not(FUN)

Michael Cohoon

If at any point you feel that this documentation is not goofy, please consult pages 38, 39, and 40.

1. Introduction

not(FUN) (pronounced "Not Fun") is a modern, beginner-friendly, object-oriented, strongly-typed programming language. The name is derived from the fact that not(FUN) is not a functional programming language. not(FUN) takes characteristics from languages like Java and C#, but is different in the following ways:

1. Conventional arrays, while used behind the scenes, cannot be directly used. Instead, the preferred style is to use "dynamic arrays," similar to Java's ArrayList. These dynamic arrays, or `Dynam`, are supported by the not(FUN) language, contain multiple built-in functions, and allow for the use of generics (denoted with <>).
2. While Java also has packages, in not(FUN) a class is required to be located inside a package. The package name must be listed with the declaration of the class.
3. In addition to public, private, and package visibility types, the programmer may also define customized object and variable visibility in order to include or exclude certain classes or packages from directly accessing data. This must be done in the package's .properties file.
4. Instead of the ! operator to denote "not," they keyword `not` has been designated.
5. The not(FUN) language uses the keywords `integer`, `bool`, and `decimal` in place of Java's `int`, `boolean`, and `double` respectfully.
6. There is no automated memory management in not(FUN).

1.1 Hello World

```
%% A Hello World example written in not(FUN).
Console.output("Hello world!");
```

1.2 Program Structure

The key organizational concepts in not(FUN) are as follows:

1. The beginning point of execution in a class must go through a public entry function (similar to Java's main method.
2. The entry function does not have to be void, but it can be.
3. All classes, functions, and loops must encompass the content within brackets to clearly define and delimit its content.
4. Each bracket must be followed by a forward slash and its definition.
5. All Boolean values must be encapsulated inside of parentheses.
6. The double equals operator (==) is used for assigning values to variables, while the single equals operator (=) is used for comparisons.

Example Code:

```
package Utilities.Example
{  \beginclass

   integer max;

   public void findMax(integer x, integer y)
   {  \beginfunction
        if(x > y)
        {  \beginif
             max == x;
        }  \endif
        else
        {  \beginelse
             max == y;
        }  \endelse
   }  \endfunction

   public static void entry()
   {  \beginfunction
        findMax(3, 5);
        Console.output(max);
   }  \endfunction
}  \endclass
```

In this example, the class `Example` is declared in the package `Utilities`. The class contains a member called `max` and two functions. The first function called is `findMax()` and takes in two `integer` parameters. These parameters are local to the function and only be accessed by the `findMax()` function. The function is `void` so it does not return a value, but does change the value of the class member `max`. The second function is the `entry()` function, and is the entry point to the class. The execution begins in the `entry()` function, which prints out the value of `max` to the console.

1.3 Types and Variables

There are two kinds of types in not(FUN): *value types* and *reference types*. Value types are those in which the variable directly contains the data. Examples of these are primitive types such as integer or decimal. Reference types are those in which the variable stores references to its data. Objects are examples of reference types. Multiple variables can both reference the same object, and therefore either could affect the objects state or data.

1.4 Statements Differing From Java and C#

Statement	Example

Expression statement	`bool b == (false);`
	`integer i == 007;`
	`decimal d == 21.12;`

`if` statement	`Dynam<decimal> d == Dynam.alloc();`
	
	`public void addMin(decimal x, decimal y)`
	`{ \beginfunction`
	
	`if(x < y)`
	`{ \beginif`
	`d.add(x);`
	`} \endif`
	`else`
	`{ \beginelse`
	`d.add(y);`
	`} \endelse`
	
	`} \endfunction`

`while` statement	`public Dynam integerToDynam(integer i)`
	`{ \beginfunction`
	
	`bool flag == (false);`
	`Dynam<integer> d == Dynam.alloc();`
	
	`while(not(flag))`
	`{ \beginfor`
	`integer temp == i % 10;`
	`if(i < 1)`
	`{ \beginif`
	`d.add(temp);`
	`i == i / 10;`
	`} \endif`
	`else`
	`{ \beginelse`
	`flag == (true);`
	`} \endelse`
	
	`} \endwhile`

	`return d;`
	
	`}  \endfunction`

`for` statement	`public bool initToZero(Dynam<integer> d)`
	`{  \beginfunction`
	
	`bool isZero == (true);`
	
	`for(integer i == 0; i < d.size(); i++)`
	`{  \beginfor`
	`%% checks to see if all the`
	`%% elements are zero`
	`if(not(d[i] == 0))`
	`{  \beginif`
	`isZero == (false);`
	`break;`
	`}  \endif`
	`}  \endfor`
	
