BFL

Chapter 0: Why BFL ?

If you in a class or just keen to get going it's OK to jump to chapter 1.

BFL (big falcon language) is intended to be a the best introductory language the computer programming, and a useful Computer language for those who program intermittently or those who would like to read but not write code.

Historically, programming languages where limited by the power of the computer that they were built on. In the old days languages like C were designed to be as simple and efficient as possible to compile and have a very brief syntax to save on typing. Modern day computer languages were based on these kinds of syntax. Now when computers regularly have more than a gigabyte of RAM these brief syntaxes only have advantages to those who are transitioning form one historic language. Yet these brief syntaxes while effective at increasing typing speed or Also very opaque for new programmers. The high dropout rates in freshman programming courses reflect not the difficulties programming but the difficulties in memorising strange legacy syntaxes.

In these modern days when computers are affecting all of our lives we seem to had reached the Stage like mediaeval Europe where the important discussions of the time were written in Latin rather than the spoken language of the people at the time. When laws were only written uninterpretable in Latin the common folk required a layer of people to read and interpret these archaic languages. The Reformation karked the beginning other translation to the language which ordinary people used. This also marked a remarkable period of democracy when people could read the Bible and laws from themselves.
In many ways software has reached a similar stage of complexity. Alan Kay once said that we would not believe someone was literate if they could only read and not right. How then can we believe people are computer literate when they cannot write their own code? BFL seeks to achieve a similar level of intellectual openness by giving the tools of production to much wider audience. Rather than teach everybody to read Latin (as many in that the time argued) why not make books available in translation. Similarly in code why use short codes when we can use English ?

The essential design principle of BFL was to match the thinking of noncomputer users. While many of the principles of current programming languages remain in the background the syntax is intended to be easily memorable. The language is designed for students of computing as languages like BASIC/PASCAL were intended to be. The language is also intended to be useful for those so called 'perpetual beginners' , programmers who may program something useful once or twice a year to enhance their own productivity. The language then is set in the hinter world of intermittent use, a space which up until this point no programming language has targeted.

This said BFL is intended to encourage everybody to have a good programming style. There are many elements which are features of the language that are only libraries in other languages. Professional programming concepts such as test driven development, assertions, self checking software are built into the language. While these could've been inserted, as many languages do, as extensions or libraries, the space that large compilers offer allows elements of the language to be presented. It is hoped that when programmers move on to less extensive programming languages they will take the styles and lessons learned onto to more complex libariy and convention hevey but smaller languages.

So given this target audience there are large number features which are intended to enhance correctness. One example is the introduction of types of numbers. In most protolanguages there are two Numbers. Integers and floating point numbers. In BFL Numbers typically have units. For example you can use $500,003,223.33 as a valuable currency. You can multiply this number by another number but if you multiply two currencies together the compiler will complain. This makes no semantic sense and as such indicates a possible error. There have been a wide number of publicly identified software failures in missions where large teams have become confused about what units they are using (miles, feet, meters). The BFL compiler therefore understands a wide range of units, distances, weights, times, speeds, forces, and currencies. It knows the writing conventions for these kind of units. As such literals can be checked for the validity. For example £40,000,000.24 is valid but £4000,00,00.2222 is not. They are also a number of verbal literals eg.
£5 million, $5 Thousand, € 4 hundred. Y one trillion.

So for BFL correctness and readability go hand-in-hand. The objective others language is to allow a syntax which I'm nonprogrammer can read and understand enough to check the correctess of the code. Again it is known that large number of finance institutions operate on in house software in the form of spreadsheets which are in part incorrect. This lack of opacity between code crafters and those overseeing the projects had been one aspect attributed to the financial crisis of 2008.

For more information about BFL and the thinking behind it see appendix 1.

Chapter 1: The first step

That is enough introduction let's now jump straight into some examples.

The easiest way to learn BFL is with a keyboard and a screen. BLF has a REPL interface. You can type at it like Python and Lisp.

Let's start with something simple try typing this code.

```
Tweet hello world, how are you?
```

What does this do?

Tweet - is a command which issues the following text as a tweet. If your environment is set up then you may see the tweet emerging on twitter. If you don't have the Twitter connection an emulator is started with which you can access a Social media like feed. This can be accessed via a web browser at http:/ 127.0.0.1:5924/

The comment has two elements
Tweet - the verb or action you want the system to take.
Hello world, how are you?

The text. Many programming languages this text is put into quotes like his "hello world, how are you?" Indeed you can also do this in BFL, but in many cases it's optional.

```
Tweet "Ed Balls"
```

If command is expecting a single text is an argument it will continue reading the line until it reaches a full stop . ! or ? If you fail to add one you might see a prompt put up to say 'I'm still waiting for the full stop. If you want to put a full stop in you tweet then put it in 'quotes.

Inany programming languages this is called a string, however the word 'string' is archaic and nondescriptive. In BFL we use the word text or sentence. The only difference being text can be arbitrarily long (as short as "a" and as long as the works of William Shakespeare). A sentence can be as long as a full stop.

Notice texts are unicode. So they can contain any valid unicode letter including emoji. It should be noted that the source text for a BFL program is also Unicode. This means that certain types of notation are included in the language for example.

```
Put √ 144
```

Will print the square root of 144

In most programming languages you have the ability to print text. Again this is confusing In BFL you can put something.

```
Put  Hello, Ed Balls.
```

Will put the Sentence Hello World out into the interactive screen. While this has a lovely history going back to the 1940's it's not how many real people work with a computer these days. You can use **put** in many circumstances like this

```
Put "Hello world" onto clipboard
```

This lets you become more productive information put into the clipboard can be used by any graphical user interface for example Word and Chrome. Text, Image, Sounds, Movies can all be copyied to the clipboard. You can now use the clipboard to perform useful tasks such as.

```
Put 3043 x 4343 onto clipboard
```

Notice you used the standard x as used in maths not the * common in many programming languages. BFL will accept * but it will convert it to x.

To be even more friendly we could use

Put 3045 times 54 onto clipboard

You can use BFL like a large calculator for example

```
Put  343 + 34 + 23 + 34 + 65 + 34 + 23 + 54+  ( 87 x 3 )
```

You can use the content of the clipboard. For example try this code

```
Get  clipboard
Put it
```

When you use the get command it will put the result into a universal scratch variable called **it.** The objective of it is to create English like sentences for example

```
Get clipboard
Put  ( it plus 20% ) followed by " including VAT" onto the  clipboard
```

Now you can take a piece of text such as £120 from a spreadsheet, switch to BFL and then jump to your email and have convert it to £216 including VAT. If you email quotes to people you have just automated a tedious, error prone task.

That sounds useful so lets make up a new word we can use when we want to avoid typing both words. How about this.

```
To plus vat
      Get clipboard
      Put  ( it plus 20% ) followed by " including VAT" onto the  clipboard
End
```

Notice the indentation (like Python) is also important. Like Hypertalk and Python BFL uses indentation to make sure it understands what you are saying.

Now when you type

```
Plus vat
```

The two lines of commands we used to define the word will be executed.

For expert programmers the fact that the name has two parts (plus vat) not plusVat or plus_vat or plus-vat might be little confusing. BFL is grown up enough to handle multiple word identifiers. You will notice

```
 plus vat
 plus    vat
```

And

```
PlusVat
PLUS vat
Plus               Vat
```

All do the same thing. BFL is largely case insensitive.

Chapter 2: Productivity

The next common command is When. This sets up a trigger. For example

```
When timer 30 seconds up
      say time is up.
End
```

Once a When is primed the code is kept and executed when the timer is up.
Notice the When uses indentation. Also it uses a new command called "say". Say is like put except it speaks the text out aloud. It is a predefined command for

```
Put hello, world. into speech.
```

You can use When in many cases for example

```
When time is 6:00
      say "It's time to go home".
End
```

Note depending upon your version of BFL if you **quite** BFL all the *When* timers and so on are lost.

