BEL
A .NET Computational Package Written in Zonnon
Part I of the Users Guide

Alan D. Freed

Clifford H. Spicer Chair in Engineering
College of Science, Engineering & Technology
Saginaw Valley State University
202 Pioneer Hall, 7400 Bay Road
University Center, MI 48710
E-Mail: `adfreed@svsu.edu`

November 12, 2013

Abstract

This document describes the core set of modules belonging to the BEL software library written in Zonnon for the .NET and Mono frameworks. BEL provides a simple, computational, programming interface for the purpose of assisting in rapid program development (especially in the area of computational continuum mechanics) by leveraging the wide diversity of existing .NET and Mono libraries with the simple logical constructs of the Zonnon programming language. BEL's interfaces for its core modules are cataloged in the appendices.

1 Introduction

BEL is an acronym taken for my research laboratory: The Biological Engineering Laboratory at Saginaw Valley State University. What BEL is and what it provides are quite different from what one might expect, given this affiliation. BEL arose out of the author's desire to interface with the .NET and Mono Frameworks for the purpose of writing computational software programs in the Zonnon programming language [1, 2].

Zonnon is the most recent descendant from the family tree Euler/Algol/Pascal/Modula/Oberon of programming languages developed by Profs. Niklaus Wirth and Jürg Gutknecht from the Computer Systems Institute at ETH Zurich (Eidgenössische Technische Hochschule, Zentrum, i.e., the Swiss Federal Institute of Technology, Zurich). Zonnon was created specifically for .NET, with compiler versions available for both the .NET and Mono Frameworks. Sponsored by Xamarin, Mono is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime. The Zonnon compiler targets the .NET Framework 2.0.

The Zonnon compiler, documentation, example programs, and even BEL itself, are free to download from `http://www.zonnon.ethz.ch`. The home page for Microsoft's .NET Framework is at `http://www.microsoft.com/net`. Mono's is at `http://www.mono-project.com/Main_Page`.

1.1 Licensing

Licensing of the BEL library, both its software and documentation, is addressed in the *Licensing of the Software and its Documentation* part to this user guide.

1.1.1 NPlot

NPlot is the .NET graphics engine used by BEL for creating plots. The NPlot home page is: `http://netcontrols.org/nplot/wiki/`. NPlot is released under the terms of a 3-clause-BSD license. Specifically:

1.2 Version

This document corresponds to software version 4.2 of BEL, November 12, 2013. Zonnon and NPlot target the .NET Framework 2. The version number is exported from module Bel.Version as the string variable `version`.

1.3 Required Libraries

Two external libraries are required to compile BEL; they are: System.Drawing.dll and NPlot.dll. NPlot has different libraries for .NET and mono, because System.Drawing.Drawing2D is not implemented in the mono framework.

1.4 Known Issues

None

1.5 Feedback

I consider this to be a living document. If there is some aspect that is wanting, or if you have corrections and/or additions that you believe would benefit other users of BEL, please forward them to `adfreed@svsu.edu`.

2 Design Philosophy

The original motivation behind my writing BEL was a need to economize on the number of numeric types available so as to avoid an explosion in the number of extended higher-level types to be created. Recent changes in the language/compiler have drastically reduced the potential for type explosion. Consequently, major changes were introduced into vs. 4.0 from it predecessors. BEL is now more a collection of libraries than it is a miniature framework. BEL is now leaner. The author continues to be guided by WIRTH's design paradigm: **Seek simplicity!**

A conscious design decision was to use *value* (or static) types as wrappers around *ref* (or dynamic pointer) types whenever possible. There are applications where this is not desirable, e.g., nodes in data structures lend themselves nicely to pointer types so that linkages can be created, broken and recreated, as needed, without the need to move their data in memory. For most computational needs, however, static types seem to work best, so as to avoid unwanted side effects. Yet, data structures like arrays and matrices are most useful when their structures are dynamically created. BEL tries to balance the best of both these worlds by appearing to the user as a collection of static types, while hiding and managing their dynamic data structures internally. The nuisance of creating dynamic types, to the extent that is possible, has been removed from the user and relegated to the BEL system.

The core set of modules that define BEL include input/output (IO), a graphing capability, a set of array/matrix types, and basic data structures.

The definition modules for various types exported by BEL are given in App. B. The software interfaces to the core modules of BEL are provided in App. C–F.

3 Input/Output

There are a great number of .NET modules that deal with file IO. The author has economized on three file types: log, data, and text files; and two graphic types: curves and plots. All files created are written into the subdirectory `<executable>/iofiles` right below where your program's executable file resides. BEL will automatically create the `iofiles` directory if it does not already exist.

3.1 Log File

Zonnon does not have a construct for throwing a runtime exception, so a logging capability was created where messages can be written that could prove useful to a programmer who is trying to debug their code. This log file is written into file `<executable>/iofiles/logFile.txt`. If a log file already exists, the old file will be copied to `last_logFile.txt`, and a new file `logFile.txt` will then be opened. If there was an existing old log file, it will be deleted before your prior log file is renamed to `last_logFile.txt`.

This log file is automatically created when module `Bel.Log` is first loaded into memory. There are three procedures that a programmer may call: `Message, WarningMessage`, and `ErrorMessage`. A `Message` writes a string into the log file, while a `WarningMessage` or an `ErrorMessage` writes predefined messages into the file, and a string that locates where the message was issued. The scripts for predefined messages reside in module `Bel.Log` and are listed in App. A. The difference between a warning and an error message is that runtime execution continues whenever a warning is issued; whereas, the program terminates immediately after an error message has been logged to file.

The last line in each executable file that you write should be the command that closes the log file, i.e., `Bel.Log.Close`.

3.2 Data Files

Data files are used to write/read data to/from a file on your hard disk in binary format. They are encoded in UTF-16 unicode format. A writer created with `Bel.DataFiles.OpenWriter` takes a file name as a string and returns a .NET framework writer of type `System.IO.BinaryWriter`. The file name that you supply will be stripped of any path information and/or extension you gave it. It will be given the file extension `.dat` and placed under the directory `<executable>/iofiles` along side the log file. When you are done writing to the file, close it by calling the procedure `CloseWriter`. The corresponding `OpenReader` and `CloseReader` procedures are to be called when it is time to read in data from a file.

The .NET binary reader and writer objects that are supplied by these procedures are, in fact, the arguments to be sent to the `Load` and `Store` methods that utilizer's of the `Bel.Object` definition must implement.