	`return (isZero);`
	
	`}  \endfunction`

1.5 Classes and Objects

New classes are created using class declarations. The example below is the declaration of a class called Employee.

```
package Resources.Payroll.Employee
{  \beginclass

      String firstName;
      String lastName;
      decimal hourly;

      public Employee(String fName, String lName, decimal hourly)
      {  \beginfunction
            self.firstName == fName;
            self.lastName == lName;
            self.hourly == hourly;
      }  \endfunction

}  \endclass
```

To create an instance of a class, the `alloc()` function must be called. This allocates memory for the object and automatically calls the constructor of the class with the values it was passed. A pointer to the instance's memory location is returned.

```
Employee emp == Employee.alloc("John", "Smith", 14.50);
```

The not(FUN) language does not have automatic garbage collecting. Once an object will no longer be used, the programmer should call the `dealloc()` function to free up the space that the object occupied.

```
emp.dealloc();
```

1.5.1 Accessibility

Each member of a class has a designated visibility, or accessibility. This allows or disallows certain functions or classes from directly accessing that data member.

Accessibility	Meaning
`public`	Access not limited
`package`	Access limited to the package the class is contained in.
`protected`	Access limited to this class or classes derived from this class.
`private`	Access limited to this class.
`specified`	Access is determined by the classes or functions listed in the package's .properties file. If no accessibility has been listed for a particular class, the accessibility is defaulted to `package`.

1.5.2 Fields

A field is a variable that is associated with a class or with an instance of a class.

```
package Graphics.Color
{  \beginclass
      public static Color Black == Color.alloc(0, 0, 0);
      public static Color White == Color.alloc(255, 255, 255);

      private byte red;
      private byte green;
      private byte blue;

      public Color(byte red, byte green, byte blue)
      {  \beginfunction
            this.red == red;
            this.green == green;
            this.blue == blue;
      }  \endfunction
```

```
}  \endclass
```

1.5.3 Methods

A *function* is a member that implements a computation or action that can be performed by an object or class.

The *signature* of a function must be unique in the class in which the function is declared.

1.5.3.1 Constructors

The not(FUN) language supports both instance and static constructors. An *instance constructor* is a member that implements the actions required to initialize an instance of a class. A *static constructor* is a member that implements the actions required to initialize a class itself when it is first loaded.

1.5.3.2 Properties

Properties are natural extensions of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have *assessors* that specify the statement to be executed when their values are read or written.

1.5.3.3 Events

An *trigger* is a member that enables a class or object to provide notifications. Clients react to triggers through *trigger handlers*. Trigger handlers are attached using the :< operator and removed using the :> operator. The following example attaches a trigger handler to the `Changed` events of a `Dynam<String>`.

```
package Examples.Test
{  \beginclass

      static integer changeCount;

      public void DynamChanged(Object sender, TriggerArgs t)
      {  \beginfunction
            changeCount++;
      }  \endfunction

      public static void entry()
      {  \beginfunction
            Dynam<String> names == Dynam.alloc();
            TriggerHandler t == TriggerHandler.alloc(ListChanged);
            names.Changed :< t;
            names.add("Christine");
            names.add("Luis");
            names.add("Anthony");
            Console.output(changeCount);
```

```
            names.dealloc();
            t.dealloc();
        }  \endfunction
}  \endclass
```

1.6 Arrays

A `Dynam`, or dynamic array, is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in a `Dynam`, also called elements, do not have to be of the same type. If generics are used to specify a particular type, then specific operations and functions can be performed on elements on the `Dynam`. If generics are not used, then any class-specific functions cannot be used. For example, if a `Dynam` contains a mixture of `Employee` objects and `Student` objects, then the function `getSalary()` cannot be used on the Dynam without first extracting the element and assigning it to a new `Employee` object. The `getName()` function can be used on the elements of the `Dynam` if `getName()` is a function specified in a `Person` class that both `Employee` and `Student` inherit from. However, if the `Dynam` was specified as being only of `Employee` objects, then a function call to `getSalary()` would be allowed. All primitive types will automatically be wrapped and unwrapped to objects when being added and extracted from a `Dynam`.

The reason that dynamic arrays are used over conventional arrays is because dynamic arrays are allowed to dynamicly change size. The not(FUN) language is aimed to be beginner-friendly, and it a lot easier on the programmer to have the aid of many built-in functions to manipulate a `Dynam`. The programmer will not need to worry about growing an array or dealing with extra, unused cells at the end of the array. Conventional arrays are used behind the scenes however. Strings are technically arrays of characters. Conventional arrays are used behind the scenes because internal operations using only dynamic arrays would cause unwelcomed overhead.

```
Dynam d1 == Dynam.alloc();
d1.add('a');

Dynam d2<integer> == Dynam.alloc();
d2.add(007);

Dynam d3<String> == Dynam.alloc("one", "two", "three");
d3.delete(d3.elementAt(0));
```

The first `Dynam` is created but not given any data. When the `add()` function is called, the size increases by one, and the character `'a'` is inserted at the end of the `Dynam`. Any type of variable can be added as an element to `d1`.