Depending upon you settings you can use whens like this

```
When the document "bob.docx" changes
      say Someone else has altered Bob.
End
```

```
When the document "http://www.schedule.co.uk" differs from "tempDox.html"
      play "alter.wav"
      say Warning - things have changed.
End when
```

A when statement actually just regularly checks in the background you can alter this when you set up the wen.

```
When time is 6:00  check every 5 minutes
      say "It's time to go home".
End
```

This will work to the nearest five minutes.

Remind me works like when but will keep reminding you

```
Remind me every two hours
      say "you should get up and stretch"
End
```

This assumes there is an internet connection and you have stored the previous web version of www.schedule.co.uk in tempDox.html If you don't just use 'changes' which will automatically store the previous version.

When's can be used in programming to - see later.

Chapter 3: Numbers

On the good side of BFL there is only one number. Called number. There are no floating point of integer numbers. These concepts do exist but don't have to be - see the section on fast and compatiable. A number in BFL can be as large as you like. 100,000,000,000,000,000,000,001 is bigger than any number you might encounter an can be held efficiently on a machine. Numbers expend up and can get as large as you like

When you detail with numbers it's useful to keep them for later use you can do this in BFL with put the result of calculation into a scratch area or technically a variable.

```
Put 394 * $3030 into cost
```

This creates a variable called cost. IF cluster cost exists the value is updated. If cost exists it must have the same type (e.g if you put dollars in it must have had dollars before).

If you type

```
Put cost
```

Notice when you do you get $1,193,820. Numbers of units. In this case we put $3030 which was in dollars into the machine. I knew if you multiply a dollar value by a number you get a number.
Try this

```
Put £40 * $40
```

You should get the message
In appropriate mix of units

BFL knows you can do this

```
Put 40m * 12m
```

But notice the result is 40m2 (meters squared). Notice that while you can operate on various units there is no automatic conversion. eg.

```
Put 40f +  12m
```

Will not work and generates a warning. BFL does not like putting 'invisible' code into operations. You can always convert.

```
12 +  Feet to Meters of 4f
12 +  Feet to Meters( 4f )
or
12m + (4ft as meters)
```

Note you cannot convert Currency you cannot covert with 'AS' between units so 4f (a distance) as Kg (a weight) will get a very very unpleasant message and stop the program working. Just so you know.

In BFL you are encouraged to use separators for example
$1000,000,000.

Notice again that you have to get them right os

```
€19,000000,0000,00000000
```

Will create an error. The purpose of this is to stop programmers entering in Malformed numbers into programs.

Notice that BFL also knows about the format of currency so that

```
£20,000.23
```

Will work but having more than 2 decimal places of cents

```
$303,000.2323232
```

Will not because you can't have more than .2323232 fractions fo a cent. If this is not to your liking you can convert to normal numbers perform the calculations without restriction and go back again.

The Units BFL accepts as static constructs in code are

```
£ - pounds
$ - Dollars
€ - Euro

304m - Meters
23cm - Centimetres
12mm
1.2km
12ft - feet
22yrds - Yards
56miles - miles

126Kg
232Lb
23Ton
23oz

14ø - 14 degrees.

34C - centergrade
23f  - farienhieght
230Kelvin - kelvin
```

Notice that all units come either before or after the digits making up the numbers.

You can also use % for percentage so 50% is the same as 0.5 as a constant so
Put 40 * 50% — will put 20 as you might expect

You can also use constants like
Millions , Thousand ,
For example 34Million
32 Thousand
2.3Billion

Variable names

One problem with on the go defitnions is the possibility of accidentally creating a second variable
You cannot have two variable names which are similar to each other.

```
Put 2 ÷ 3  into work to be done
Put work to be done + 1  into work to be donne  — - accident creating new var
```

BFL will not allow you to create two variable names which look and sound the same. The specification is in the appendix. This helps both remove accidental errors AND makes the code more readable to humans. It also permits the removal of predefining variables as found in languages like C/C++/Java/Java script.

By forcing names to be far enough away the compiler is able to guess at simple miss spellings such as

```
Put 2 ÷ 3  into work to be done
Put work to be don + 1  into work to be done – compiler guesses correct name
Put wok to be done into output  - -  assumes that you meant work to be done
```

Not this guess facility can only be used for variables which exist (created previously).

What if the complier complains about two words which you are sure are NEVER going to confuse.

If you need two variable names which are close to each other my position in space x , my position in space y Then use

```
LOCAL  wok to be done
LOCAL  work to be done – force existence of second similar one
LOCAL  Y  as Matrix
```

IF LOCAL IS USED THEN NON CLOSE TYPES ARE DROPPED

NOTES ON TYPE
Put 2 - 300 into cost As dollar

1…45 — generates range object. Which turns into array ?

4 Intersects 1…56

MAKE A TYPE AS Number called pace pc — pc has to be unique from all other units
MAKE A TYPE AS Number called floor flr with Range 1..16
MAKE A TYPE AS Currency called floor with range 1..45 — same as number but multiplication forbidden

14Floor — this is OK
-1Floor — error outside rage
12Floor * 35Floor — error multiplation
11Floor + 45floor — error computation outside rage

to Convert pace to feet — Give code to handle conversion

Put 45pc * 343

Int into =ddd

ACCESSING ATTRIBITES OF AN OBJECT
Put [a. b. c. d. e. f. g.] into stuff as list
Put stuff's count into my size
Put stuff's average into my average - - this is just an attribute fetch
Put stuff's median into my average
Put [[1 2] [1 2]]'s dimension into list
Put [[1 2] [1 2]]'s tensor into list
Put [[1 2] [1 2]]'s transpose into list

Tell my window to move up **with** 56
Tell *my window* to
    move up with 56
    resize with 40 and width to 400
    set the background colour with green

Tell my window to move up **to** 56
Tell [ObJECT OR OBJECT LIST] to << ACTION >> [**with** I **to** << ARGUMENT >>]

With height of 45 AND depth of 67 — NAMED arguments to method Height , Depth

Put 4 into coordinate's x — calls accessor
Put cooridinate's y into distance — calls distance

— this would cause a conflict with existing word There so it's assumed to be

Chapter 3: Calculations

Maths has very little to do with computing and it's hard for many to see the difference. On problem with many beginners is telling where maths is or is not correct for example to many beginners
x = x + 1
Seems like a mathematical tautology. Yet this is common in many programming languages. This comes over the semantics of = which means assignment not equivalant. By using put BFL avoids this confusion. As students become more proficient they can graduate. As mentioned BFL uses unicode so we can use all the mathematical symbols in Unicode for their proper purpose. For example

```
Put  45 x 23
Put  45 - 23
Put  16 ÷ 4
Put  23 + 19
```

We can use brackets
Put (34 + 2) ÷ 3

We can use square root.

```
Put   √ 64
```

We can use exponentials

```
Put 45!
```

We can raise numbers

```
Put 56 to the power of 2  — or 56^2
```

This is were we discover a significant feature of BFL. You can type put 34 * 54 into BFL. Except when you compile the compiler is free to alter the text. It will convert * to x and / into ÷ for clarity. There are many such elements like '{' becomes begin and '}' becomes end this gives all the brevity of use of languages like C with out any of the lack of general readability.

We can also use variables or a place to store things while you work on them. You can make variable using put.

```
Put 45 into thingy
Put 23 into other thing
```

Notice variable can be more than one word. The case is not significant
Put 56 into bob
Put BOB x 34

Given the text is uicode you can use practically any character in any language - including emoji to be part of the name

```
Put happy. Into 😃
```

Is perfectly legitimate

One common problem with declare as you type languages is the accidentally miss typing of names for example

Put 34 into flariervich — now contains 34
Add 1 to flariervich — now contains 1 expecting 35.