3.3 Text Files

Text files mirror data files in all aspects, except that these files are human-readable text files instead of machine-readable binary files. They are encoded in UTF-16 unicode format and given a `.txt` extension. **Note**: if you try to view one of these files in an editor that does not support full unicode, then it will look like it is a binary file.

The programmer has more control over how a text file looks or is constructed, versus a binary data file, via the various methods that belong to the .NET classes of `System.IO.StreamWriter` and `System.IO.StreamReader`, instances of which are supplied by the `Bel.TextFiles` procedures `OpenWriter` and `OpenReader`. The methods belonging to these .NET classes can be found on their documentation web pages.

3.4 2D Graphical Curves

Three enumerated types exist to aid in specializing any given data set in a graph; they are: `Color`, `Dimension`, and `Marker`. Type `Color` can have one of nine values: `black`, `blue`, `gray`, `green`, `orange`, `pink`, `purple`, `red`, and `yellow`. Type `Dimension` can have one of five values: `tiny`, `small`, `medium`, `large`, and `huge`. And type `Marker` can have one of eleven values: `circle`, `cross`, `diamond`, `flagDown`, `flagUp`, `none`, `plus`, `square`, `triangle`, `triangleDown`, and `triangleUp`.

Instances of type`Bel.Curves.Attributes` provide the characteristics that belong to a given curve in a plot. Attributes `color`, `dimension`, and `mark` belong to the enumerations `Color`, `Dimension`, and `Marker`. Specifying the string variable `label` provides the name of the curve as it is to appear in the legend. Markers `circle`, `square`, and `triangle` can be filled (`filled` = ***true***) or unfilled (`filled` = ***false***), as can the vertical bars in a histogram. These bars can either be centered at their associated x value (`centered` = ***true***) or not (`centered` = ***false***), as can the horizontal steps in a step plot. Procedure `Initialize` sets: `centered` ← ***true***, `color` ← `black`, `dimension` ← `medium`, `filled` ← ***false***, `label` to the empty string, and `mark` ← `none`.

Two types of data sets are considered for plotting. Both contain the `Bel.Curves.Attributes` data in field `attributes`. Instances of the type `Bel.Curves.DataY` provide the ordinate (y) values, with the abscissa (x) values being assigned sequential values of 1, 2, 3, ..., assigned by the row position of their ordinates. Procedure `Assign` converts Zonnon arrays into .NET arrays that NPlot understands and can use, and assigns them to the variable `data`. Likewise, instances of type `Bel.Curves.DataXY` provide the ordinate (y) and abscissa (x) values to be plotted. Procedures `AssignX` and `AssignY` convert Zonnon arrays into .NET arrays that NPlot understands and can use, and assigns them to the `xData` and `yData` variables. Procedure `Initialize` initializes the attributes and sets the data vectors to ***nil*** for both types.

Six procedures are provided in module `Bel.Curves` for constructing various curves for plots. `LineCurveY` and `LineCurveXY` produce line curves, accepting data belonging to `DataY` and `DataXY`, respectively. `PointCurveY` and `PointCurveXY` produce point curves, accepting data belonging to `DataY` and `DataXY`, respectively. And StepCurve produces a step curve for data belonging to type `DataY`. These five functions have a like fingerprint. They accept a data set describing the curve and they return an instance of the associated curve as an NPlot object. The sixth procedure has a different interface; it is for plotting histograms. Procedure `Histogram` accepts two inputs: `histogramData` is an instance of `DataY` whose attributes describe the appearance of the histogram, while `normalCurve` is an instance of `Attributes` describing how the bell curve is to be drawn. Procedure `Histogram` provides five outputs: `mean`, `median`, and `stdDev` provide the sample mean, median, and sample standard deviation of the data in the histogram, which are also used in constructing the bell curve, while variables `histogram` and `normalPlot` provide NPlot objects for the histogram and bell-curve to be drawn in a plot.

Noticeably missing is the ability to draw dashed and dotted curves. This capability requires library `System.Drawing.Drawing2D`, which is not part of the mono distribution of .NET. Consequently, this feature is not included here so that BEL can be used in both the .NET and mono frameworks.

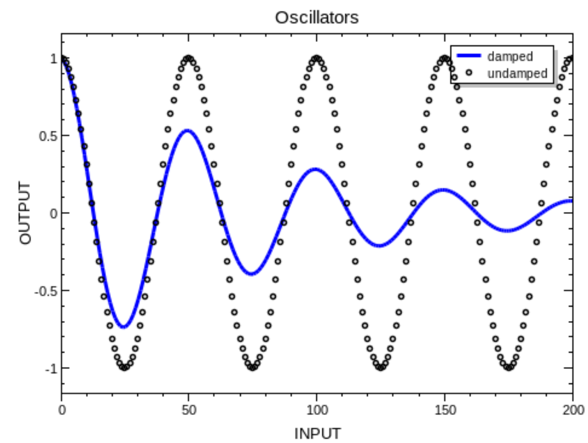

Figure 1: Graph created by the software distributed in the `testPlot` directory in BEL's distribution folder.

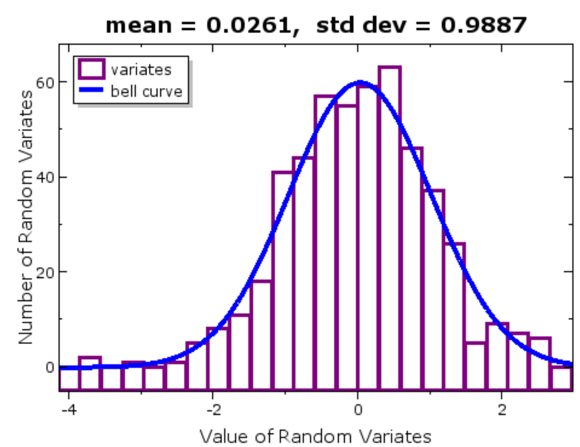

Figure 2: Graph created by the software distributed in the `testPlot` directory in BEL's distribution folder.

3.5 2D Graphical Plots

NPlot is the underlying graphics engine that is used in `Bel.Plots`. After one has created a collection of curves of types `NPlot.LinePlot`, `NPlot.PointPlot`, `NPlot.StepPlot`, and/or `NPlot.HistogramPlot`, which are supplied by the various procedures of `Bel.Curves`, one can create a `Bel.Plots.Plot`. Type `Plot` has ten methods to help you create your figures.