The second `Dynam` is created also without any original data. When the `add()` function is called, the size of the `Dynam` increases by one and the integer `007` is added to the `Dynam`. Because `d2` can only contain integers, adding another type of variable will result in a compile-time error.

The third `Dynam` is created and initialize with the Strings "`one,`" "`two,`" and "`three`" into the 0th, 1st, and 2nd indices respectively. The element at the 0th index (the String "`one`") is then removed.

2. Lexical Structure

2.1 Programs

A not(FUN) program consists of one or more *source files*. A source file is an ordered sequence of Unicode characters.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters
2. Lexical Analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic Analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the not(FUN) programming language where it differs from Java and C#.

2.2.1 Lexical Grammar Where Different From Java and C#

The lexical grammar of not(FUN) is similar to the lexical grammar of C#. However, unlike Java and C#, identifiers cannot consist of digits or symbols, even if the identifier starts with a character.

<String>	-->	*" <character list> <space> <character list> "*
	-->	*" <character list> "*
<identifier>	-->	*<character> <character list>*
	-->	*<character>*
<character list>	-->	*<character> <character list>*
	-->	*<character>*
<digit list>	-->	*<digit> <digit list>*
	-->	*<digit>*
<character>	-->	a, b, …, y, z
	-->	A, B, …, Y, Z
<space>	-->	space character
<digit>	-->	0, ..., 9

2.2.2 Syntactic ("Parse") Grammar Where Differing From Java and C#

function-statement:

function-identifier (*argument-list*)

```
{   \beginfunction
        embedded-statement
}   \endfunction
```

if-statement:

```
if ( (boolean-expression) )
{   \beginif
       embedded-statement
}   \endif
```

if-else-if-statement:

```
if ( (boolean-expression) )
{   \beginif
       embedded-statement
}   \endif
else if ( (boolean-expression) )
{   \beginelseif
       embedded-statement
}   \endelseif
```

if-else-statement:

```
if ( (boolean-expression) )
{   \beginif
       embedded-statement
}   \endif
else ( (boolean-expression) )
{   \beginelse
       embedded-statement
}   \endelse
```

for-statement:

```
for ( initialization-statement; comparative-expression; increment-statement )
{   \beginfor
       embedded-statement
}   \endfor
```

while-statement:

```
while ( (boolean-expression) )
{   \beginwhile
       embedded-statement
}   \endwhile
```

do-while-statement:

```
do()
{   \begindowhile
        embedded-statement
while ( (boolean-expression) ) }   \enddowhile
```

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 expansion of that non-terminal symbol into sequences of non-terminal 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) )
{   \beginwhile
        embedded-statement
}   \endwhile
```

defines a *while-statement* to consist of the token `while`, followed by the token "(", followed by another token "(", followed by a *boolean-expression*, followed by the token ")", followed by another token ")", followed by the token "`{  \beginwhile`" to denote the start of the while block, followed by an *embedded-statement*, followed by the token "`{  \endwhile`" to denote the termination of the *while-statement*.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. 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 consist of one or more statements.

2.3 Lexical Analysis

2.3.1 Line Terminators

Line terminators divide the characters of not(FUN) into source file lines.

new-line:

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

2.3.2 Comments

Two forms of comments are supported: single line comments and delimited comments. *Single-line comments* start with the characters `%%` and are extended to the end of the source line. *Delimited comments* start with the characters `%%\` and end with the characters `%%/`. Delimited comments may span multiple lines. Comments do not nest.

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
decimal-literal

character-literal
string-literal
operator-or-punctuator

2.4.1 Keywords different from Java and C#

A *keyword* is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.

List of reserved keywords:

```
\beginclass
\endclass
\beginfunction
\endfunction
\beginif
\endif
\beginfor
\endfor
\beginwhile
\endwhile
\begindowhile
\enddowhile
\beginelseif
\endelseif
Dynam
integer
uinteger
decimal
udecimal
nil
specified
```

Removed keywords:

```
goto
int
double
short
long
float
namespace
```

3. Basic Concepts

3.1 Application Startup

Application startup occurs when the execution environment calls a designated function, which is referred to as the application's entry point. This entry point function is always named `entry`, and can have one of the following signatures:

```
public static void entry()
public static void entry(Dynam<String> args)
public static integer entry()
public static integer entry(Dynam<String> args)
public static String entry()
public static String entry(Dynam<String> args)
```

3.2 Application Termination

Application termination returns control to the execution environment.

If the return type of the application's *entry point* function is `int`, the value returned serves as the application's *termination status code.* The purpose of this code is to allow communication of success or failure to the execution environment.

If the return type of the application's *entry point* function is `String`, the value returned is a precise String of the information pertaining to the success or failure.

If the return type of the application's *entry point* function is `void`, the `} \endfunction` token terminates the function. To break out of the function earlier, the `return` statement will also result in function termination. Either way, the no information regard success or failure is given from a `void entry()`.

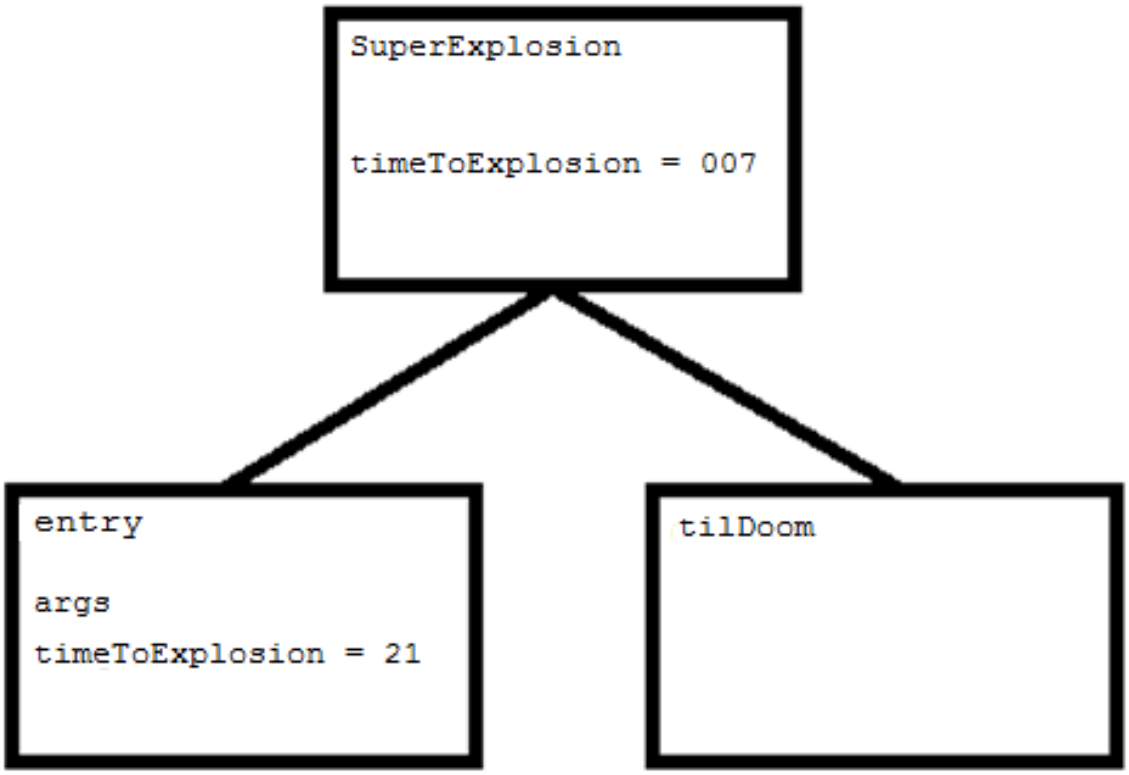
3.3 Scope

The not(FUN) language utilizes static scope. Static scope is when the scope of a variable is determined prior to execution, and is based on the spatial relationship of the subprograms. If a variable is assigned values more than once, the closest one to the call is used. If the variable is rewritten in that subprogram, that one is used. If not, it goes up a function call to check to see if that layer has the variable.

Example code:

```
package Examples.SuperExplosion
{  \beginclass

      integer timeToExplosion == 007;
      public static void entry(Dynam<String> args)
      {  \beginfunction
            integer timeToExplosion == 21;
            tilDoom();
      }  \endfunction
      public void tilDoom()
      {  \beginfunction
            Console.output(timeToExplosion);
      }  \endfunction

