
		Console.sighln("The diodes down my left side still hurt, no need to check.");
		endRant;
	}
}
```

Just a very basic example of how the program would pass variables to a basic rant and how a fruitless rant is called.

1.3 Types and variables

In MaRVN there predominately exist **earthling** and **robot** types of variables. While both types are exceedingly dismal the earthling types include data that exists on the stack, meaning

floats, ints, and chars. The robot types point to collections of data that exist on the heap, meaning the complex types such as Objects, Lists and Strings, and the actual variables exist on the Stack and contain the memory address of the complex types. In Improbability Fields a third type exists, the pointers which can be used to point to any data type, even other pointers, for the purposes of the language robot types encapsulate pointers.

1.4 Statements Differing from C# and Erlang

Statement	Example	Notes

improbabilityField (args){
…
endRant args;
}
Or
improbabilityField (){
…
endRant;
}
	String [] args = new Array [3];
	improbabilityField(args){
	int a = 42;
	int b = &a;
	int* p = b;
	int c = *p;
	endRant a,b,c;
	}
	>> args[0] = 42 ,args[1] = some address, args[3] = 42
		Please note that args represents pass by result parameters, so data may be returned from the block in an ordinal endRant. (replaces unsafe{}) args is optional. *note: the args is there to enforce good style, it is NOT actually necessary in all cases as the improbability field is statically scoped and has all the variable from the "parent" scope.

MaRVN(args){
…
endRant;
}
Or
MaRVN(){
…
endRant;
}
	String [] args = new Array [2];
	MaRVN (args){
	Thread t = new Thread(ClassName::getX);
	Thread b = new Thread(getY);
	t.Start(args[0]);
	b.Start(args[1]);
	endRant;
	}
		The all-important MaRVN block. The MaRVN block is basically a thread that carves out a bit of the stack and heap to allocate to the threads it creates. The start function in MaRVN is overloaded to hold a variable that will be used for pass-by-result assignment that will then be passed to the original variables in the argument list and usable in the code. endRant is only called once all created threads have either timed out or executed, additionally endRant t,b; will kill the block and threads at once. Note the scope resolution operator used to specify a rant in a class. Args is optional. *note: the args is there to enforce good style,

		it is NOT actually necessary in all cases as the MaRVN Block is statically scoped and has all the variable from the "parent" scope.

fieldType{…};	int{
	x,y,z;
	a = 2;
	};
		FieldType may be one of 3 tiers.
		1.Static
		2.Public,Protected, Private
		3.Variable type
		When nesting the following hierarchy is used 1->2->3 and there may be no duplicate 2s in a 1 scope or duplicate 3s in a 2.

1.5 Classes and objects

As MaRVN is born of two languages with support OOP classes and objects are quite logically present.

```
class Point{
        private{
                int x,y;
        };
        public{ };
        protected{ };


        public Point(int x = 0, int y = 0 ){
                this.x = x;
                this.y = y;
                endRant;
        }
        public void setX (int x){
                this.x = x;
                endRant;
        }
        public void setY (int y){
                this.y = y;
                endRant;
        }
        public String toString(){
                endRant "(" + x + "," + y + ")";
        }
```

}

As show above default values are allowed in the constructors of MaRVN classes, for the most part they function as those in C++ and C#(I was going back through my language when I realized C# actually has these too, which I was not expecting) with ordinal importance and are evaluated left to right (for instance Point(), Point(x) and Point(x,y) are all valid calls, but Point(y) is not). Additionally, there is the presence of the **public**, **private**, and **protected** blocks for variables to make code more readable and variable specifications more ordered (the languages does support private int varname; and similar declarations but MaRVN will throw a warning). These blocks if empty don't need to be there. Additionally there is a fourth block **static** which can enclose the other three adding the static modifier to them. Note the semicolon at the end of the blocks this is mainly to enforce that these code blocks are not rants or scopes (hence the lack of endRant) additionally note that the class doesn't have an endRant (as they aren't subRants).

To create a new object the new operator must be used.