Does not produce the correct result as flariervich and flariervich are subtly different

In Bou cannot make two variables in the same scope which sound the same and can be confused when reading. BFL can be quite pesky about this.

```
Put 34 into flariervich
Put 40 int   flariervich  – compiler complains
```

One of the problems with declare as you type languages is the problems of miss-typing variables a problem which plagues many beginners. Strictly speaking BFL regards any to words as identical if there Soundex value is the same. This can be changed to give differing levels of strictness and flexibility.

Chapter 4 Type

We have already met some of the built in types for example units and sentences. Some variables like it can contain both. Many times its useful to have one or another. Type semantics are important in many more advanced languages and it's important that BFL can support transitioning to those languages. IN BFL type this is done via the 'as' construct

```
Put 34.00 into  account as number
Put 34.00  as number into account
Put  hello, world. Into message as text
Put  304  into account as counting number
Put 304 into account as integer  – shortcut  int account  <- 303

Put 405.3 into account as measurement
Put 404.2 into account as floating point  – short cut float account <- 404.
Put 405.2 into account as scientific
```

A counting number or integer is just that a number which you can count with. For example the number of students or chairs in a class. It can be negative

```
Put 30000 into account as cardinal number – short cut 3000-> unsigned account
Put 36434 into account as an id number
A cardinal number or id number can include 0 but never been negative.
```

For programming language experts numbers like integers and unsigned are familiar low level constructs. In BFL these numbers have all The advantages of being unlimited in size. If you want to talk to hard ware you need to use the 'fast' adjective. For example

```
Put 303 into mem as fast int
Put 4034. Into thing as fast float
```

Notice you are expected to understand that fast numbers exist to function on the hardware quicker. There are a number of limitations on fast numbers. They occupy fixed amount of memory and can run out of numbers or limit accuracy. While good for real world uses the understanding of limits can confuse those new to the subject. Not many values in external data will be 'fast' data and it will be common to 'clip' data to fast sizes for other programs to use.

Note once a variable has a type it can't be changed.

```
Put 394 as message into account
Put hello. Into account – error cannot put sentence into a message
```

Note all units atomically convert to typed units. You can remove this by removing the restrictions

```
Put $506.44 into account as scientific.
Put account – prints 506.44
```

You can use the type of there variables for example
Put 403 into deduction with same type as account

If account is defined as dollars then deduction will be dollars if the account is pounds then deduction will be pounds.

```
100Kg's pound – convert using attribute
100Kg as Pound   – convert using convert
100Kg as text -> "100Kg"
"100Kg" as Kg  -> 100Kg converts text to weight Kg
"100Kg" as number – converts to 100 looses type
23 + "23" -> 46 looks in var auto converts
23 + #b101001110011000111001 – both are numbers so fine

Note all things have real and virtual attributes accessed by the possessive 's

100Kg's type – Kilogram
100Kg's lock – YES  - - a literal is a constant so you can't change it.

(100Kg * 100Kg )'s type – "Kilogram2"
(100Kg/100Kg)'s type  - "number"
```

Along with units, Currency BFL can also understand literal dates (see a later chapter) and coorindates in longitude and laitire

For example

```
Put 40° 26′ 46″N 79° 58′ 56″ W into location
Put 40° 26.767′ S 79° 58.933′ E Into other location
```

Note the whole format must be used eg.

```
Put 40° 26.767′ S 0° 0′ E. Into other location
```

If you wish to send a message to surround by round brackets

```
Put ( 40° 26.767′ S 0° 0′ E.)'s type  – prints lat-long
```

Note while BFL knows what a lat-long is you need to import a Geodesic module to do computations with them.
Note while BFL does not handle complex numbers directly (again see the module) it does understand the format with the I unit here I is the square root of minus one.

Put (3 ; 3i)

Complex numbers just use BFL's tuples for representation

For completeness there is also Percentage so 90% represents 0.9 this allows the program to use normalised numbers for example you might talk about 50% * screen width.

Chapter 5 Locking variables

You can also lock variables so they can't be changed for example

```
Put 3.1419 as locked scientific  into PI
```

By convention locked numbers are given in capitals or with a leading small k

```
Put 30000 as locked counting number into kMAX ACCOUNT
```

Locked numbers can go faster and allow the use of subtle optimisations. In more difficult programming languages you wlll hear these referred to as constants.

Notice numbers and all variables can be locked retrospectively for example

```
Put 405 into jam jar
Lock jam jar
Put jam jar  — fine prints 405
Put 303 into jam jar — can't it's locked
```

If they are they can be unlocked

```
Unlock jam jar
Put 303 into jam jar  — this is no fine.
```

```
Local locked Jam jar  with 3.22.112
Put 3.141 into Jam Jar as locked  - - cannot be unlocked.
```

Sometimes you see this called mutability - a mutable variable is one that can changed as normal an unmutable variable is locked. Another term used is read only.

Text and sentences

You can use a sentace where ever you need a text but not the other way around. A sentence is any list of words (they don't have to make sense) which ends with a full stop, like this.

BFL understands texts are composed of lines, which have sentances and sentences contain words, words contain letters.

```
Put the first word of Hello how are you?
Put the second word of "hello how are you?"
Put the third letter of hello.
```

You can use put to get information from a file stored next to the program.

```
Put the document "Shakespeare.txt"  into Will
Put the second line of Will
```

You can build these up

```
Put the second letter  first word of the second sentence of the 100th line of
Will
```

Or

```
Put the  letter 2 of word 1 of sentence 2 of line 100 of will
```

You can do quite a lot with text for example

```
Get next tweet "#BFL"

If it contains Bob. do this
      say "Bob has spoken" joined to it
End if
```

Note if your say doesn't work use put instead.

The action contains looks in one text (it) for the following text (bob) and then does the list of commands afterwards. This kind of decisions making statement is useful when your have a when statement

```
When tweet "#BFL" arrives
      //The tweet is in it.

      If it contains Bob. Then  do this
            say "Bob has spoken" joined to it
      End if
End if
```

What would be do if we only want to speak if the tweet begins with Bob ? We could use find

```
Put "Bob said this" into message

If  location of "Bob " in message  is 1 then do this
      say "Bob has spoken" joined to message
End If
```

Location returns the location of one text within another.

CHAPTER Dates and times

BFL understands literal dates in text such as

```
Put 3rd of April 1999  into my date
2 of April 2000
April 2000
Mar 25 2015
```

It also record
ISO format

```
2/April/2015
3rd/04/2015 –3rd of April  with out ambiguity
```

Notice that ISO is the only format which will accept a pure date string separated by slashes. There can only be 4 digits in the year (so dates before the first millennium are forbidden by ISO format.

```
2015/03/02
```

The reason for this is to avoid problems with confusion over UK and US style dates. All literal dates are checked for correctness for example 51/Jan/2000 will be rejected at compile time.

```
Put 3rd of April 1999 - now  into my difference
```

Dates are objects and you can use the object syntax to do operations on them such as

```
Put my difference's seconds – - put the number of seconds from now to 3rd of
April 1999
```

```
3rd/04/2015 ~ 3rd of April 2015  — WOW

"2/3/2019"'s us date  — can fail
 "2/3/2019" as US Date
```

Put the number of hours from <<date-time>> to <<date-time>> — standard function
Put the number of working days from <<date-time>> to <<date-time>>
Put the number of Months from from <<date-time>> to <<date-time>>

BFL can also handle time formats

Put 3rd of April 1999 00:12:22 into my date

It can also handle Time zones

```
Put 3rd of April 1999  00:12:22  GMT+0000 into my date
```

ISO APPLIES
2015-03-25T12:00:00Z

Dates and times can be given to ranges

```
3rd of April 1999  00:12:22 … 4rd of April 2030  00:12:22 into time range
```

Chapter more advance puts

Put open file handle "bob.txt" into success **with** the file handle
Put open file handle "bob.txt" into success **;** the file handle

In this case the function open file handle returns two arguments - a flag called success and the file handle

Methods can return more than one argument.