}  \endclass
```

Tree diagram:



3.4 Memory Management

The not(FUN) programming language does not employ automatic memory management. Programmers are required to manually allocate and deallocate the memory required for objects. The `alloc()` function should be called when instantiating the object with values, and when the object is no longer needed, the `dealloc()` function should be called to free up the memory it was occupying. This greatly differs from Java which uses a garbage collector to automatically manage memory. Hidden pointers are used, and this process is similar to C#, which uses a destructor function to destruct an instance of a class.

4. Types

In the not(FUN) programming language, types are divided into two main categories: *value types* and *reference types*.

4.1 Value Types

not(FUN) has eight different value types, and stores them on the stack. They are:

`bool, integer, uinteger, decimal, udecimal, char, byte, enum`

4.2 Reference Types

not(FUN) contains reference types which are stored on the heap. The reference types include:

`String, Dynam,` and user-defined objects

It is not possible for the programmer to directly access pointers.

5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in that variable. The not(FUN) programming language is a type-safe language.

5.1 Variable Categories

Types and examples:

`bool` – Stores a value of either true or false
```
bool b == (false);
```

`integer` – Stores any signed numerical value that is a whole number
```
integer i == -5921;
```

`uinteger` – Stores any unsigned numerical value that is a whole number
```
uinteger ui == 315;
```

`decimal` – Stores any signed numerical value that is contains a floating point portion
```
decimal d == 0.235;
```

`udecimal` – Stores any unsigned numerical value that is contains a floating point portion
```
udecimal ud == 24.15;
```

`char` – Stores any single Unicode character
```
char c == 'a';
```

`byte` – Stores an eight-bit signed two's complement integer
```
byte b == 100;
```

`enum` – Stores a named value
```
enum Planets (MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS,
              NEPTUNE, PLUTO);  %% I learned Pluto was a planet, so it
              is staying as one.
```

`String` – Stores a series of Unicode characters, including characters, symbols, spaces, and digits.
```
String s == "This is a test String for Alan.  Moo.";
```

`Dynam` – Stores different types into a single type (`Dynam`) in the form of a dynamic array
```
Dynam dyn == Dynam.alloc("one", 2, 't');
```

Static variables are those that do not belong to a certain function or object. They belong to the class and do not need to be instantiated. There is only one copy that gets accessed. The keyword `static` is used to denote static variables.

```
static String bestBond = "Craig";      %% Just kidding. :)
```

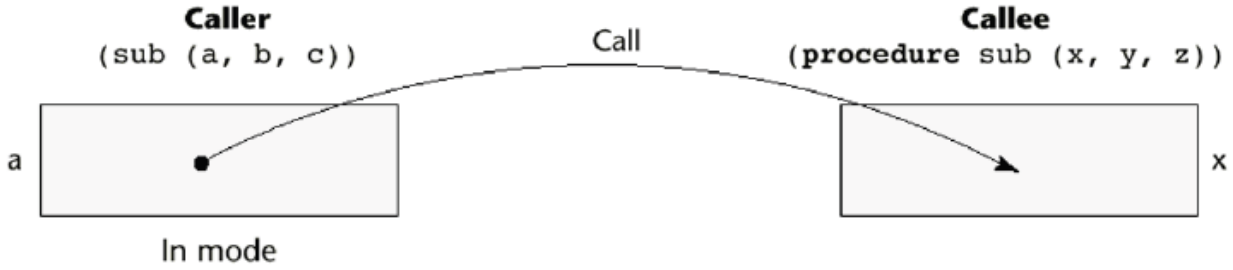
Constant variables are those who value cannot be changed. Once a constant variable has been instantiated, it must forever remain that value. The keyword `const` is used to denote constant variables.

```
const uinteger priceOfGas = 5;
```

6. Parameter Passing

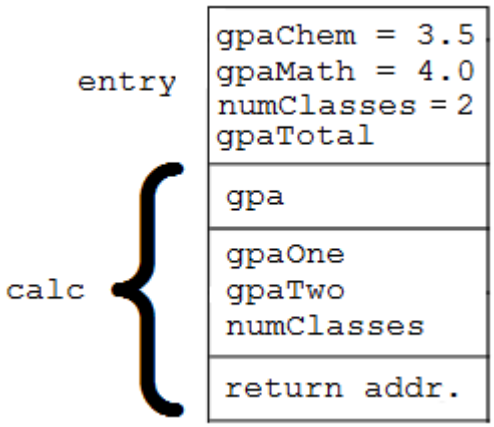
6.1 Methods

not(FUN) utilizes positional parameters, which is where a parameter is linked to its position and must be specified in the order in which it appears. Each parameter must go in its correct position, and parameters may not be omitted. Furthermore, *in-mode* semantics are used to implement *pass-by-value* parameter passing. When a parameter is passed by value, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram. This is done by copying the actual parameter and assigning it to the formal parameter.



6.2 Examples

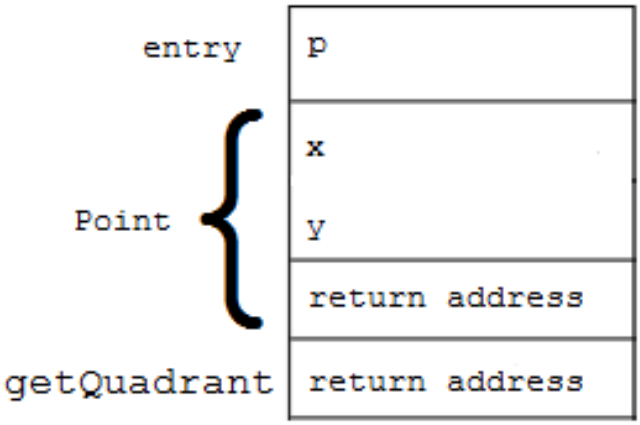
```
package Examples.GPA
     {  \beginclass