`Bel.Plots.Plot.Create` supplies the blank canvas upon which your graph will be drawn. It has two argument, the width and height of your graph in `pixels`. 500 pixels for the width and 400 pixels for the height were assigned in the test code. (The outputs shown in Figs. 1 & 2 have been scaled by LaTeX.)

`Bel.Plots.Plot.AddAxes` writes axes unto the graph. Automatic scaling is turned on and, at present, cannot be turned off. There are four arguments for this procedure. The first two are ***`boolean`*** valued switches. When ***`true`*** they make their respective axis logarithmic; otherwise, that axis will be taken to be linear. The last two are ***`string`***s that assign a label to each axis. The first argument of each pairing is for the x-axis, while the second is for the y-axis.

`Bel.Plots.Plot.AddLegend` is the procedure that will write a legend onto your graph. It can be placed at one of the four inside corners of the plot, positioned according to the passed value of enumeration type `Location` which can have values: `lowerLeft`, `lowerRight`, `upperLeft` and `upperRight`, the latter two being shown in Figs. 1 & 2, or it can be placed outside the plot to the upper-right by passing `Location.outside`. A legend will be displayed only if this command is called.

`Bel.Plots.Plot.AddTitle` writes a title above the graph. It is passed as a ***string*** argument.

`Bel.Plots.Plot.AddLineCurve` draws a line curve on the plot's canvas. These curves are supplied by either `Bel.Curves.LineCurveY` or `Bel.Curves.LineCurveXY`. Line curves are drawn as straight segmented lines.

`Bel.Plots.Plot.AddPointCurve` draws a point curve on the plot's canvas. These curves are supplied by either `Bel.Curves.PointCurveY` or `Bel.Curves.PointCurveXY`.

`Bel.Plots.Plot.AddStepCurve` draws a step curve on the plot's canvas. This curve type is supplied by `Bel.Curves.StepCurve`.

`Bel.Plots.Plot.AddHistogram` draws a histogram on the plot's canvas. This curve type is supplied by `Bel.Curves.Histogram`.

The order in which curves are attached to a plot is the order that they are displayed in the legend.

`Bel.Plots.Plot.AddText` writes a string passed as `text` to the plot positioned `atX` and `atY`, which hold world coordinate values, i.e., coordinates that align with those of the x- and y-axes.

Note: You must call `AddAxes` before adding any curves to the plot via `AddLineCurve`, `AddPointCurve`, `AddStepCurve`, or `AddHistogram`. You must call `AddText` after the curves have been added to the plot.

A call to `Bel.Plots.Plot.Save` will write your finished graph to a bitmap file given the name that you supplied. It will have a `.bmp` extension, and will be written into the `iofiles` subdirectory below where your executable file resides, in standard BEL manner.

NPlot has many capabilities that are not utilized in BEL. Some of these enhancements are expected to be adopted in future releases.

4 Numeric Types

Module `Bel.Types` exports six numeric constants; they are: `MaximumInteger` is the largest possible value of type ***integer***, `Epsilon` is machine epsilon, `MaximumReal` is the largest possible value of type ***real*** that retains full precision in its digits, `MinimumReal` is the smallest positive value of type ***real*** that retains full precision in its digits, `NaN` represents not a number, `NegativeInfinity` handles values less than -`MaximumReal` while `PositiveInfinity` handles values greater than `MaximumReal`.[1]

Module `Bel.Types` provides a number of definitions for a variety of array and matrix types; specifically: `BooleanVector, IntegerVector, RealVector;` and `BooleanMatrix, IntegerMatrix, RealMatrix`; all of which have *math* attributes. The syntax of these additional operations mimics that of the popular MatLab[2] development environment; specifically, assignments can be made much more simply via

$$
\begin{array}{ll}
\text{array} & \text{x} := [x_1, x_2, \ldots, x_n] \\
\text{matrix} & \text{A} := [[A_{11}, A_{12}, \ldots, A_{1n}], \\
& \qquad [A_{21}, A_{22}, \ldots, A_{2n}], \\
& \qquad \vdots \\
& \qquad [A_{m1}, A_{m2}, \ldots, A_{mn}]]
\end{array}
$$

Element-wise operations include:

unary	"−"	negation
	"∼"	inversion
binary	".∗"	multiply
	"./"	division

1. `Epsilon`, `MaxValue`, and `MinValue` exported by `System.Double` are distinct from BEL's constants `Epsilon`, `MaximumReal`, and `MinimumReal`.

2. MatLab is a powerful but pricy mathematical programming environment that is marketed by MathWorks `www.mathworks.com/products/matlab`.

".<"	less than
".<="	less than or equal
".>"	greater than
".>="	greater than or equal
".="	equal
".#"	not equal

Array and matrix operations include:

unary	"!"	transpose
binary	"+"	addition
	"−"	subtraction
	"*"	multiplication via index contraction
	"/"	right division
		`X := B/A` solves $\mathrm{X} \cdot \mathrm{A} = \mathrm{B}$ for X
		`x := b/A` solves $\mathrm{x}^T \mathrm{A} = \mathrm{b}$ for x
	"\"	left division
		`X := A\B` solves $\mathrm{A} \cdot \mathrm{X} = \mathrm{B}$ for X
		`x := A\b` solves $\mathrm{Ax} = \mathrm{b}$ for x

Math arrays and matrices have other operators/ functions that can be used by them, but the above are the most common and useful.

This module provides a variety of useful procedures. `IsFinite`, `IsInfinite`, and `IsNaN` provide tests that check the value held by a *real*. Procedures `BooleanToString`, `IntegerToString`, `NumberToString`, and `RealToString` print out the values held by their arguments. `NumberToString` prints out reals in decimal notation, while `RealToString` prints out reals in scientific notation, both being able to be written to a specified number of significant digits. To parse these fields, one can call procedures `StringToBoolean`, `StringToInteger`, and `StringToReal`.

For writing the base Zonnon types to file, one can use the `Bel.Types` procedures `StoreString`, `StoreBoolean`, `StoreInteger`, `StoreReal`, `StoreBooleanVector`, `StoreIntegerVector`, `StoreRealVector`, `StoreBooleanMatrix`, `StoreIntegerMatrix`, and `StoreRealMatrix`, which can in turn be read from file via their counterpart procedures `LoadString`, `LoadBoolean`, `LoadInteger`, `LoadReal`, `LoadBooleanVector`, `LoadIntegerVector`, `LoadRealVector`, `LoadBooleanMatrix`, `LoadIntegerMatrix`, and `LoadRealMatrix`.