```
Point pointObj  = new Point();
```

Additionally the MaRVNClass modifier may be added to the class header as follows:

```
class MaRVNClass Point{…}
```

The compiler will statically check for any signs of variable reassignment in the class and throw errors at compile time. If this modifier is added to the class header all rants are considered to be MaRVNRants.

For inheritance the extends keyword is used:

```
Class Box extends Point{ }
```

If the parent class wasn't a MaRVNClass the child may not be made a MaRVNClass, but the child doesn't need to be a MaRVNClass if the parent was.

1.5.1 Accessibility

Accessibility	Meaning
Public	No limitations on access.
Protected	Only classes of the inheritance hierarchy may access.
Private	Access is limited to the Class
MaRVNRant	May only be accessed from within a MaRVN block, data accessibility may be further altered as follows (in regards to the MaRVN block): public MaRVNRant, private MaRVNRant, protected MaRVNRant. The default accessibility is public MaRVNRant.
MaRVNClass	The only types of classes that may be created in a MaRVN block. Ex. class MaRVNClass Point{…}

1.5.2 Fields

MaRVN fields are the variables associated with the class. As mentioned before they are declared in blocks with the accessibility modifier as shown below.

```
public{
```

```
        int j;
};
private{
        String x;
};
protected{
        Boolean zed;
};
static{
        public{
                double  j;
        }
        private{
                Point p;
        }
        protected{
                char [] z;
        }
};
```

Note that in the static declaration the public, private and protected blocks don't need semicolons at the end of the braces. Fields may also be declared without these blocks, however this methodology is preferred. Additionally, if a large number of one type exists the following declaration is valid:

```
        int{
                j,k,l;
                z  =10;
                x =30;
        };
```

Variables declared with no value may be comma delimited followed by a semi-colon and variables to be initialized may be initialized as their own statement.

1.5.3 Methods

In MaRVN methods are called **rants** (as they are subprograms to the superRant) and they are the implementation of code that may be called by another part of the program dependent upon the accessibility modifier. Rants may use the public, private, protected and MaRVNRant accessibility modifiers. Additionally, rants must have an endRant regardless of the return type specified in the **signature**.

Each class may only have one rant with the same signature, overloading a rant is allowed as the full signature changes.

1.5.3.1 Constructors

MaRVN supports both **instance** and **static constructors**. Instance constructor initializes the data for a single instance of the object and is formatted as such:

```
        public Point(){…}
```

Static constructors are also allowed and are activated as soon as an instance of the class is created. Static constructors are less preferred as they can sometimes make code harder to read and take some control of the program from the programmer but no warning is thrown when attempting to use them as they do have their uses.

static Point(){…}

1.5.3.2 Properties

Properties in MaRVN are a somewhat more static variation of fields (in the sense that they are nonvolatile). A MaRVN rant may use properties to store data needed to restart the rant after it and its object have be destroyed. The access is done through an access() rant specified in the Properties class that facilitates the use of properties. Writes are done with write(whatHasToBeAdded). Both are invoked by an instance of Properties. Properties files are independent from the program and must be loaded or created, if a create is called and the properties file already exists an exception will be thrown.

1.5.3.3 Events

In MaRVN events are called **mishaps**. To MaRVN, regardless of the value of the event, they are all a mistake, hence mishap. Mishaps are handled by **GPPs**, short for General Phenomenon Proxies (or Genuine People Personality chose your poison), and are affixed with :+ and removed with :- .

```
using brainSizePlanet;

class OhAnotherExample{
	static fruitlessRant beginRant(){
		String agrajagName = “Surprised Whale”;
		Properties agrajagNameFile = new Properties(“agrajagNames.txt”);
		int maxNameChenges = agrajagNameFile.load(“max”);
		int numNameChanges = 0;