Put check file with open file handle "bob.txt" into result

Here check file accepts two arguments

Function arguments separators ; TO With (arg name)
Put the cosine of 34
Put the dot product of a TO b
Put the dot product with A THEN B THEN C THEN D ?
Make a Student called my student with name "bob" with age 24 with course Physics.
Make a vector called v with dimension 34 ; 45
Make a vector called v with args 10222,22222 ; dcopdjcpodj; pspdojcpsodj :
Aguments separated by ;
Put { 1 ; 2 ; 3; 4…100; 11} into some numbers — this defines a set
Put [1;2;3;4;5] into some numbers — defines a list or a vector if all numbers.
Put { 45 to 34 ; 23 to 24 ; "hello" to 23 ; "bob": 42 ; } — creates a hash map
Put the new person with name and age and

Named arguments with

Repetition

Repeat is the way to repeat something for example

```
Put 1 into count as counting number
Repeat five times
      add 1 to count
      put count
End repeat
```

Notice we used five not 5, which would have been ok two. All the numbers one, two three, four five six … ten exist as locked built in constants.
Notice that all the elements which are to be repeated are indented. Like Python and HyperTalk BFL indents to show the nested relationships.

In fact we didn't need to put count in because our old friend it is initialised in the loop.

```
Repeat one hundred times
      put it
End repeat
```

We tend to use repeat when looking at something for example

```
Repeat for each word in hello, world how are you?
      put it
End repeat
```

Will put each word of hello, world how are you each

```
Repeat for each letter of hello word how are you today.
      put it – it is now a letter
End repeat
```

You don't have to use repeat for each for just words. You can use sentences

```
Repeat for each sentence of hello word how are you today.
      put it – it is now a letter
End repeat
```

Or lines - which is quite common.

```
Repeat for each line of hello word how are you today.
      put it – it is now a letter
End repeat
```

```
Repeat for each noun of "this is a message said Bob"
      put it
End repeat
```

```
Repeat for each [word,letter,sentance,item] [ in|of ]  [text | list | list
generator ] [ in steps of number]
      – indented body
End repeat
```

Repeat does need to have to start from zero. Try this

```
Repeat from 1997 to 2017
      print " the year is " &&  it
End repeat
```

Nor does have to work in single steps

```
Repeat from 1997 to 2017 in steps of 4
      print " the Olympic year is " &&  it
End repeat
```

Note the compiler will not permit steps to work outside of the range so -4 would not be acceptable unless it was 2017 to 1996

If you don't want to use IT for your variable you can use with

```
Repeat with counter from 45 to 90
      put counter
End Repeat
```

```
Repeat with counter down from 90 to 45 in steps of 3
      put counter
End repeats
```

This isn't all repeat can do we will see more later.

Matching strings

Match any word from <<Expression>> —
Match (any letter 'a' to 'z' Or any letter 'A' to 'Z') followed by word "bob" from <<Expression>>
Match all that do not match (any Capital letter followed by lower case letter) followed by anything
Match one digit followed by one letter

Drawing

Lets do some drawing. There are two ways to draw in BFL - via the turtle and map coordinates. Lets start with turtle.

```
Turtle forward 4  – draw line
Turtle turn 90 – turn 90 degrees
Turtle forward 4 – draw line
```

When you do this a window should open to show you the results. Typing turtle is a pain so lets create an alias for Turtle

Put Turtle into T

```
T turn 90
T forward 4
T Turn 90
```

By default numbers are degrees but you can use radians
 T turn 3.414radiance.

```
Turtle reset
Turtle clear
```

We can ask a turtle what is beneath it.
Get turtle foot print colour

Defining your own actions, verbs or commands

We have already met how to create a verb.

```
To plus vat
      Get clipboard
      Put ( it plus 20% * it ) followed by "including VAT" onto the  clipboard
Finished  plus vat go back
```

FYI The term to was taken from LOGO.

Using it is simple

```
Plus vat
```

In technical parlance an action is a function, subroutine, co-routine, routine or procedure. We can use a action. The end at the end tells the function your done. You can also use the term 'END' which is shorter. You can leave the empty space below and the interactive compiler will put the code for you. If you haven't experienced the interactive compiler then think of it as your assistant. You should find that BFL will go back into your code and fix things for you. Quite often this is in the form of questions written as comments. These are intended to clarify things. You can use it to save typing. If the compiler can figure something out (like putting in End where you left a space.). The interactive compiler is how BFL bridges the brevity enjoyed by many programmers who don't like typing, with the clarity of English writing. Note you can switch style in the header by using configure style C, configure style normal, configure style poetic, configure style python.

Ways of ending an action verb (all equal

```
END
End NAME OF ACTION
Finished  NAME OF ACTION go back
Finished
Finished go back
Return – no indentation.
} –- will be replaced in normal style
– empty line no indentation. Will be replaced in normal style nnnn
```

For new programmers Finished go back is a good way to realise that the flow of control will leave the action and return to where ever it was called from (and so until it reaches the command line).

```
To plus vat
      Get clipboard
      Put ( it plus 20% * it ) followed by "including VAT" onto the  clipboard
End
```

What if we want to have a way of customising a function? For example perhaps we want to have a version of plus vat which takes how much VAT is currently. We could do it like this.

```
To plus vat parameter vat rate
      Get clipboard
put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

We need the key word parameter to identify the start of a parameter or vat rate rather than having a very long function name. Long names are harder to use and very long names are not record. You don't have to have the parameter on the same line as the verb. This gives us the ability to have ' as percentage' which forces the argument as the percentage.

```
To plus vat
      parameter vat rate as percentage
      Get clipboard
put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

Now we have a verb with a function we can use it.

```
Plus vat 20%
```

Unfortunately we now have to keep typing the parameter each time. While it's nice to have the flexibility of having a different rate it's a pain to have to keep typing it in. If only there was a way to getting BFL to type any missing parameters in with us. Fortunately we can

```
To plus vat
      param vat rate as percentage default to 20%
      Get clipboard
      put ( it plus vat rate * it )followed by "including VAT" onto the
clipboard
End
```

```
Plus vat – this works defaults to 20%
Plus vat 8% – this works
```

Currently we have a problem what if we type

```
Plus vat -20%
```

This will be calculated as much as everything else. Part of the process of developing a working piece of software is checking arguments for correctness.

```
To plus vat
      param vat rate as percentage default to 20% check vat rate > 0.01
 Get clipboard
 put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

Notice when this goes wrong it fails at the CALLERS site. The caller is responsible for calling parameters correctly.