4.1 Series

`Bel.Series` exports three kinds of series that are either infinite series summed to some convergence criteria, or that are truncated series. These are: continued fractions, power series, and rational power series. A continued fraction has the form

$$y = b_0(x) + \cfrac{a_1(x)}{b_1(x) + \cfrac{a_2(x)}{b_2(x) + \cfrac{a_3(x)}{b_3(x) + \cdots}}}$$

where coefficients $a_i(x)$ and $b_i(x)$ can be functions of x. In contrast, a truncated continued fraction has the form

$$y = b_0 + \cfrac{a_1 x}{b_1 + \cfrac{a_2 x}{b_2 + \cfrac{a_3 x}{\cdots + \cfrac{a_{n-1} x}{b_{n-1} + \cfrac{a_n x}{b_n}}}}}$$

where here the coefficients a_i and b_i are constants. A power series has the form

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots ,$$

while a rational power series has the form

$$y = \frac{a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots}{b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \cdots}$$

wherein a_i and b_i are constants. These kinds of series are often used to define or approximate mathematical functions in computer applications [3].

The procedures that compute a series to convergence require the user to write instances of procedure type `Bel.Series.GetCoef`, which in turn is to be passed to the procedures in this module so that a series' coefficients can be determined. Such a function will have three arguments. The first sends an integer that specifies the index for which the coefficient is to be returned. The second sends the argument of the function being evaluated. And the third is updated by the procedure; it being a boolean flag to force termination of a summation prior to convergence (e.g., when it is taking too long). Such a procedure returns a coefficient for the series, e.g., a_2.

4.2 Math

Module Bel.Math exports two, fundamental, mathematical constants; they being: $E = e^1 = e = 2.71828182845905$ and $PI = \pi = 3.14159265358979$, when expressed to machine accuracy. Math also exports the more common math functions that one will likely have need of:

Random returns a random number in [0,1].

Max returns the greater of its two arguments.

Min returns the lesser of its two arguments.

Ceiling returns the smallest integer value that is greater than or equal to its argument.

Floor returns the greatest integer value that is less than or equal to its argument.

Round returns the integer value that is closest to its argument.

Abs returns the absolute value of its argument.

Sign returns one if its argument is greater than zero, minus one if its argument is less than zero, and zero if it is zero.

Sqrt returns the square root of its argument.

Power returns the first argument, x, raised to the power of the second argument, y, viz., x^y.

Pythag returns the Euclidean distance between two numbers, i.e., $\sqrt{x^2 + y^2}$.

Log returns the base 10 logarithm of its argument.

Ln returns the natural logarithm of its argument.

Exp returns the exponential of its argument.

Sin returns the sine of its argument.

Cos returns the cosine of its argument.

Tan returns the tangent of its argument.

ArcSin returns the inverse sine of its argument.

ArcCos returns the inverse cosine of its argument.

ArcTan returns the inverse tangent of its argument.

ArcTan2 returns the quadrant-correct inverse tangent of its two arguments, i.e., $\tan^{-1}(y/x)$, where rise y is its first argument, and run x is its second.

Sinh returns the hyperbolic sine of its argument.

Cosh returns the hyperbolic cosine of its argument.

Tanh returns the hyperbolic tangent of its argument.

ArcSinh returns the inverse hyperbolic sine of its argument.

ArcCosh returns the inverse hyperbolic cosine of its argument.

ArcTanh returns the inverse hyperbolic tangent of its argument.

5 Definitions

A Zonnon **definition** is a contractual interface that all who implement it must obey. One definition can **refine** another, but not in a circular manner. Zonnon uses a compositional inheritance model based on aggregation [2].

5.1 Entity

A Bel.Entity is the base definition for all Zonnon objects that are to be created within the BEL framework. Any implementation of Bel.Entity is required to export two methods. Initialize is called to prepare an object for use. Nullify is called after an object is no longer needed. Its purpose is for setting all dynamically allocated internal variables to nil, when they exist, so that they can be collected later by the garbage collector of the .NET and Mono frameworks.

5.2 Object

Bel.Object is a persistent Bel.Entity; in other words, implementations of interface Bel.Object can be saved to a file for later retrieval and use. Bel.Object refines the Bel.Entity definition. In addition to the two methods required by the Bel.Entity interface, the Bel.Object definition requires three more. Method Clone returns an empty instance of its type, i.e., it is void of any dynamically allocated data. Method Load retrieves data for an object, read from a binary file, and assigns it to itself. Typically, data are loaded from a file into an existing clone. Method Store writes the data of an object to a binary file. Load is the antithesis of Store. Said differently, Load and

Store are to be a one-to-one mapping between the temporary existence of an object in a computer's memory, and a persistent replica of that object on the computer's hard disk.

5.3 Typeset

Definition Bel.Typeset can aggregate with any of the above definitions, when it makes sense to do so. It provides two methods. Print converts an object into a Zonnon ***string***, while method Parse converts a ***string*** into its associated object. Like Store and Load, Print is the antithesis of Parse, and vice versa.

6 Data

Being able to manipulate data arrays and structures is a vital part to creating software applications.

6.1 Sorting

Module Bel.Sort exports two procedures. IntegerVector takes in an integer vector, sorts it from most negative to most positive, and returns the sorted vector. RealVector takes in a real vector, sorts it from most negative to most positive, and returns the sorted vector. The returned vectors overwrite the supplied vectors.

6.2 Structures

The data structures implemented here came from NIKLAUS WIRTH's classic text on the subject [4].[3] In fact, the software examples in his book have been ported over to Zonnon, and can be downloaded from the Zonnon website. His algorithms have been extended here so that they hold data that are instances of type Bel.Object, making these structures very flexible and also persistent.

The programmer needs to know *a priori* what data type is held within a data structure that has been stored to file before it can be read back in. In order to read data in from a binary file, a programmer must program the following two steps:

i) Specify the data type held within a data structure that is to be read in from a file by first calling the method: <a new data structure>.Configure(<a clone of the stored data type>).

ii) Read in its data using a reader attached to this file by calling the method: <a new data structure>.Load(<reader to the binary file>).

Consequently, if a BEL data structure is to be persistent, then it is restricted to hold just *one* type or kind of data.

By supplying a clone for the type of data being held by a data structure, you enable that data structure to load itself from file. This is not complete meta-programming, where a data structure need not know what type of data it will read-in in advance. Such a capability, although it would be nice, is seldom necessary, and is currently not supported.