		agrajagName.Changed :+ GPP(ohBoy);
		while(42 == brainSizePlanet.AQLUE){
			if(Math.random() % 1000 == 42){
				agrajagName = agrajagNameFile.load((String)numNameChanges);
				numNameChanges ++;
			if(numNameChanges > maxNameChanges)
				numNameChanges =0;
			}
			endRant;
	}
	public fruitlessRant ohBoy(){
		Console.sighln(“ARTHUR DENT!”);
		endRant;
	}
}
```

Some notes, Math.random() in MaRVN picks a random number from 0 to the max value of unsigned long ($2^{64} - 1$)explaining the use of % which is the standard modulo. The constructor of Properties has a load built in. 42 == brainSizePlanet.AQLUE is equivalent to true as brainSizePlanet.AQLUE is a constant from brainSizePlanet equal to 42 (AQLUE stands for Answer to the Question of Life, the Universe and Everything which may or may not change). True would suffice for an infinite loop as well.

1.6 Arrays

In MaRVN **arrays** aren't particularly emphasized or altered, drawing their base syntax from C#. The use of arrays is strongly discouraged with recursion and parallel programming in MaRVN and will throw warnings in MaRVN blocks, a list, queue or stack structure is preferred in MaRVN blocks if a collection of objects is needed.

Example of Array declarations:
int [] anArray = new Array[6]; //an array of integers length 6
int [] anotherArray = {1,2,3,4,5,6} //an array of integers length 6
int [,] enoughArrays = new Array [2,3] //multidimensional array
int [][] jadedWithArrays = new Array [2] [] //the second and/or first array may be left blank and both may have length specifications

2. Lexical structure

2.1 Programs

In MaRVN superRants are composed of one or more source files one of which should have a beginRant method to allow the superRant to execute (except if one were to use .dll). A source file is composed of characters of the Unicode variety as shown in all of the previous examples.

The compilation process is as follows.
1. The compiler converts the source file to a standardized Unicode character set so it may be properly analyzed.
2. The converted source is then sent through the lexical analyzer to convert the Unicode character stream to lex tokens.
3. The lex tokens are then taken and passed through the syntactical parser which creates the concrete syntax tree which may then be checked for semantics and ultimately become usable executable code.

2.2 Grammars

This specification presents the syntax of the MaRVN programming language where it differs from C# and Erlang.

2.2.1 Lexical grammar where different from C# and Erlang

identifier : (doesn't equal *keywords*)
    \b[a-zA-Z \u005F]+\b
literal :
    integer-literal
    real-literal
    character-literal
    string-literal
operators:
    operator-or-punctuator
fieldType:
    static
    private
    protected
    public
    VariableTypes
VariableTypes:
    Earthlings
    Robots

2.2.2 Syntactic ("parse") grammar where different from C# and Erlang

MaRVN-block:
MaRVN (*String-array*) { *statementList* endRant ; }
MaRVN () { *statementList* endRant ; }
improbability-Field:
improbabilityField (*String-array*) { *statementList* endRant *String-array*; }
improbabilityField () { *statementList* endRant; }
fieldBlock:
fieldType { *statementList*};
fieldType { *fieldBlock* };
fieldBlockList:
FieldBlock ;
FieldBlock, FieldBlockList ;
statement :
variableType *statement*
identifier = literal ;
identifier = identifier ;
identifier = statement
literal operator statement
identifier operator statement
literal ;
identifier ;
{ *statementList* }
{ *statementList* };
(there are LOADS of these, this is but a few)

statementList:
statement statementList
statement
String-array:
String-Array-identifier
(*note this indicates that the string array is a particular type of robot that has been defined before in C# and the exercise to define it is a bit overboard)

2.2.3 Grammar notation

The lexical and syntactic grammars are presented using **BNF *grammar productions***. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of nonterminal or terminal symbols. In grammar

productions, *non-terminal* symbols are shown in italic type, and terminal symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

while-statement:

while (*Boolean-expression*) *statement*

defines a *while-statement* to consist of the token while, followed by the token "(", followed by a *Boolean-expression*, followed by the token ")", followed by a *statement* (a statement is delimited by {} or a ; .

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separatelines. For example, the production:

statement-list:

statement

statement-list statement

defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.

2.3 Lexical analysis

2.3.1 Line terminators

Line terminators divide the characters of a MaRVN source file into lines.

new-line:

Carriage return character (U+000D)
Line feed character (U+000A)
Carriage return character (U+000D) followed by line feed character (U+000A)
Next line character (U+0085)
Line separator character (U+2028)
Paragraph separator character (U+2029)

2.3.2 Comments

Identical to C#, // indicates single line comments and /*…*/ deliminates multiline comments.

2.3.3 White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

whitespace:

Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)

Form feed character (U+000C)

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

token:

identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

tokens specific to MaRVN:

keywords:

MaRVNRant, MaRVN, MaRVNClass, extends, fruitlessRant, endRant, improbabilityField, beginRant, as well as all keywords originally featured in C# except for goto, void, return and unsafe.

Operators:

:- :+ ~ ::

2.4.1 Keywords different from Erlang or C#

New keywords:

New keyword	Why
MaRVNRant	The most vital element of MaRVN. A MaRVNRant specifies a class/rant that may only be accessed from a MaRVN block. A MaRVNRant is a rant optimized for parallel programming, meaning no variable reassignment, tail-end recursion and the like.
MaRVN	Used for MaRVN blocks (MaRVN(some pass by result variables){some parallel code}) which partition a portion of the stack and heap for the code to prevent collisions with non-MaRVN blocks. Additionally variables can be returned from the MaRVN block using an ordinal endRant. For example endRant aList, aStack; would do the following assignments for MaRVN(globalList, globalStack){} globalList = aList and globalStack = aStack
MaRVNClass	Specifies a class that may be instantiated in a MaRVN block. No variables may be reassigned. Basically a specification of the static keyword.
extends	Used in inheritance.

Removed keywords:

Removed	Why
Goto	It's not a keyword anymore, now just trying to compile a program with a goto shuts down your

	computer and changes your desktop background to a look of disapproval (MaRVN doesn't play around, but let's leave that to the person implementing it).

Modified keywords:

Original keyword	New keyword	Why
Void	fruitlessRant	As subprograms are rants this enforces that the rant returned nothing(it was fruitless)
Return	endRant	endRant is more accurate to describe how the return works in this language. While brackets are still there all rants require this to end rant.
Unsafe	improbabilityField	Crazy stuff can happen in unsafe code blocks, one time a thermonuclear warhead was transformed into a bowl of petunias and a program got overwritten by its own data.
main	beginRant	Programs are called super rants, it only makes sense that the entry point begins them.

3. Basic concepts

3.1 Application Startup

In MaRVN **application startup** is through the beginRant rant using either of the following signatures:

```
Static fruitlessRant beginRant(){…endRant;}
Static fruitlessRant beginRant(String [] args){…endRant;}
```

As shown above the application startup(entry point for the application) may be called with arguments that can be used to initialize aspects of the code or not. This String [] can be particularly useful in testing code.

Note that MaRVN has no int return type for the application startup. This is due to the fact that MaRVN considers all applications ultimately fruitless and as such all superRants are in fact fruitlessRants.

3.2 Application termination

In MaRVN **application termination** occurs at the beginRant rant's endRant. As the entry point of the application in MaRVN is analogous to void by default the **termination status** of the application will return 0 indicating that the application ran successfully and rendered control back to whatever execution environment ran the application (In MaRVN generally the CLR). If nothing is returned it is entirely likely that something is fatally wrong with your code and MaRVN will take solace in the fact that that you can't mess anything else up or ask it to add 2 and 2 with your locked down computer.

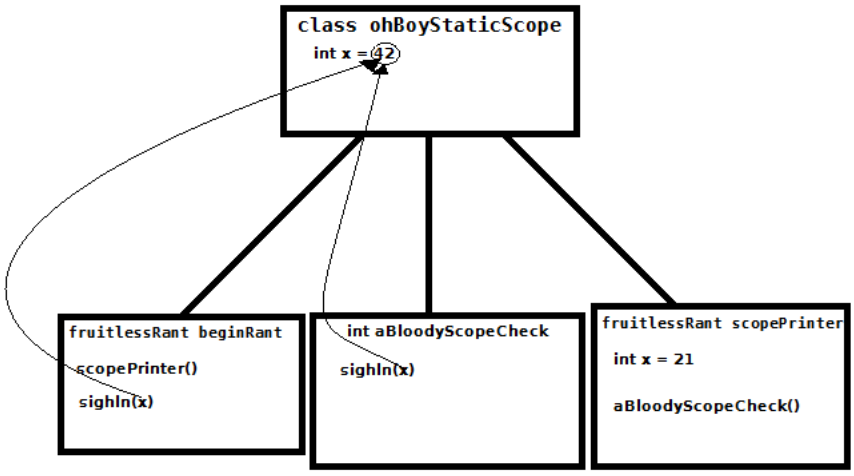
3.3 Scope

MaRVN's scope is static, predominately because dynamic scope can end up with hard to determine errors.