          public static void entry()
          {  \beginfunction
               decimal gpaChem == 3.5;
               decimal gpaMath == 4.0;
               integer numClasses == 2;
               decimal gpaTotal == calc(gpaChem, gpaMath,
                                        numClasses);
          }  \endfunction
          public decimal calc(decimal gpaOne, decimal gpaTwo,
                              numClasses)
          {  \beginfunction
               decimal gpa == (gpaOne + gpaTwo) / numClasses;
               return gpa;
          }  \endfunction

     }  \endclass
```



In this example, the `entry()` function does not take any parameter or have a return address. The values of the actual parameters `gpaChem`, `gpaMath`, and `numClasses` from `entry()` are copied to the formal parameters of `gpaOne`, `gpaTwo`, and `numClasses` in `calc()` respectively. The value of the variable `gpa` is returned and assigned to the variable `gpaTotal` in `entry()`. The return address is then popped off the stack so the program knows where to resume from.

```
package Examples.Point
      {  \beginclass

            integer x;
            integer y;
            bool isFirstQuadrant;

            public Point(integer x, integer y)
            {  \beginfunction

                  self.x == x;
                  self.y == y;
                  if(getQuadrant() == 1)
                  {  \beginif
                        self.isFirstQuadrant == (true);
                  }  \endif
                  else
                  {  \beginelse
                        self.isFirstQuadrant == (false);
                  }  \endelse

            }  \endfunction

            public integer getQuadrant()
            {  \beginfunction

                  if((self.x >= 0) && (self.y >= 0))
                  {  \beginif
                        return 1;
                  }  \endif
                  else if((self.x < 0) && (self.y >= 0))
                  {  \beginelseif
                        return 2;
                  }  \endelseif
                  else if((self.x < 0) && (self.y < 0))
                  {  \beginelseif
                        return 3;
                  }  \endelseif
                  else
                  {  \beginelse
                        return 4;
                  }  \endelse


            }  \endfunction
```

```
        public static void entry()
        {  \beginfunction

              Point p == Point.alloc(2, 2);
              Console.output( (p.isFirstQuadrant()) );
              p.dealloc();

        }  \endfunction

}  \endclass
```



In this example, `entry()` does not have any local variables or parameters. The `entry()` function has a `Point` object called `p`, and a reference to `p` is placed on the activation record. The `Point` constructor does not have any local variables, but it does have two parameters called `x` and `y`. The return address of `entry()` is also stored. The `getQuadrant()` function is called by `Point` constructor, and `getQuadrant()` does not have any local variables or parameters. It stores the return address of the `Point` constructor.

7. Conversions

A *conversion* enables an expression to be treated as being of a particular type.

7.1 Implicit Conversions

Implicit conversions are those done automatically. The not(FUN) programming language does not support implicit conversions. The exception to this rule is division operator dealing with dividing an `integer` by a `decimal`. If this is the case, and the quotient is being assigned to another `decimal` variable, the `integer` will automatically be converted to a `double` for just that calculation. Otherwise, if there is a type mismatch, an appropriate error will be printed to the console.

7.2 Explicit Conversions

Explicit conversions are those done intentionally by the programmer to convert an expression to a particular type or to convert one type to another. This is done through casting. The programmer must specify the type to convert by enclosing the type inside parentheses before the variable or expression.

Conversion example:

```
integer i == 3;
Console.output((decimal) i);
```

In the above example, the output is `3.0` and not `3`. This is because the `integer` value was converted to a `decimal`.

8. Statements

Examples of statements in not(FUN) that differ from Java and C#, as noted in the introduction:

Dynamic array example:

```
Dynam<integer> dyn == Dynam.alloc();
dyn.add(3);
dyn.add(0);
dyn.add(1);
dyn.delete(d3.elementAt(1));
Console.output(dyn);
dyn.dealloc();
```

The output of this example is: 3, 1. It is formatted through an implicit call to the `toString()` function.

Package and class header example:

```
package Bond.DieAnotherDay.Brosnan
{  \beginclass
     %% embedded statements
}  \endclass
```

In this case, the class name is `Brosnan`, but the full package path must be included.

Custom visibility example:

In the class:

```
package ExampleCode.Caterpillar
{  \beginclass
     specified String hippo == "User-defined visibility.";
}  \endclass
```

In the Caterpillar.properties file:

```
specified = [self, ExampleCode.TuringMachine,
             ExampleCode.Stopwatch, Crazy.Awesome];
```

In this example, only the classes `TuringMachine` and `Stopwatch` in the `ExampleCode` package, `Awesome` in the `Crazy` package, and the `Caterpillar` class itself can directly see and access the `String hippo`.

`not` *operator example:*

```
if(not(somethingStrangeInTheNeighborhood))
{  \beginif
     goOutside();
}  \endif
else
{  \beginelse
     callGhostBusters();
}  \endelse
```

Primitive type example:

```
integer i == 11;
bool b == (false);
decimal d == 6.12;
```

Memory management example:

```
Ninja n == Ninja.alloc(); %% to allocate new memory
n.dealloc();              %% to deallocate memory used by n
```

9. Example Programs

```
package Math.Stuff.Fibonacci
{  \beginclass

      public integer nthNumber(integer n)
      {  \beginfunction

            if(n <= 2)
            {  \beginif
                  return 1;
            }  \endif
            else
            {  \beginelse
                  return nthNumber(n - 1) + nthNumber(n - 2);
            }  \endelse

      }  \endfunction

      public static void entry()
      {  \beginfunction

            integer num == Console.input.Integer();
            Console.output(nthNumber(num));

      }  \endfunction

}  \endclass
```

The above example is the Fibonacci sequence written in not(FUN). The class and package are specified at the beginning of the class, and keywords denote the start and end of all content enclosed inside of a brace. This makes it easier for the programmer to determine what function or loop ends at which brace. Input from the user is neatly built in through the console function, and the type of input can be specified at the point of reading. `Console.input.Integer()` expects to receive and integer, and if that does not happen, an exception is thrown and the user is prompted that the input must be an integer. Alternatively, `Console.input.getInput()` can be used to receive any keyboard regardless of its type. Printing to the console is also very simple, and `Console.output(nthNumber(num))` is all that is required to print the nth number of the sequence.

```
package Schools.Classroom.Student
{  \beginclass

      String name;
      integer grade;

      %% Class constructor
      public Student(String name, integer grade)
      {  \beginfunction
            self.name == name;
            self.grade == grade;
      }  \endfunction

      public double getClassAverage(Dynam<Student> dyn)
      {  \beginfunction

            integer sum == 0;
            for(integer i == 0; i < dyn.size(); i++)
            {  \beginfor
                  sum == sum + dyn[i].getGrade();
            }  \endfor

            return sum / dyn.size();

      }  \endfunction

      public integer getGrade()
      {  \beginfunction

            return self.grade;

      }  \endfunction



      public static void entry()
      {  \beginfunction

            Student s1 == Student.alloc("Chico",   80);
            Student s2 == Student.alloc("Harpo",   95);
            Student s3 == Student.alloc("Groucho", 60);

            Dynam<Student> students == Dynam.alloc(s1, s2, s3);

            Console.output(getClassAverage(students));
```

```
            Student s4 == Student.alloc("Alan",  -2);

            students.add(s4);

            Console.output(getClassAverage(students));

            s1.dealloc();
            s2.dealloc();
            s3.dealloc();
            s4.dealloc();
            students.dealloc();

        }  \endfunction

    }  \endclass
```

In the example above, the class `Student` is implemented in the not(FUN) programming language. The `Student` objects are created and then stored in the `Dynam` called `students`. The class average is calculated by passing the `Dynam` into the function and traversing it, storing each grade into a `sum` variable. The sum is the dividing by the size of the `Dynam` (the number of `Students`) and is returned. In this scenario, a student misbehaved in another class and had to be put into this classroom. Thus, the student needs to be stored into the `Dynam` containing all of the `Student` objects. If a Java array was used, we would need to spend time growing the array. In not(FUN), the `Dynam` is a dynamic array that automatically increases in size when a new element is added, and automatically shrinks when an element is removed. This is quicker and easier on the programmer. The two outputs would be `78` and `58` in that order. Since the fourth student's grade was so low, the class did not get their pizza party.