It is not necessary to create or pass a node for any of the BEL data structures. All nodes are handled internally by the data structures themselves. All you have to do, as a programmer, is insert the data and, if necessary, supply its associated key. Bel.Keys.Key are used by lists and trees to sort data within their structures. Each datum entered into a list or tree must have a unique key.

6.2.1 Queues and Stacks

A *queue* is a first-in first-out (FIFO) data buffer. A *stack* is a first-in last-out (FILO) data buffer. Bel.Queue and Bel.Stack are implementations of definition Bel.Object, and are therefore persistent. Method Configure assigns the clone by which method Load reads itself in from some file. In addition, classes Bel.Queue and Bel.Stack possess a method Push to place a datum onto the queue or stack, and a corresponding method

3. When the copyright expired on WIRTH's textbook "*Algorithms + Data Structures = Programs*", he rewrote the manuscript for Oberon and made it publically available, placing it on the website http://www.oberon.ethz.ch.

Pop to release the next available datum from their structure. Method Length returns the number of data currently held by the buffer.

6.2.2 Keys

Module Bel.Keys exports type KeyType, which establishes the internal (or hidden) type of a key. Why reveal what the actual hidden type is? Well, by creating this secondary type, the interface for type Key can be made to be independent of any base system type. This allows for future growth without altering the exported interface for type Key. Say, for example, it became necessary to use System.Decimal as the raw data type for a key. This could be done easily by changing just type KeyType. The exported interface to type Key need not change. Now, certainly, the internal workings of this type would have to change, but the contract exported to the rest of the world need not. This gives its interface robustness.

A Bel.Keys.Key is an implementation of two definitions: Bel.Object and Bel.Typeset where Parse and Print pertain to the integer value held by a key. In addition to the methods required by these two interfaces, type Key also has methods Get and Set, and GetWord and SetWord, which are called by the overloaded assignment operator ":=". Methods Equals, LessThan, and GreaterThan are called by their appropriate overloaded operators "=", "#", "<", ">=", ">", and "<=".

The exported constants alphabet and characters provide the characteristics of an admissible **word** (assigned via a string). It can have up through 10 characters that are comprised from an alphabet of 78 symbols; specifically:

word = char { char }
char = letter | digit | punctuation | arithmetic | alien
letter = "A" .. "Z" | "a" .. "z"
digit = "0" .. "9"
punctuation = " " | "." | ":" | ";" | "," | "_" | "∼" | " ' " | ' " '
arithmetic = "+" | "−" | "∗" | "/" | "∧" | "="
alien = "?"

Even though white space is a permitted character within a word, any leading and trailing white space is stripped away by method SetWord, e.g., "et al." is an admissible word of six characters.

6.2.3 Lists

There are many kinds of lists that can be used as data structures in programming [4]. The list implemented in BEL is double-linked, and whose keys are sorted in ascending order. This means that its rider can traverse the list in either direction without having to restart each search from its home position.

A Bel.List inherits the Bel.Object interface, i.e., it is persistent. Method Length returns the number of data currently held by the list. A datum, when accompanied by a new key, can be loaded into a list through its method Insert or, when accompanied by a key that already exists in the list, that datum can be loaded into the list via method Update, taking the place of the datum previously held there. To remove a datum from a list, one sends its associated key to the list via method Delete.

A Bel.List has a rider to aid in its management, although it is not explicitly exported as such. Method Home sends this rider to its start position, i.e., that datum with the smallest key. Method Next moves the rider ahead by one node, if it isn't at the last node, while method Previous moves the rider back one node, if it isn't at the first node, or home. Alternatively, method Find locates that node which has the attached key. Once the rider is placed where the programmer wants it, the node's datum can be retrieved with method GetData, and its key can be extracted with method GetKey.

6.2.4 Trees

Lists are useful whenever the number of data are small, or whenever they are to be accessed sequentially, or whenever their structures are volatile in that new nodes (i.e., datum-key pairs) are constantly being created and/or destroyed. But whenever a data structure gets to be large, or whenever its data are to be accessed in a random manner, or whenever the overall structure is largely static, as is often the case in data bases, then a tree is preferred over a list. Like lists, there are numerous types of trees that can be used as data structures. The tree implemented in BEL is a balanced, binary, ADELSON-VELSKII-LANDIS (AVL) tree. WIRTH'S [4] algorithms for managing an AVL tree are employed here, only changing what was necessary so that the tree becomes a repository for any implementation of a `Bel.Object` definition, or any of its refinements.

`Bel.Tree` inherits the `Bel.Object` interface. Method `Entries` tells how many data are currently being held by a tree, while `Height` specifies how many generations or levels of branching there are in a tree. Methods `Insert`, `Update`, and `Delete` behave like they do for lists.

A BEL tree, like its kindred list, has a rider built into it to assist in its management. Method `Home` moves the rider to its home position which, in the case of a tree, is keyed about midway between its minimum and maximum nodes. Methods `Left` and `Right` move the rider up the tree by one node along the associated branch, if it exists, while method `Previous` moves the rider down the tree by one node towards home, if it isn't already at home. Alternatively, method `Find` locates that node which has the attached key. Once the rider is placed where the programmer wants it, the node's datum can be retrieved with method `GetData`, and its key can be extracted with method `GetKey`.

7 History of Changes

7.1 Changes to Version 4.2

Module `Bel.Sort` was added for sorting integer- and real-valued arrays.

7.2 Changes to Version 4.1

Licensing information is now written out to the terminal window at the completion of a job. Added histograms and segmented lines to BEL's plotting capability. Two data types now exist: `DataY` and `DataXY` for assigning to curves, and a capability to add text to a plot now exists. Data and text file encodings were changed from UTF-8 to UTF-16. Moved math functions `Erf`, `Erfc`, `Gamma`, and `Beta` from `Bel.Math` to `BelMath.SpecialFunctions`.

7.3 Changes to Version 4.0

With impending changes to be made to Zonnon and its compiler, it will no longer be necessary to introduce types Number, Array, and Matrix; thereby, greatly simplifying the overall set of BEL libraries. (These changes were largely motivated by the shear numbers of overloaded operations needed to implement Number, Array, and Matrix.) The change is actually quite simple: all constants assume their default size for their associated type. As a result, a new module had to be written, viz., Bel.Types, and the `BelMath.Functions` module was moved into the core as `Bel.Math` (which required moving `BelMath.Series`, too). The namespace for the core was changed from BelCore to just Bel. Also, words were added as admissible key values in `Bel.Keys`. The core BEL definitions were changed, too, to better separate concerns with respect to how they are typically used.