```
Plus vat -20%
Something has gone badly wrong plus vat expects vat rate > 0.01
```

Note you can have multiple checks which is good practice. BFL reserves the right to expect the checks to make sense. For example

check vat rate >0.01
check vat rate <= 0.01

May cause BFL to complain about the injustice of it all.

Multiple descriptions arguments can be handled by adding double indentation.

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
 Get clipboard
 put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

Notice that check as a describe as message. This is a human readable message describing the thinking behind the assertion. Common checks include is not null, is not Nan.

Named parameters

```
To plus vat
      parameter how much as number
      optional parameter vat rate
            as percentage
            default to 40% — You MUST HAVE DEFAULT VALUE
      optional parameter isChildren called childrens
            as Question
            default to false — You MUST HAVE DEFAULT VALUE
End
```

Sometimes you need lots of optional parameters - for example when making a window. To help we can use optional named parameters here to use them we use With and look for

Put plus vat of 34 with childrens as YES

Put cosine 34 with degrees as YES
Put goonal 34 with grobmal as 34 * 34 fapal as 12 * 12

Action verb variables

What happens in vegas stays in vegas. Writeing a large program an become a pain. Partly because you run out of names for things. One of the dangers early programmers discovered is using the same variable over and over again for different purposes. To help with this it would be useful to have a way of using temporary or scratch variables which are secret - that is can only be seen or used within a function or action verb. You can these are called local variables. By default every variable you make within a action verb or function exists only within that funtion. That is it stops existing when the code is running.

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      global default rate  — must mention before using
  Get clipboard
  Put ( it plus vat rate * it ) into the basic rate
  put basic rate followed by "including VAT" into the  clipboard
End
```

Action verb: annotations

You can skip this if your keen to get on with making things. Programs do get bit and when you do organising things becomes a big problem. To help you there are special annotations we can have on a verb definitions. These are more for the software engineering end of things, the include documentation, comments, testing, bug tracking and optimisation.

Todo
Todo's let you leave notes in your code things you want to do and come back later. To dos unlike most of the other

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      todo ' split this into separate  function and verb. '
 Get clipboard
 put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

You can use this on your classes

```
 List all todods
```

This will list all the todos. Todo can be followed by urgent or low priority

Description

Description describes the verb in English terms.

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      description  this verb copies the argument rate to clipboard
 Get clipboard
 put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

You can use the description to generate documentation. Parameters can also have descriptions.

If you want to go beyond a single line you can use leave the line after description blank

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      description
              this verb copies the argument rate to clipboard
              this is another line. I can use *B* for *bold*
              I can refer to other functions via see 'with out'
              I can mention the author via *AUTHOR*
               for italics I can use -this-
              For pictures I can use *INCLUDE pathname*
              for head line __I USE ONE UNDERSCORE_
      end description – end description is optional.
 Get clipboard
 put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

If you want to know more about a function you can to this

```
Plus vat.help
Documentation for plus vat
```

```
 this verb copies the argument rate to clipboard
             this is another line. I can use *B* for *bold*
             I can refer to other functions via see 'with out'
             I can mention the author via *AUTHOR*
              for italics I can use -this-
             For pictures I can use *INCLUDE pathname*
             for head line __I USE ONE UNDERSCORE_
```

If you use description you can include example

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      description
              this verb copies the argument rate to clipboard
              this is another line. I can use *B* for *bold*
              I can refer to other functions via see 'with out'
              I can mention the author via *AUTHOR* automatically
               for italics I can use -this-
              For pictures I can use *INCLUDE pathname*
              for head line __I USE ONE UNDERSCORE_
              .STOP.USING.THIS.
              .OUT.OF.DATE.
      end description
      example
            put 100 into clipboard
            plus vat
            put paste
      end example
 Get clipboard
 put ( it plus vat rate * it )followed by "including VAT" onto the  clipboard
End
```

You can use example at the keyboard

```
Plus vat.example
            put 100 into clipboard
            plus vat
            put paste
```

If you include description and example then you can automatically generate documentation like this

```
Compile documentation for plus vat
Compile documentation for all
```

To test

A frequent augment to example is to test. This is code to test the correct operation of function. This can be either code or it can be reusing the example.

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      example and to test
            put 100 into clipboard
            plus vat
```

```
            put paste
      end example
 Get clipboard
```

If we say

```
Run self diagnostics
```

Or

```
Plus vat.self test
```

Then the code in the example is run. If the function aborts then this will be marked as failure. The example is not a strong test (but it's better than nothing which is the industry average). You can make the code more stringent like this

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      example and to test
            put 100 into clipboard
            plus vat
            put paste into it
            check it not empty – test 1
            check it = 100   – test 2
      end example
 Get clipboard
…
```

Tests can get involved. It is common to use an external test function known as a unit test. Note when compiling documentation the checks are not emitted.

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      example
            put 100 into clipboard
            plus vat
            put paste
      end example
      to test unit test plus vat
 Get clipboard
….
```

This basically refers the reader to a verb called test unit test plus vat. It's short so it fits on lne.

Reviewed

Code review is another method to make sure action verbs operate correctly. How to do code review is out of scope for this text. BFL helps to keep everything about a function together with the code. Experience has shown that in agile development situations this can allow quick refactoring of code. Reviewed is a less used description can be followed by either by or on.

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      reviewed by the code police  on 9/9/1999
```

```
 Get clipboard
```

BUG or Reported error
Tracking bugs is an important process. Again verbs can contain the bug description. This helps maintain a log of which bugs were reported and found in this code.

```
To plus vat
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01 describe as  'the vat rate is oddly low'
      Bug #30303 fixed.
      Reported error out standing
      Bug 'weridness ' still out standing
      Reported error #0432 fixed see check vat rate.
 Get clipboard

These become attributes of plus vat
Put plus vat's bug
Put plus vat's description
```

Functions

You have met some functions before - for example

```
Put "the time is" && the time
— puts 'the time is now 12:45
Put the date
Put "log  is" && the log of 45
Put "sine is " && sine( 45 )
```

The [function name]

Functions are a special type of verb. Functions have particular properties. Firstly functions can return items secondly functions can be 'pure' or tainted.

```
Function add vat
      param price
            as currency
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01
      returns new price
            as currency
Begin

End function
```

Returns is like param except it's something which is being returned. A return can have a type check and it can have a check in general. Checks can be used by the compiler to optimise functions. Checks are applied either at compile time or run time where appropriate. Checks may involve calling pure functions. For final operation run time checks can be disabled so speeding code up.

```
To swap
      param   lat as length
      param   long as length
```

```
      return  hoz as length
      return  vert as length
Begin
      hoz = long
      vert = lat
      return hoz ; vert
End
```

```
Put  swap( 23, 34 ) into  x ; y
```

You can have a variable which is both a parameter and returns

```
To add vat
      param price
            as currency
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01
      returns price – both input and out put
Begin

End
```

Notice that returns can also be checked. Entry

```
To add vat
      param price
            as currency
            – notice we don't check price is not empty
      param vat rate
            as percentage
            default to 20%
            check vat rate > 0.01
      returns price
            check price is not empty
            check price is > 0
Begin

End
```

You may have noticed that with out adding in checks we can identify problems. Here we have a more complex check

```
To add param x
      param y
            check  y >=  0
      returns result
            check result >= x
   Put  x + y  into result
Finish adding and return
```

Notice the check on result can be complex. As we see with pure functions later this allows the compiler to reason about functions and reason about functions of functions. If a compiler notices something it will complain

```
To something param x
      check x > 0
```

```
      returns result
            check result > 0
… code here.
End something

To other param x
      check x < 0
      returns result
            check result < 0
… code here.
End something

To combinedThing param x
      check x  is 0
      return result is 0
   Return something of ( other of x ) – compiler
End combinedThing
```

```
Error combinedThing will never ever work, sorry. It's like impossible. Have a
look at the code and check your self it's logically flawed. Believe me this is
for the best.
```

To square param x
        returns result
        check if x > 0 then result > x

.....

Pure functions

Functions can be pure. By pure we mean functions which always work the same way given the same arguments. Mathematical functions (like sin, cos, tan, log etc) are examples of pure functions. Some people think having lots of pure functions makes making a softer or one day can be used to build programs which could check programs and make sure they don't crash. Argument's about functional programming aside you can get a number of performance benefits. The primary advantage is you can copy and paste a pure function from one program into another with out problems. It's completely stand alone and will always work.

If we wanted to make a pure function this is how we would make one. All this does it forbid a number of operations which are regarded as in pure. For example saving to files, use of the clipboard or call insure functions.