```
class ohBoyStaticScope{
	int x = 42;

	public int aBloodyScopeCheck(){
		return x;
	}
	Public fruitlessRant scopePrinter(){
		int x = 21;
		Console.sighln(aBloodyScopeCheck());
	}
	static fruitlessRant beginRant(){
		scopePrinter();
		console.sighln(x);
		endRant;
	}

}
```



As shown above the scoping of MaRVN is staunchly static. The value of x in beginRant and aBloodyScopeCheck is pulled from the scope that encloses the scope of the rant that needs it. As a result this program will print 42 twice, were the scope dynamic it would print 21 (newline) 42.

3.4 Automatic memory management

MaRVN uses **garbage collection** to handle memory management with a mark and delete algorithm (meaning the robots to be deleted will be marked then deleted). The garbage collection is automatic, but a request for collection may be made of the garbage collector with the call brainSizePlanet.pickUpThatPaper();

Additionally, programmer defined memory management is possible with the use of a destructor similar to C++:

~Point(){…endRant;}

Once the robot goes out of scope (as only robots may use a destructor as they "live" on the heap) the destructor will kill off any earthlings and execute any code that handles fields that live on the heap. This is advantageous when dealing with pointers as the object will destroy itself when no longer needed not caring about references. Additionally, the scope being referred to is that of the scope that the instance of the object was made in.

The preprocessor directive:

Using noGarbageCollection;

This must be used in any class that will not use garabage collection, if used in the entry point class it will disable the Garbage Collection for the execution NOT RECOMMENDED.

This memory management technique is dangerous and only to be used by the most advanced of users, but it exists for the sake of doing more complex jobs with MaRVN.

4. Types

In MaRVN Earthling and Robot types are the two existing types. For MaRVN the type checking is via name type equivalence

4.1 Earthling types

Type	Size(bytes)	Valid values	Example declaration
int	4	-2 billion to 2 billion	int x =42;
long	8	-9 quintillion to 9 quintillion	long zed = 2112;
short	2	-32,767 to 32,767	short q = 700;
byte	1	0 to 255	byte b = 300;
double	8	15 sig figs	double lue = .042;
float	4	7 sig figs	float f = 1.01;
decimal	24	28 to 29 sig figs	decimal d = -1.222;
char	2	Unicode characters	char c = ‘w’;
bool	2	true, false	bool isTrue = true;

The above examples are identical to those specified in C# as they trump the Erlang primitives when it comes to ease of use for the programmer and they afford a lot more write ability in the code. Additionally the signed and unsigned key words may be used to allow for negatives in the variable or not. int, long, short, double, decimal, and float are all implicitly signed, while byte is implicitly unsigned.

4.2 Robot types

As Robot types include objects, Strings and arrays that exist in the heap with an address in the stack the memory allocation for the stack is a 4 byte memory address (as it is in C#). For the heap the limit is literally the memory of the system it is running on. One creates a Robot as follows:

Point p = new Point();

The variable p holds the memory address of the object itself while the new operator carves out the needed space in memory for the Robot. These variables are similar to pointers, however, p = &anotherPoint will not work outside of an improbabilityField block. For all intents and purposes Robot types are pointers and they are allowed to behave as such within improbabilityField blocks, similarly to C# and unsafe blocks.

4. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. MaRVN is a strongly typed language by default the variables are mutable, but in MaRVN blocks all variables are non-mutable.

5.1 Variable examples & 5.2 Variable categories

In MaRVN there are a great deal of potential variables. As mentioned before there are the Earthling types (which all have an example in the previous section under Earthling types). Some standardized variable types that exist for robot types include the following:

Pointers, not really a robot type, as in MaRVN they are allocated on the stack, but anything on the heap has at least one pointer:

```
//creates an int pointer
int *p;
int x = 7;
//sets the pointer equal to the address of x
p = &x;
//will print the contents of x
Console.sighln(*p);
```

Strings, exist on the heap in the form of a list, and can be declared verbosely with the new constructor or with just the double quotes.

```
String helloDullards = new String(“hello”);
String goodbyeDullards = “So Long and Thanks for all the Fish”;
```

StringArrays Syntaxtically identical to Strings, but exist for those who are unable to comprehend the glory of List based Strings, or need an array based string for whatever reason.

```
StringArray helloDullards = new String(“hello”);
StringArray goodbyeDullards = “So Long and Thanks for all the Fish”;
```

Arrays, in MaRVN pure arrays are NOT dynamically allocated, that’s what lists, queues and stacks are for:

```
int [] anArray = new Array[6];
int [] anotherArray = {1,2,3,4,5,6};
```

5.3 Activation Record examples

Consult parameter passing’s activation record.

5. Parameter Passing

6.1 Method

MaRVN uses In(by value or reference), In-Out(by reference) and in the case of MaRVN blocks, Improbability fields and threads, pass by result. For all of the cases the In and Out Reads/writes are left to right.

6.2 Examples

In Parameter passing is the default parameter passing technique of MaRVN. In parameter passing in MaRVN is done by copy in which the values of the actual parameters are passed into the formal parameters. If Robot Types are passed they are passed by reference by default.

Example call:

```
foo(12,31);
```

Function example:

```
public fruitlessRant foo (int x, int y){
        Console.sighln(x*y);
        endRant;
}
```

Activation record:

Code	beginRant	…foo(12,31);…
	foo	Console.sighln(x*y); endRant;
Data	beginRant	Local Params: null
	foo	Local Params: null Parameters: int x=12 int y=31 return address: "&beginRant"

In-Out Parameter passing in MaRVN is identical to C# using the ref key word to indicate that a variable is passed by reference (the only way In-Out is allowed in MaRVN) indicating that the formal parameters are aliases.

Example call:

```
int x = 7;
int z = 4;
foo(ref x,z);
```

Function example:

```
Public fruitlessRant foo(int y, int q){
        y = y*q;
        q++;
        endRant;
```

}

Code		
	beginRant	…int x = 7; int z = 4; foo(ref x,z);…

	foo	y = y*q;
		q++;
		endRant;

Data		
	beginRant	Local Params: int x =7 int z =4
	foo	Local Params: null Parameters: int y= &x int q= z return address: "&beginRant"

Pass by result occurs in MaRVN blocks, improbabilityFields and Threads. Pass by result in MaRVN uses aliases indicating that all pass by result calls are implicit and only occur in the following signatures {MaRVN(args){… endRant;}; , improbabilityField(args){… endRant args;}; , threadID.start(arg);}. The first two of these for the passing mechanism is more or less to enforce what values will be changed, although the variables are generally in scope as it stands.

Example:

```
String [] args = new Array[3];
improbabilityField(args){
        int h = 12;
        int y =16;
        int z =17;
        endRant h,y,z;
};
```

Code		
	beginRant	… String [] args = new String[3]; …

	improbabilityField	int h = 12;
		int y =16;
		int z =17;
		endRant h,y,z;

Data		
	beginRant	Local Params: String[] args = {null,null,null}
	improbabilityField	Local Params: int h= 12

		int y =16 int z =17 Parameters: args return address: “&beginRant”

note If the value of an array element is initialized it will be set to null if passed into the field.

7.Conversions

A ***conversion*** enables an expression to be treated as being of a particular type. In MaRVN implicit conversion and explicit conversions with an emphasis on the explicit conversions.

7.1 Implicit conversions

Implicit conversions occur in two places in MaRVN, the first being in String concatenation. Say one was trying to put an int into a String, it would be a pain to cast all of them to String, so an implicit conversion is needed for ease:

```
String z = "1,2,3";
int newInt = 4;
z = z + "," + newInt;
```

As shown above the MaRVN compiler will do an implicit conversion of newInt to String through some internalized mechanism yielding the final string "1,2,3,4". As shown above the string syntax for MaRVN is more like C# for declaration, but the Eralang syntax is also valid (this cuts down on readability in some cases, but makes recursion with strings far easier to handle and basic string creation a breeze).

Another example of explicit conversion is in mathematical functions. Meaning if a mathematical function requires the two variables to be the same it will cast the smaller (in terms of memory size) up to the same type of the larger. The hierarchy is as follows:

byte ->short ->int ->long
float -> double -> decimal
byte ->short ->int ->long ->float ->double ->decimal

Unsigned variables (save byte) will not implicitly cast. The simplest example of this in action is through integer and non-integer division:

```
//this will yield 1
int x = 5/3;
//this will yield 1.667 casting 3 to a double
double x = 5.0/3;
//this throws a type mismatch error as double can't be cast implicitly to an int
int x = 5.0/3;
//this will not throw an error as the implicit cast flow is preserved
double x = 5/3;
```

Even if the implicit conversion does follow the flow in mathematic examples it will throw warnings indicating that explicit conversions are preferred in these cases.

7.2 Explicit conversions

Explicit conversion in MaRVN is done using (type)variableToCast which is identical to the casting of C#. Using the explicit casting in MaRVN allows you to do quite a bit including going against the cast flow of the numeric variable types:

```
	//this will now execute, however, it will throw a warning at compile time due to potential
loss in accuracy
	int x = (int)5.0/3;
```

8. Statements

Examples of statements in MaRVN that differs from C# and ERLANG:
1. Unlike Erlang the ; is used to delimit the end of a statement.
2. Accessibility may be denoted using the following statement block:
	accessibilityType{variables and methods can go here};
3. Variable type may be similarly delimited using this statement block:
	VariableType{variable identifiers and assignment};

9. Example Programs

9.1 example of a MaRVNRant for a very basic parser

```
Public MaRVNRant bool hasSemicolon(String statement){
	if (statement.length > 0){
		if (statement.charAt(0) == ';'){
			endRant true;
		}
		else{
			//this will cascade back true if there exists a semicolon
			endRant hasSemicolon(statement.tail());
		}
	}
	else{
		endRant false;
	}

}
```

9.2 Example of a MaRVNClass for typical recursive functions

```
using BrainSizePlanet;

class MaRVNClass MExample{
```

```
	//Will return i!
	public int factorial(int i){
		if(i >1)
			endRant i * fact(i-1);
		else
			endRant 1;
	}
	//Works exactly like the erlang on assignment 2
	public String decrypt (String toShift, int shiftAmt){
		endRant decrypt(ShiftAmt, toShift, toShift.head(), toShift.length);
	}
	public String decrypt (int shiftAmt ,String toShift, char currentChar, int currentLength){
		if (currentLength == 1){
			if( currentChar == (char)32){
				//implicitly casts to a char based on ascii table
				endRant [32];
			}
			else{
				endRant [(((int)currentChar + (26-shiftAmt)) – 65 %26 +65)];
			}
		}
		else if( currentChar == (char)32){
			//.tail() and .head() are methods of the String Class that return a String and
			char respectively
			endRant [32| decrypt(ShiftAmt, toShift.tail(), toShift.tail().head(),
					toShift.tail().length)];
		}
		else{
			endRant [(((int)currentChar + (26-shiftAmt)) – 65 %26 +65)|
						decrypt(ShiftAmt, toShift.tail(), toShift.tail().head(),
						toShift.tail().length)];