7.4 Changes to Version 3.3

The library was divided into a number of separate DLLs to make it easier for my students to digest. These include the CORE, TYPE, MATH, GAlg and PHYS in the standard distribution of BEL. CORE contains the most basic set of modules. TYPE contains the BEL math types. MATH contains the various math modules written to date. GAlg contains a genetic algorithm, which is a nice Zonnon application for teaching purposes. And PHYS extends the types of TYPE by combining them with physical units, viz., SI units. The various numeric types in MATH were restructured to take better advantage of efficiency enhancements that will be part of the next compiler release. For the most part, these changes took place under the hood. Only minimal changes took place at the interface. Nevertheless, this release was a major restructuring of the code.

7.5 Changes to Version 3.0

Type `Bel.Numbers.NumberType` was changed from an aliasing to a record type to clean up the interface. Numerous improvements in the low-level modules were made in the coding. Several bugs were fixed.

7.6 Changes to Version 2.0

There are two major changes with this release, and numerous minor ones. First, the fundamental definitions from which all BEL objects are based have been altered, and now include: `Bel.Object, Bel.Datum, Bel.Field` and `Bel.Typeset`. Second, the applications have been moved to their own libraries, with their own documentation, in an effort to simplify.

7.7 Version 1.0

Original port of the author's CAPO (Computational Analysis Platform in Oberon) package from Oberon to Zonnon.

Acknowledgments

The speed by which both Eugene Zueff and Roman Mitin have fixed compiler bugs uncovered during the writing of BEL has made this a fun project for me to be involved with. I was able create something useful for myself, and to also help the team of compiler developers at ETH Zurich debug their product during its development.

References

[1] Gutknecht, J., 2009, *Zonnon Language Report*, ETH Zürich, Switzerland, 4th ed., available online at www.zonnon.ethz.ch.

[2] Gutknecht, J., Mitin, R., and Gonova, N., 2010, *Zonnon Tutorial*, ETH Zurich, Switzerland, available online at www.zonnon.ethz.ch.

[3] Hart, J. F., Cheney, E. W., Lawson, C. L., Maehly, H. J., Mesztenyi, C. K., Rice, J. R., Thacher, Jr., H. G., and Witzgall, C., 1968, *Computer Approximations*, The SIAM Series in Applied Mathematics, Wiley, New York.

[4] Wirth, N., 1986, *Algorithms + Data Structures = Programs*, 2nd ed., Prentic-Hall.

A Log Error/Warning Messages

A listing of the pre-built log codes known to `Bel.Log`. Not all code numbers are assigned; in fact, most available numbers are not assigned.

Codes: 0–49 future plans or implementation limitations

0	An invalid error number - valid numbers are in [1..999]
1	This is a code limitation to be removed someday
5	The assigned glyph was the unknown character '?'
6	The assigned character cannot be handled
10	Complex numbers are not yet implemented
15	The exponent lies outside its range of [-9..9]
20	Program execution was terminated prematurely

Codes: 50–99 handle system errors

50	Arithmatic overflow
51	Arithmatic underflow
55	Iteration terminated due to round off error
60	The requested binary data file does not exist
61	The requested UTF-16 text file does not exist
70	The element is not a member of the enumerated type
71	The number sent does not belong to the required subset
80	Iteration terminated when an upper iterate limit was hit
81	There was an extrapolation instead of an interpolation
90	Computation was allowed to continue, nothing was changed
91	Computation was allowed to continue, changes were made

Codes: 100–199 handle BelPhys.Scalars and similar errors

100	Units were not compatible for assignment
101	Units were not compatible for arithmatic operation
102	Units were not compatible for boolean arithmetic

Codes: 200–299 handle BelPhys.Vectors and similar errors

200	Units were not compatible for assignment
201	Units were not compatible for arithmatic operation
202	Units were not compatible for boolean arithmetic
204	An array of zeros was assigned
205	No array was assigned, the request was ignored
206	A boolean array of 'false' was assigned
207	A boolean 'false' was assigned
210	An index was out of range
211	Index ranges were not compatible for contraction
212	Index ranges between arrays are not compatible
220	The zero vector has no direction
225	The array had the wrong dimension
230	Dynamic array must be created before accessed

Codes: 300–399 handle BelPhys.Tensors and similar errors

300 Units were not compatible for assignment
301 Units were not compatible for arithmatic operation
302 Units were not compatible for boolean arithmetic
304 A matrix of zeros was assigned
305 No matrix was assigned, the request was ignored
306 A boolean matrix of 'false' was assigned
307 A boolean 'false' was assigned
310 A matrix index was out of range
311 Index ranges were not compatible for contraction
312 Index ranges between matrices are not compatible
315 Two matrices have incompatible dimensions
320 The matrix must be square; it was not
321 The matrix does not have full rank
322 The matrix was not positive definite
323 The matrix was singular; its inverse dosn't exist
325 The matrix had the wrong dimensions
330 Dynamic matrix must be created before accessed
340 Matrix norm too large for Taylor series expansion

Codes: 400–499 handle function/procedure/method errors

400 An argument sent to the function was: positive infinity
401 An argument sent to the function was: negative infinity
402 An argument sent to the function was: not a number
403 An argument sent to the function was: a nil object
404 An argument sent to the function had the wrong type
405 An argument of the function lies outside its domain
406 An argument of the function must be dimensionless
407 An argument of the function had the wrong units
408 Arguments supplied to the function were inconsistent
410 A nil function was supplied
411 Function computation produced: a 0/0
412 Function computation produced: an Infinity/Infinity
413 Function computation produced: a complex number
415 The function was not executed
416 The function returned a default value
417 The function is not defined for this argument
418 The function is not defined for these arguments
420 The function returned: positive infinity
421 The function returned: negative infinity
422 The function returned: not a number
423 The function returned: a nil object
424 The function returned: false
425 The function returned: zero
426 The function returned: an array of zeros
427 The function returned: a matrix of zeros
428 The function returned: a dimensionless scalar
429 The function returned: a dimensionless vector
430 The function returned: a dimensionless tensor
431 The function returned: a dimensionless quad-tensor
440 The supplied instance of a procedure type was nil

Codes: 500–599 handle object errors

500	The object supplied was nil
501	The object supplied was of the wrong type
502	The object supplied was of an unknown type
510	Once set, a key cannot be reset
511	The key supplied is already in use in the data structure
515	An inconsistent data set was supplied
520	Failed to insert a datum into a data structure
521	Failed to update a datum in a data structure
522	Failed to delete a datum from a data structure
523	Failed to locate a datum in a data structure
550	Curves in log-plots must contain only positive valued data

Codes: 600–999 are reserved for future growth.