```
Pure Function square
      param value
      returns result
      result = value * value
End function
```

The only practical benefit is the ability to use items like this

```
Pure Function square
      param value cache 40
      returns result
      result = value * value
End function
```

The cache allows 40 values of the argument to be stored. If the function were long and slow then the cached values could be checked before the long calculation. Being pure also lets the compiler be free to perform certain optimisations for example it could evaluate a constant at compile time so

```
Put  the square  of 2 into temp – same as pure 4 into temp
```

If two pure functions are used they can be evaluated and compiled and reevaluated. When a function is pure the compiler is allowed to go crazy and optimise a group of functions to death.

```
Put square ( 2 ) - square( 2 ) into temp – same as  put 0 into temp
```

Note if the argument has a check the check is applied to the variable at compile time. For example if you declare this.

```
Pure Function squarePositiveOnly
      param value cache 40
            check value > 0  "sorry only positive numbers can be square."
      returns result
      result = value * value
End function
```

The call like this

```
Put squarePositiveOnly ( -4 ) – this will cause an error at compile time.
```

Will cause a compile error as will this

```
Put squarePositiveOnly( square( 0 ) ) – both functions are pure.
```

General purpose functions

So far we have really encouraged you to make your functions as specific as possible.

```
Function do double
            parameter x  as number
            returns d as number
      add x to x – doubles by adding x to its self
      put x into d
      return d
End function
```

By default All BFL's functions are generic

We can use add on more then numbers - lists can be part of add. So how to we have a function which 'doubles' numbers and doubles list. Some systems use types but BFL uses something simpler. Type of.

```
Function do double
            parameter x
            returns d as same type as x – <-- look new thing!
      add x to x
      put x into d
      return d
End function

Function do double
            parameter x
            returns d as x's type  – <-- look new thing!
      add x to x
      put x into d
      return d
End function
```

You can use **type of** when you want to use the type of another variable. Now do double can double both number and lists. It will return what ever type it's passed in. Again this permits the logic system to check the integrity of system.

More on put

Put *expression*
Put *expression* **into** [chunk of] *container*
Put *expression* **after** [chunk of] *container*
Put *expression* **before** [chunk of] *container*

Chunk can be letter , word , item or lien
Put 235 after letter 4 of word 3 of line 2 of some text

Put expression into x of position — access like position.x = expression calls set.
Set the property of object to *expression* — position.x = expression calls set
Get the property of object — puts it into x calls get

If it is OK. Then
  Action
End if

Lists (single columns)

When you have more than one thing it's tempting to just number them like this
Person1
Person2
Person3

You will notice that if you put numbers at the end of something you can use a different syntax

```
Put 1 into person1
You can also use
Put "Bob" into person#1
Put person#1
```

Hash means 'number' so person#1 is person number 1. When ever you create a person ending with a number you are creating what is known as an array. Think of an array as a single row or single column in a spreadsheet.

NOTE FOR PROGRAMMERS OF OTHER LANGUAGES. You can use a number at the end of a name to indicate a list. In many programming languages you can create variables such as x1ss3dd322 . You can still do this in BFL but you are in true created a multi dimensional array or a tree. In general stick to the good programming policy of using clear variable names all in English.

When you use the # syntax you don't have to use a number you can use an expression for example

```
Put Bob. Into greeting1
put 1 into which as number
Put greeting number 1 — get the first item in the list
Put greeting#which — same as greeting number 1
Put greeting#( (34 -33) )  —
```

The is the same as greeting1 but we can use any item. Notice we can now put index into a loop.

```
Repeat with index from 1 to 20
      Put "Green Bottle " joined to index  into greeting#index
      Put greeting#index
End repeat
```

Fortunately repeat has a syntax for lists

```
Repeat with each item of greeting
      put it
End repeat
```

If you want to put a number of times in to an array you can like this each item is separated by semicolons.

```
Put ( 1;2;3;4;5;6;7;8;9;10 ) into myList
```

This is known as the 'compressed' list syntax.

Or like this

```
Put  'bob' ; 'Broklyn' ; 'Jenny'  ; 'Joe' ; 'Kalya'  into people as texts
```

For longer lists you can arrange vertically to do this use **put list** and then start each item on a separate line as a new item.

```
Put list
      bob.
      Broklyn.
      Jenny.
      Joe.
      Kalya.
Into people as texts.
```

Note Many languages use square bracket syntax like this people [1]. If you prefer this you can switch it on with the legacy or abbreviated control verb.

```
Use abbreviated
Put (1;2;3;4;4;5;6 ) into people
Put people [ 1]
Put 34 into people[1]
```

Another way to build up a list is with the add action.

Add 1 to **the list** people
Add 2 to people **as a list**

If you ignore **the list** or **as list** then BFL will assume you want to add 1 to each item of a list. All the math verbs work like this

Add 1 to people — gives (2;3;4;5;5;6;7)
Multiply *people* **by** 5 — gives (5;10;15;20;20;25;30)
Divide *people by 5*
Subtract

If the list is a list of strings this will join something
Add "€" to people ("€1"; "€2" etc..)

Lists of numbers are treated specially - see the section on vectors below

Now we have lists lets see what we can do.

Lists and repetition
What makes lists powerful is instead of repeating the same code a number times you can use a loop.

Before

```
Put people#1
Put people#2
Put people#3
Put people#4
Put people#5
Put people#6
```

```
Repeat with index from 1 to 6
  Put people#index
End repeat
```

This assumes your 100% sure how many items there are in the list

```
Repeat with index from 1 to length of people
      Put people#index
End repeat
```

Finally and most commonly of all you can use something which people call a for each loop.

After

```
Repeat with each item of people
   Put it –
End repeat
```

When you say repeat with each item of people, the loop puts which item your looking at into people. Notice there are two other words containing values **the previous** and **the next**. If you looking at the first item in a list **the previous** is either *empty* (null). IF your looking at the last item **the next** is empty (null). You should check for this. There is also **the index** which allows you to know which item your looking at.

Note lists have a number of methods (things they now how to do) we have seen one length of

Vector

List of numbers are special and when they are, they are called vectors. The only difference between a vector and list, is that as a list of numbers, vectors have more permitted operation are.

```
put  1 ;2;3;4;5;6;7;8;9;10 into v as vector
Put the sum of v
put  v.sum
Put the average of v
put  v.average
Tell v to average  – result in it
Put it
```

Vectors have a number of methods, both for fast processing and for statistics.

```
put  1 ;2;3;4;5;6;7;8;9;10 into v as vector
put  1 ;2;3;4;5;6;7;8;9;10 into v2 as vector
Put v.cosine distance(v2)
Put v.distance Between(v2)
Put the hypotenuse of ( v  ; v2 )
Put v.T Test Paired(v2) – gives YES
Put v.detailed T Test paired(v2 ) – gives 1  p value 0.001
```

Fast vectors

By default BFL uses BFN(big falcon numbers). Inside the computer there are representations which hold less detailed numbers. These a lot faster but they have problems - for example if you add 1 to the biggest fast number it will complain. You can't used these fast numbers for extensive cyptographic purposes. In traditional programming languages these are called things like INTEGER or INT and Floating point, Real or double and unsigned. You can naturally use them like any other number.

```
Put 1 into index as fast number
Put 1 into index as fast  integer — integer is fast any way
Put 1 into index as integer —
```

You probably won't notice the difference. The place you will notice the difference is when you have large numbers of them. If you find your self doing large numbers of complex calculations with vectors you might want to switch to fast vectors. Remember these are really really fast but inaccuacys can result
Put 0.1 into a as fast float
Put 0.2 into b as fast float
Put 0.3 into c as fast float
Put a + b = c — result no 0.1 + 0.2 does not equal 0.3 it's a fast float thing people live with

If you use fast before a vector you get some useful advantages. For example if available the system will use CUDA to run the operations on fast vectors on the CPU. Note that results may differ from those used with vector of numbers. You can always disable CUDA to force BFL to switch to on CPU emulation.

You can make sure everything is happening on CUDA by using with CUDA do.