		}

	}


}
```

9.3 A MaRVN Block using the MaRVN class

```
using BrainSizePlanet;
//no import for the class as it is assumed to be in the same package of classes for this example.

class MBExample{
	static fruitlessRant beginRant(){
		String[] aBitStringy= new Array [2];
```

```
            MaRVN(aBitStringy){
                //These use the scope resolution operator to use rants outside of the current
                class, otherwise it would just be the method name
                Thread t1 = new Thread(mExample::factorial(7));
                Thread t2 = new Thread(mExample::decrypt("Heart of Gold",7));

                //These actually start the thread with the argument acting as the return
                point for the endRants
                t1.start(aBitStringy[0]);
                t2.start(aBitStringy[1]);

                //will stop the code in the MaRVN block so beginRant doesn't end before
                the concurrent rants have executed. Please note this code isn't astounding,
                rather just to give a feel for the syntax, as doing this in such small code
                would be pure folly.
                MaRVN.haltSequentialUntil(t1,t2);

                endRant;
            }
            Console.sighln("Factorial result:" + aBitStringy[0] + "\nDecrypt result: " +
                                aBitStringy[1]);
            endRant;
        }
}
```

9.4 Some example input with looping

```
using BrainSizePlanet;

class OhBoyUserInput{
    static fruitlessRant beginRant(){
        String userInput = "";
        int cash = 1000;
        int bet, randomNum, moneyChange;

        //So begins a game in which you either win big or break even
        //Will exit if you hit zero or 1000000 dollars outputting every single loss.
        while(cash >0 && cash < 1000000){

            userInput = Console.order("Place your bets");
            bet = Integer.parseInt(userInput);

            randomNum = Math.random()%50

            if(randomNum < 25){
                moneyChange = bet * Math.random()%3;
                cash -= moneyChange;
```

```
                        Console.sighln("Dreadful, you lost $" + moneyChange + "
                                yielding $" + cash ".");
                }
                else{
                        moneyChange = bet * Math.random()%3;
                        cash += moneyChange;
                        Console.sighln("Boring, you gained $" + moneyChange + "
                                        yielding $" + cash ".");
                }
        }

        endRant;
    }
}
```

10. Conclusion

I will admit that my language is a bit odd. This partially stems from the fact that that name was the first concrete idea I had for this project and I just ran from there. The sarcasm suggestion aside I feel as though this language has some rather nice features absent in C# and Erlang that makes it more powerful.

First and foremost MaRVN retains Erlang's ability to handle concurrency nicely (I picked out some of the better behind the scenes aspects of Erlang and my favorite syntax) yet variable reassignment and good selection control structures exist, setting it apart from its parent. Additionally, there exist the MaRVN blocks which nicely create an environment in the execution for Erlang like concurrency in an isolated and concurrent Thread allowing for imperative code and lock free concurrent code(Erlang code) to run simultaneously in the same execution, something I thought would be seriously interesting to see the use/power of.

I certainly overused C# for this, but as it stands C# is a pretty awesome language. In retrospect the addition of an args field to MaRVN blocks and improbabilityFields was a bit irrational, but I feel that it adds a bit more readability to the code and it enforces that the variables passed will be altered in the block of code. Additionally a major improvement over C# is the way MaRVN handles fields which to me is by far a more tidy way of handling fields as enforces good code style in the field declarations, which if there is one thing I have learned deciphering the code of others is a major issue.

The MaRVN language is an interesting language with some interesting features (at least to me) and I would actually love to code in this language. It must be noted, however, that there are a lot of options in MaRVN which certainly boosts write ability, but the readability takes a less than dramatic hit (meaning it's not as bad as say PHP and it looks like a godsend to an APL programmer) as certain aspects do exist in an effort to improve readability (ex the field blocks). Ultimately, MaRVN makes a decent attempt at blending the power of functional programming with the relative ease of imperative in an attempt to create a wholly new experience from either of its constituent parts.