B Definitions

```
definition Bel.Entity;
   procedure Initialize;
   procedure Nullify;
end Entity.

definition Bel.Object refines Entity;
   import
      System.IO.BinaryReader as BinaryReader,
      System.IO.BinaryWriter as BinaryWriter,
      Bel.Entity as Entity;
   procedure Clone () : object{Object};
   procedure Load (br : BinaryReader);
   procedure Store (bw : BinaryWriter);
end Object.

definition Bel.Typeset;
   procedure Parse (s : string);
   procedure Print () : string;
end Typeset.
```

Implementations Bel.EmptyEntity and Bel.EmptyObject provide templates from which one can create their own types.

C IO Modules

```
module Bel.Version;
   var
      version : string;
end Version.
```

```
module Bel.Log;
   procedure Open (directory : string);
   procedure Close;
   procedure Message (message : string);
   procedure WarningMessage (inputErrorCode, outputErrorCode : integer;
                      originOfError : string);
   procedure ErrorMessage (inputErrorCode, outputErrorCode : integer;
                      originOfError : string);
end Log.
```

```
module Bel.DataFiles;
   import
      System.IO.BinaryReader as BinaryReader,
      System.IO.BinaryWriter as BinaryWriter;
   procedure FileExists (fileName : string) : boolean;
   procedure OpenReader (fileName : string) : BinaryReader;
   procedure OpenWriter (fileName : string) : BinaryWriter;
   procedure CloseReader (br : BinaryReader);
   procedure CloseWriter (bw : BinaryWriter);
end DataFiles.
```

```
module Bel.TextFiles;
   import
      System.IO.StreamReader as StreamReader,
      System.IO.StreamWriter as StreamWriter;
   procedure FileExists (fileName : string) : boolean;
   procedure OpenReader (fileName : string) : StreamReader;
   procedure OpenWriter (fileName : string) : StreamWriter;
   procedure CloseReader (sr : StreamReader);
   procedure CloseWriter (sw : StreamWriter);
end TextFiles.
```

D Numeric Types

```
module Bel.Types;
   import
      System.IO.BinaryReader as BinaryReader,
      System.IO.BinaryWriter as BinaryWriter;
   var {immutable}
      MaximumInteger : integer;
      Epsilon, MaximumReal, MinimumReal, NaN,
                      NegativeInfinity, PositiveInfinity : real;
   type
      BooleanVector = array {math} * of boolean;
      IntegerVector = array {math} * of integer;
      RealVector = array {math} * of real;
      BooleanMatrix = array {math} * , * of boolean;
      IntegerMatrix = array {math} * , * of integer;
      RealMatrix = array {math} * , * of real;
   procedure IsFinite (x : real) : boolean;
   procedure IsInfinite (x : real) : boolean;
   procedure IsNaN (x : real) : boolean;
   procedure BooleanToString (b : boolean) : string;
   procedure IntegerToString (i : integer) : string;
   procedure NumberToString (x : real; significantDigits : integer) : string;
   procedure RealToString (x : real; significantDigits : integer) : string;
   procedure StringToBoolean (s : string) : boolean;
   procedure StringToInteger (s : string) : integer;
   procedure StringToReal (s : string) : real;
   procedure LoadString (r : BinaryReader) : string;
   procedure LoadBoolean (r : BinaryReader) : boolean;
   procedure LoadInteger (r : BinaryReader) : integer;
   procedure LoadReal (r : BinaryReader) : real;
   procedure LoadBooleanVector (r : BinaryReader) : BooleanVector;
   procedure LoadIntegerVector (r : BinaryReader) : IntegerVector;
   procedure LoadRealVector (r : BinaryReader) : RealVector;
   procedure LoadBooleanMatrix (r : BinaryReader) : BooleanMatrix;
   procedure LoadIntegerMatrix (r : BinaryReader) : IntegerMatrix;
   procedure LoadRealMatrix (r : BinaryReader) : RealMatrix;
   procedure StoreString (w : BinaryWriter; s : string);
```

```
   procedure StoreBoolean (w : BinaryWriter; b : boolean);
   procedure StoreInteger (w : BinaryWriter; i : integer);
   procedure StoreReal (w : BinaryWriter; x : real);
   procedure StoreBooleanVector (w : BinaryWriter; a : BooleanVector);
   procedure StoreIntegerVector (w : BinaryWriter; a : IntegerVector);
   procedure StoreRealVector (w : BinaryWriter; a : RealVector);
   procedure StoreBooleanMatrix (w : BinaryWriter; m : BooleanMatrix);
   procedure StoreIntegerMatrix (w : BinaryWriter; m : IntegerMatrix);
   procedure StoreRealMatrix (w : BinaryWriter; m : RealMatrix);
end Types.

module Bel.Series;
   import
      Bel.Types as T;
   type
      GetCoef = procedure (integer; real; var boolean) : real;
   procedure ContinuedFraction (getA, getB : GetCoef; x : real) : real;
   procedure TruncatedContinuedFraction
                       (a, b : T.RealVector; x : real) : real;
   procedure PowerSeries (getA : GetCoef; x : real) : real;
   procedure TruncatedPowerSeries (a : T.RealVector; x : real) : real;
   procedure RationalSeries (getA, getB : GetCoef; x : real) : real;
   procedure TruncatedRationalSeries
                       (a, b : T.RealVector; x : real) : real;
end Series.

module Bel.Math;
   var {immutable}
      E, PI : real;
   procedure Random () : real;
   procedure Max (x, y : real) : real;
   procedure Min (x, y : real) : real;
   procedure Ceiling (x : real) : real;
   procedure Floor (x : real) : real;
   procedure Round (x : real) : real;
   procedure Abs (x : real) : real;
   procedure Sign (x : real) : real;
   procedure Sqrt (x : real) : real;
   procedure Power (x, y : real) : real;
```

```
   procedure Pythag (x, y : real) : real;
   procedure Log (x : real) : real;
   procedure Ln (x : real) : real;
   procedure Exp (x : real) : real;
   procedure Sin (x : real) : real;
   procedure Cos (x : real) : real;
   procedure Tan (x : real) : real;
   procedure ArcSin (x : real) : real;
   procedure ArcCos (x : real) : real;
   procedure ArcTan (x : real) : real;
   procedure ArcTan2 (y, x : real) : real;
   procedure Sinh (x : real) : real;
   procedure Cosh (x : real) : real;
   procedure Tanh (x : real) : real;
   procedure ArcSinh (x : real) : real;
   procedure ArcCosh (x : real) : real;
   procedure ArcTanh (x : real) : real;
end Math.
```