```
put  1 ;2;3;4;5;6;7;8;9;10 into v as fast vector
put  1 ;2;3;4;5;6;7;8;9;10 into v2 as fast vector
With CUDA do
      Put v.cosine distance(v2)
      Put v.distance Between(v2)
      Put the hypotenuse of ( v  ; v2 )
      Put v.T Test Paired(v2) — gives YES
      Put v.detailed T Test paired(v2 ) — gives 1  p value 0.001
      Add 4 to v — done on GPU now.
End with
```

Now

Truth, Questions Booleans and Logic

When writing programs we often get the machine to changes what it is doing according to the result of a question. BFL defines two logical states YES and NO in traditional computing these are TRUE and FALSE or Zero and not zero You can use either even mix them when you want. YES means TRUE and FALSE means NO. BFL uses YES/NO because it better reflects the real world use of logic
For example

```
Put the engine is on — yes
```

Is less confusing than

```
Put the engine is on — true
```

By reducing confusion we eliminate the possibility of errors.

If you know that you are going to hold the result of a question (yes, no) then you need to say it is a logical variable

Put yes into the engine isOn as logic
Put yes into the engine isOn as boolean — opptiona.
Put yes into the engine isOn as ?

We frequently note this by using ? As a variable attedum.
Put yes into the engine isOn? — not there is no type
NOTE Question marks can be put at the end of any variable/container/jam jar. For example

```
Put 34 into my height? —  Is perfectly legitimate.
```

Note the word boolean or bool refers to Henry bool a victorian mathematician. Unless you know about the rich history of maths, logic and symbolic logic the word bool is meaning less. As such BLF encourages you to use the word logic so your code is readable by non programmers. Remember BFL encourages the many eyes and few fingers approach to code. Logic is one of those areas which nonprofessionals have a great deal of difficulty in both using and expressing, Particular machine terms. It is unfortunately also where the business most wants to Express itself. This is so common there is the industry term 'Business logic' which describes the important possibly existential aspects of a company's thinking. An example might clarify this is rather brief statement. If your company is creating the sales system they won't be worried about the choice of database, networking system, protocol, language etc. But they will need to state with some clarity that customers with four or more purchases will get a 10% discount. This discount be added to if they have a discount voucher etc. Is the definition of rules which are the businesses prerogative. Will take a bank they may have complex logic representing who they will and him will not lend money to. It is vital that this software implements the Bank's logic correctly, buy correctly we mean to the Bank's satisfaction of their original intention.

Clearly implementing these rules can often be the difference between the software the client customer wants and the software that fails. Such reviewing the logic of the program when it affects customer is an important area, and one which BFL seeks to excel. If you show any of your codes to a client it is the logic that is most likely to be scrutinised, it is well worth the effort to get this toy level where external scrutiny is viable.

We have discovered the if statement already. Let's look at some more details.

Let's imagine a dangerous fairground ride. If you are greater than 1.6 m tall you can ride on the ride.

Lets start with the most clear and poetic way of coding.

```
Put 1.5m into my height

Is my height  > 1.6m?  — - result put into it.
If it is YES then
  Put You can ride.
End if
```

This begins with the question Is my height > 1.6m? If the statement begins with IS then it's assumed to be like GET.
The next statement is a frequently used one called IF.
This takes the value in IT and if it is YES then calls the block.

```
Put 1.5m into my height

If my height  > 1.6m then
  Put you can ride.
End if
```

You want to be very clear you could write it like this.

```
Put 1.5m into my height

If my height  greater than 1.6m then
  Say you can ride.
End if
```

If you have to be under 1.m to enter the play pit you would have this

```
If my height < 1m then
  Say you can enter the ball pit.
End if
```

Or

```
If my height less than 1m then
  Say you can enter the ball pit.
End if
```

Or

```
If my height is less than 1m then
  Say you can enter the ball pit.
End if
```

EQUAL

Some times you want to say if something is equal

```
If my height = 1m then
      Say you can enter the ball pit, but not next year.
End if
```

Or

```
If my height is equal to 1m then
      say Say you can enter the ball pit, but not next year.
End if
```

Or

```
If my height equal 1m then
      say Say you can enter the ball pit, but not next year.
End if
```

Or

```
If my height equals 1m then
      say Say you can enter the ball pit but not next year.
End if
```

But

```
If my height is 1m then
      say Say you can enter the ball pit but not next year.
End if
```

Sounds better.

Equals, greater than and less than all operate on numbers , texts, it also works on lists.

The problem is you often have errors in measurement what happens if the child is 1.0001m tall or 0.99999?

```
If my height almost equals 1m then
      say Say you can enter the ball pit but not next year.
End if
Or we can use the unocde symbol
If my height ≅  1m then
      say Say you can enter the ball pit but not next year.
End if
```

By default *almost* means within 1% of the other. So 0.99m to 1.1 you can use with almost at 5%

```
With almost at 5%
      If my height almost equals 1m then
      say Say you can enter the ball pit but not next year.
      End if
End with
```

Note if you EQUALS within an almost block the compiler will put this in your code.

```
With almost at 5%
      If my height equals 1m then  — DAVE THERE IS NO ALMOST ARE YOU SURE ?
      say Say you can enter the ball pit but not next year.
      End if
End with
```

To say yes use exactly equal

```
With almost at 5%
      If my height exactly equals 1m then
            say Say you can enter the ball pit but not next year.
      End if
End with — DAVE I'M NOT SURE ABOUT THE USE OF WITH ALMOST HERE.
```

The compiler is free to insert sarcastic comments into your code. This is part of the interactive compilation process. You'll notice the compiler will put observations and suggestions into your code. Sometimes this can be used to reduce typing. for example if you forget to put something the compiler what Adding the code for you rather than stopping the compilation process.
You type

```
      If my height exactly equals 1m
            say Say you can enter the ball pit but not next year.
      End if
```

The machine might add the then

```
      If my height exactly equals 1m then
            say Say you can enter the ball pit but not next year.
      End if
```

Actually most of the additions are silently done the compiler will let you know if it changed something - this is inorder to bring it to you attention supposed you had used the one line version

```
If my height exactly equals 1m turn OK to enter on
```

The compiler might

```
If my height exactly equals 1m then turn OK to enter on — Dave I added the then,you're welcome.
```

There are number verbs to control this. Sarcastic comments can get intensive if the logic is wrong. The machine will always let you do what you want.

```
If my height is greater then 1m and my height is less than 1m then
   put "hello" — DUDE,LIKE THIS IS NEVER GOING TO HAPPEN.
               — IT'S LOGICALLY IMPOSSIBLE. TRY UNREACHABLE
End if
```

Equals for text

Note exactly and almost also operates for text. Ignore the auto generated comment for now.

```
If "BOB" almost equals "bob" then
 put "hello bob"— DUDE, this will aways happen.Why the if statement, life form?
End if
```

```
If "böb" almost equals "bob" then
 put "hello bob"– DUDE, this will aways happen.Why the if statement?
End if
```

If this is not what you want then use exactly

```
If "böb" exactly equals "bob" then
 put "hello bob"– HEY, this will NEVER happen.Why the if statement?
End if
```

There is also

```
If "bob" sounds like "Burb" then
```

This is useful when dealing with names. It uses the soundex alogirthum like SQL. It's not amazing but it's Ok for most uses.

If you want to use another algorhutm you can configure with a with

```
With sounds as myfunction do
 If "bob" sounds like … then
End with
```

Note about The logic check comments. This can be quite helpful if you have done something complex. For example Cos is a pure function and the compiler can look at cos of a constant and figure it out as 1. 1 is never going to be > than 1 so.