```
package Network.Web
{  \beginclass

      public static Document download()
      {  \beginfunction

            WebClient c == WebClient.alloc("www.labouseur.com");
            Document d == Document.alloc(c.getDocument());
            c.dealloc();

            return d;

      }  \endfunction

      public static void entry()
      {  \beginfunction

            Document doc == Document.alloc(download());
            Console.output(doc);
            doc.dealloc();

      }  \endfunction

}  \endclass
```

In the above example, it can be seen how not(FUN) deals with memory management. When the programmer needs to reserve memory for an object, the programmer must call the `alloc()` function. Any parameters passed into the `alloc()` function are passed directly into the constructor of that object. Once the programmer is certain that the object will no longer be used or referenced, the `dealloc()` function should be called to free up the memory space the object was using.

```
package Examples.Math
{  \beginclass

      public static integer floor(decimal d)
      {  \beginfunction

            return (integer) d;

      }  \endfunction

      public static integer ceiling(decimal d)
      {  \beginfunction

            integer i == (integer) d;

            if(d = (decimal) i)
            {  \beginif
                  return i;
            }  \endif
            else
            {  \beginelse
                  return i + 1;
            }  \endelse

      }  \endfunction


      public static void entry()
      {  \beginfunction

            decimal dec == 3.14;

            Console.output(dec.floor());
            Console.output(dec.ceiling());

      }  \endfunction

}  \endclass
```

The above example shows the explicit conversions of not(FUN). If a variable is to be converted to a different type, the type must be specified in parentheses preceding the variable. This class used casting to implement the `floor()` and `ceiling()` functions for decimals. Unlike Java, in the not(FUN) programming language, the == operator is used for assignment, and the = operator is used for comparison.

10. Conclusion

The not(FUN) programming language incorporates what modern object-oriented programming languages do well, and makes adjustments to the areas that have some issues. The not(FUN) language was supposed to be beginner-friendly, and that is accomplished in many ways. Both output and input to the console are extremely easy, and they simply require the programmer to call a built-in function. When reading keyboard data, the programmer has the option of specifying what data type he is looking for, and if a different type is received, an exception is thrown and the user is automatically prompted. After every bracket in a `class`, `function`, `loop`, and `if`-statement, the programmer is required to denote what the bracket is doing and what code it is containing. This visually allows the programmer to determine what code block is finishing where, and allows for an easier time for beginner programmers to insert code into the proper locations. This feature also promotes self-commenting code.

An important new feature to the not(FUN) language is the `specified` accessibility. This feature enables the programmer to specify what classes and packages can directly see and access a certain variable or function. The visibility is set in the .properties file of the class, and all variables or functions in that class that have the `specified` visibility can only be accessed by the classes and packages explicitly stated in the .properties file. This new feature enables the programmer to finely select only the classes that should be allowed to see this data, promoting data privacy and security. This capability was not implemented in Java or C#.

While arrays are used behind the scenes, the programmer cannot directly implement the use of arrays. Instead, the programmer is encouraged to use the `Dynam`, or dynamic array. The `Dynam` is similar to Java's ArrayList, and its size dynamically changes depending on the number of elements inside. If the `Dynam` is full but needs to add another element, it automatically will add another cell to insert the new data. If an item is removed from the `Dynam`, the size is automatically decremented, saving space in

memory. As arrays cannot dynamically change in size, the programmer would either need to grow the array, or have unused cells occupying memory.

The programmer is unable to directly utilize pointers, and functions implement pass-by-value parameter passing. This is done to ensure that memory is not corrupted or deleted when it should not have been. Also, the programmer must conduct his own memory management. When an object is created, the `alloc()` function is called. This allocates space in memory and automatically calls the constructor of the object. When the programmer is finished using an object, the `dealloc()` function should be called to free up the space that the object was in. This allows for other data to be stored in that location. Because memory management is left for the programmer to control, there are no long pauses caused by the garbage collector hijacking the program.

The keywords that not(FUN) supports increases code readability. The self-commenting code allows for others to read through the code and efficiently understand the process. Because of the code readability, an idea of strong writability is slightly compromised. The rationale behind favoring readability over writability is this: code that is easier to read and understand is easier to debug. Readability promotes “debugability” and allows for multiple programmers to efficiently and effectively understand and analyze code segments. Easy-to-follow code is also well suited for beginning programmers.

Overall, I believe that not(FUN) programming language has a some very useful features that I wish that real, modern languages incorporated. Specifically, the ability to set custom accessibility limits for variables or functions would be extremely useful, allowing for very specific accesses to certain pieces of data. The language itself may sometimes be a little bulky, but it is an easy-to-follow, simple language that has a powerful capacity to automate certain functions that Java and C# do not include. Many unique ideas from this language have the potential to be valuable to the programmer, and if real

languages incorporated some of these possibilities, it would be easier to create secure and efficient code.