E Plots

Module `System.Drawing` is shown in the imports of modules `Bel.Curves` and `Bel.Plots`, not because they appear in their interfaces, but because these modules require library `System.Drawing.dll` to be added to the list of referenced libraries at compile time, along with library `NPlot.dll`.

```
module Bel.Curves;
   import
      System.Drawing,
      System.Single as Single,
      NPlot as NP,
      Bel.Types as T;
   type
      Color = (black, blue, gray, green, orange, pink, purple, red, yellow);
      Dimension = (tiny, small, medium, large, huge);
      Marker = (circle, cross, diamond, flagDown, flagUp, none, plus, square,
                        triangle, triangleDown, triangleUp);
      SingleVector = array * of Single;
```

```
    type {value} Attributes = object
      var
        centered, filled : boolean;
        color : Color;
        dimension : Dimension;
        label : string;
        mark : Marker;
      procedure Initialize;
    end Attributes;
    type {ref} DataY = object
      var
        attributes : Attributes;
        data : SingleVector;
      procedure Initialize;
      procedure Assign (vec : T.RealVector);
    end DataY;
    type {ref} DataXY = object
      var
        attributes : Attributes;
        xData, yData : SingleVector;
      procedure Initialize;
      procedure AssignX (vec : T.RealVector);
      procedure AssignY (vec : T.RealVector);
    end DataXY;
    procedure Histogram (histogramData : DataY; normalCurve : Attributes;
                         var mean, median, stdDev : real;
                         var histogram : NP.HistogramPlot;
                         var normalPlot : NP.LinePlot);
    procedure LineCurveY (data : DataY) : NP.LinePlot;
    procedure LineCurveXY (data : DataXY) : NP.LinePlot;
    procedure PointCurveY (data : DataY) : NP.PointPlot;
    procedure PointCurveXY (data : DataXY) : NP.PointPlot;
    procedure StepCurve (data : DataY) : NP.StepPlot;
end Curves.
```

```
module Bel.Plots;
   import
      System.Drawing,
      NPlot as NP;
   type
      Location = (lowerLeft, lowerRight, upperLeft, upperRight, outside);
   type {ref} Plot = object
      procedure Create (xPixels, yPixels : integer);
      procedure AddAxes (xIsLog, yIsLog : boolean; xLabel, yLabel : string);
      procedure AddLegend (locate : Location);
      procedure AddTitle (title : string);
      procedure AddHistogram (curve : NP.HistogramPlot);
      procedure AddLineCurve (curve : NP.LinePlot);
      procedure AddPointCurve (curve : NP.PointPlot);
      procedure AddStepCurve (curve : NP.StepPlot);
      procedure AddText (text : string; atX, atY : real);
      procedure Save (fileName : string);
   end Plot;
end Plots.
```

F Data Modules

Procedures inherited from definition interfaces are not repeated here.

```
module Bel.Sort;
   import
      Bel.Types as T;
   procedure IntegerVector (var x : T.IntegerVector);
   procedure RealVector (var x : T.RealVector);
end Sort.
```

```
object {ref} Bel.Queue implements Object;
   import
      Bel.Object as Object;
   procedure Configure (dataClone : object{Object});
   procedure Length () : integer;
   procedure Pop () : object{Object};
   procedure Push (o : object{Object});
end Queue.
```

```
object {ref} Bel.Stack implements Object;
   import
      Bel.Object as Object;
   procedure Configure (dataClone : object{Object});
   procedure Length () : integer;
   procedure Pop () : object{Object};
   procedure Push (o : object{Object});
end Stack.

module Bel.Keys;
   import
      System.Int64 as Integer,
      Bel.Object as Object,
      Bel.Typeset as Typeset;
   const
      alphabet = 78;
      characters = 10;
   type KeyType = Integer;
   type {value} Key = object implements Object, Typeset
      procedure Get () : KeyType;
      procedure GetWord () : string;
      procedure Set (k : KeyType);
      procedure SetWord (w : string);
      procedure Equals (k : Key) : boolean;
      procedure LessThan (k : Key) : boolean;
      procedure GreaterThan (k : Key) : boolean;
   end Key;
   operator ":=" (var l : Key; r : Key);
   operator ":=" (var l : Key; r : integer);
   operator ":=" (var l : Key; r : string);
   operator "=" (l, r : Key) : boolean;
   operator "#" (l, r : Key) : boolean;
   operator "<" (l, r : Key) : boolean;
   operator "<=" (l, r : Key) : boolean;
   operator ">" (l, r : Key) : boolean;
   operator ">=" (l, r : Key) : boolean;
end Keys.
```

```
object {ref} Bel.List implements Object;
   import
      Bel.Object as Object,
      Bel.Keys as K;
   procedure Configure (dataClone : object{Object});
   procedure Length () : integer;
   procedure Delete (key : K.Key; var success : boolean);
   procedure Insert (obj : object{Object}; key : K.Key
                     var success : boolean);
   procedure Find (key : K.Key; var found : boolean);
   procedure Home;
   procedure Next (var moved : boolean);
   procedure Previous (var moved : boolean);
   procedure GetData () : object{Object};
   procedure GetKey () : K.Key;
   procedure Update (obj : object{Object}; key : K.Key;
                     var success : boolean);
end List.

object {ref} Bel.Tree implements Object;
   import
      Bel.Object as Object,
      Bel.Keys as K;
   procedure Configure (dataClone : object{Object});
   procedure Entries () : integer;
   procedure Height () : integer;
   procedure Delete (key : K.Key; var success : boolean);
   procedure Insert (obj : object{Object}; key : K.Key;
                     var success : boolean);
   procedure Find (key : K.Key; var found : boolean);
   procedure Home;
   procedure Left (var moved : boolean);
   procedure Right (var moved : boolean);
   procedure Previous;
   procedure GetData () : object{Object};
   procedure GetKey () : K.Key;
   procedure Update (obj : object{Object}; key : K.Key;
                     var success : boolean);
end Tree.
```