```
If cos(0) > 1 then
      put "hello" – DUDE,LIKE THIS IS NEVER GOING TO HAPPEN.
              – IT'S LOGICALLY IMPOSSIBLE.
End if
```

More complex logic is also possible

```
If  a > 1 and b < 1  then
    … more code here there is put something to a and only pure functions called
    If a > 1 then
      Put "what" – DUDE, this will aways happen.Why the if statement?
    End if
    /// more code
End if
```

The compiler will also add suggestions

```
If a > 1 then
      If b < 1 then  – why not use if a > 1 and b < 1 ?
      End if
End if
```

You can control all of this with

```
Turn BLF code police off
```

Which will be acknowledged with

```
Turn BLF code police off – I'm there for you when you need me.
```

Or

```
Turn BLF code police off – I know when I'm not wanted Dave.
```

BTW Turn is another built in verb. It only handles logic variables or flags. Like switch or flip. Turn changes compiler variables but can handle both your logic variables.

```
Turn BFL optimisation on
Turn  BFL auto correct off

Put  YES  into is searching  as logic  –
Turn is searching on  –- is searching  yes. Puts yes in it
```

```
Switch is searching off – is search no
Flip is searching – if on is off,  if is off is on.  Puts new state in it
```

Notice Turn, Flip and Switch is thread safe and atomic - meaning if you have multiple processors you can guarantee nothing is going change this during the 'turn' statements. Note there is a state between a constant and a variable called a compile flag. This lets you change the state of the compiler if necessary. Turn is one of the few verbs you can use in code between methods.

Turn BFL compile off
— this code is ignored
Trun BFL compile on

STRINGS and MATCHING

The verb find is very useful it's a function
Put find("ello" in "Hello") into location and the text and found
If found then
	say "I found {the text} at {location}"
End if

What if we want to find each word

Put   WORD into pattern
Put find(pattern in "Hello") into location and the text and found
If found then
	say "I found {the text} at {location}"
End if

This can be more complex. For example supposed you want to make sure someone has typed in a telephone number
Put the pattern 3 digits and space and 3 digits and comma and 3 digits into my pattern
What about international codes
Put the pattern > 2 digits a space and 3 digits and comma and 3 digits into my pattern
Put the pattern   <1 letter and zero or more (letters or digits) into my pattern
Letters in a patter are them selves.
Put the pattern   ('H' or 'h') and "ello" into my pattern
Put the pattern   any of 'a' ; 'e' ; 'i' , 'o' , 'u' into vowels
— patterns can include other patterns
Put the pattern   zero or more letter follow by pattern vowels

Put the pattern "http://" followed by many (letters or digits) and period and one or more (letters or digits) and period and one of 'com' ; 'org' , 'net ' , 'edu'

Or the pattern "more than one (letters or digits) and two (period followed by more than one (letters or digits))

More than one <SOMETHING>
Possebly
At least one = 0 or more
A = 1
4   — exactly 4
4 or more
>= 4
Digit
Letter
'L
 Of (list)
Followed by
OR (group)
NOT

IF and else

So far we have seen what happens when you flip an if statemen

More groups of booleans

If a and b then
If a or b then
If not a then
If a exclusive or b then
If

Binary logic

Machine word is a new number type. Its the equivalent of **unsigned int** in languages like C and C++. It is designed to reflect the low level hardware on the machine. There are a number of fixed types

Machine 16 bit word
Machine 32 bit word
Machine 64 bit word
Machine 8 bit word — Byte
Machine Byte
Machine 1 bit word — bit
Machine word — either 16,32 or 64 depending on the specific hardware your on can be 63 bits

```
Put 45 into a as machine word
Put 43 into b as machine word
```

Special infix operators

```
Put 45 bit and b
Put 46 bit or b
Put 989 bit xor b
Put bit flip b
Bit  bit not  b
Put 45 rotate left  5
Put  45 rotate right 2
Put  hex 02033222 roll left 3  - - fill
```

Like many numbers there are different units like

```
Put HEX FACE
Put HEX C3DE

Put OCTAL 0343
Put OCT 0343
```

```
Put BIN 10100101010
```

Convert

Convert handles the conversion of one thing into another if possible for example

```
Convert "3043" into  number  – number 3034 is now in it
Convert 343.22 into integer  – 343 is now in it
Convert 343.22m into inches –
Convert $2000 into £ – error use function.
But you can say
Convert $200 into number – 200  this is called a down cast, the number is 200
convert it to £
```

Wich is about as smart as saying
Convert (convert $200 to number) to inch
You can do this on complex things such as list
Convert [1;2;3;4] into text — '1;2;3;4'

If something cannot be converted you will get empty in it.

Convert [1;2;3;4;5] into number

```
Put set  { 1;2;3;4;5;6;7;8;9;10}   into mySet
Convert mySet to list – it contains mySet as list

Convert mySet to hashSet –
```

Sets

There are other specialised lists. Sets come to mind a set is like a list except an item can only be in a set once.

```
Put set  1;2;3;4;5;6;7;8;9;10  into mySet
```

Internally Sets can be implemented in different ways. Making a set in different ways are easy.

```
Put hash set 1;2;3;4;5;6;7;8;9;10  into mySet
Put list set 1;2;3;4;5;6;7;8;9;10  into mySet
Put tree set 1;2;3;4;5;6;7;8;9;10  into mySet
```

Internal representations optimise certain times of processing

Once you have multiple sets you can operate on them.

```
Put mySet ∩ otherSet  – do intersection
Put mySet ∪ otherSet   – do union
```

Dictionaries

Along with lists it's useful to use one thing to find another. These a have a number of names, hashmaps, maps and BLF use dictionaries

Put

Tables

Tables are ways that BFL remembers information. Tables are an extension of lists. They are very like a simple spreadsheet

```
There is a table called stock kept in SQL
        columns
                product name as string
                product prices  as currency
                product number as fast integer
        check
                table is not null
                table exists
                table is writeable
                count is > 0
To updateStock
        parameter what as string
                what is not empty with

End table


Tell each item of tableName which column  name is not empty to put name

Tell each row of tableName to
        move it up by 5 — 'it' is the row it understands up by 5
End tell

Or put 1 into item 1 of stuff
```

Making your own types (classes)

In BFL tables are a good way to store information but sometimes things need to be more complex. BFL provides these under the official title of a class.

In a file called XXXX.interface wrote this

```
The class XXXX extends YYYY
        property x is a number  with set get and default 34
End class
```

Property allows you to quickly create an attribute (like a column in table). Set and get create simple methods. Notice that you can have a *virtual* property i.e one which is computed.

```
Make a new XXXX — put into 'it'
 Put a new XXXX into bob as XXXX

Put bob.x — returns 34
Bob.x = 40  — calles set x
Put bob.x  * 10  — returns 400
```

Forget bob — if no one remembers bob then bob is tidied up i.e deleted. Bob is now empty.
Check bob is **forgotten** — makes sure that bob is free to be tidyed up. No one else should know me.
Disavow bob — forget bob and check that bob is forgotten

On *tidied*
    disavow my name — name is property contains string its mine no one else should have it.
    disavow my address — address is a property contains type address.
End tidied

Notice we used **on** not **to** because tidied is defined in the super thing (base class) . It's also an event you can optionally handle.

For more complex classes it's common to put what are called 'methods' or actions. These are verbs which belong to the thing you are making. They hide in second file called XXXX.BFL

[*This is the*] *Implementation* of XXXX

End [*Implementation]*