ENTER FREYA

by J. R. Latasa and I. Marteens
(www.marteens.com, www.moebiusootics.com)

***Summary**: Freya is a pure, non-hybrid, object oriented programming language, inspired in the syntactic style of the Pascal lineage of languages. Freya is designed to create applications that run as first-class citizens in the .NET Framework. This article explains the basic syntactic features of Freya, and some of the reasoning behind Freya's specification.*

The Delphi Pascal heritage

Object oriented constructions were introduced for Turbo Pascal 5.5, but the big leap came when Borland announced Delphi back in 1995. In 1997, Delphi 3 added *interface types* to the language, and Delphi 4 rounded some edges by adding dynamic arrays, method overloading and default parameters. Since then, few changes have been done to Delphi Pascal as a language.

With the arrival of the .NET framework, Borland faced a big challenge: adapting its more popular language product for the new framework runtime model, while keeping the compatibility with most of the old Delphi's libraries. We think Borland has done its best in order to keep a minimal compatibility with old Delphi's code base. We also think this is a very restrictive constraint to deal with. So, the question we asked ourselves was: how would be a modern Pascal-derived language if it could be designed from scratch? The answer is a language named after the Norse goddess of love: Freya.

Design goals

We have stated three explicit design goals for the new programming language:

1. Freya must be a pure object oriented language, running in the .NET Framework.

   But the main consequence is that we can cut off a lot of inherited features that have little or no use in an object-oriented world. Some of these features are: first-level procedures, nested procedures, global variables and variant records.

   There are more subtle features in Delphi that defy object purity. One of those features, for instance, is the way numeric variables are incremented. Delphi provides an *Inc* procedure. Freya couldn't use *Inc*, except by turning it into a static procedure from some global class. We have preferred, instead, to support the widely known ++ operator from C++/Java/C#.

2. Freya should respect, as long as possible, the distinct syntactic flavour of Delphi Pascal.

   Sure, we have to state first what this flavour is. Some people use Pascal because they think it's not as cryptic as the C language family, with all those curly braces, cumulative pluses and other strange thingies. This could be regarded as an advantage... especially when your code editor lets you type words like ***initialization*** in a single breath, despite your QWERTY keyboard.

   But the most decisive characteristic of Pascal is its "split" nature. While most modern languages, like C#, Java or Eiffel, declare and implement their features in the same place, Pascal shares with C++ a clear division between declaration and implementation.

   There's another more shameful reason for explaining this chasm... and it has to do in part with those prehistoric devices once known as magnetic tapes. Pascal, and most languages from that generation, was carefully designed to allow one-pass compiling. Have you ever used a ***forward*** directive? Magnetic tapes and punched cards were only partially responsible for this, and part of the blame should be put on the low memory availability common at those times. Modern languages are compiled by constructing first an *abstract syntax tree* in memory, so there's no need for the dreaded *"declare first, use later"* rule.

   If we had discarded the one-pass style, we would have ended with a simplified Eiffel clone on our hands. Freya is the result of respecting as long as possible the one-pass

heritage of Pascal: no need to *"declare first"*. Nevertheless, class members are implemented in a separate **implementation** section, which could contain other implementation details for a class.

3. Freya must fit neatly into the .NET framework as a first-class citizen.

This is a two-way requirement. It means, on one hand, that Freya classes and applications should be able to use every feature from the Common Language Runtime (CLR) other languages enjoy. Moreover, these features should be used in a natural way. A clear example of what we'll be avoiding is the support by Delphi of operator overloading: while Delphi v8 allows it, it forces the programmer to define the operator using the literal name associated to the operator, instead of the own operator.

On the other side, new features may be added as long as their impact on other .NET languages is acceptable. Freya classes should be accessible from other .NET programming languages. We do not pretend to attract programmers to Freya and then keep them prisoners as a consequence of using advanced Freya features not supported by the mainstream languages. As Zen has it: you can hold a coin in your hand by closing your fist, but also by turning the hand palm up.

The truth is that our third requirement is very demanding. It means that the design of Freya should not deviate too much from the core .NET object model. But that's not our goal, neither... at least for this first version.

The entry point: no more programs

We will proceed top-down, starting with the high level syntactic structures. As a matter of fact, these high level features will endure most of the simplifications needed to make Freya a streamlined language.

The first victim of the scissors will be the classic ***program*** construction. Let's face it: ***program*** is a relic from the old good structured programming times. The very existence of ***program*** forces Delphi to define three additional keywords and their associated structures: ***library*** and ***package***, in order to match ***program*****'s** role for Delphi DLLs and component libraries, and ***unit***, for the modular compilation ... err... unit.

This way, ***program*** disappears and it is substituted by a *Main* static function, as in C#. The following listing shows a basic *Hello World* console application in Freya:

```
using System;
namespace Freya.Hello.World;

implementation

    method Main(Args: Array[String]);
    begin
        Console.WriteLine('Hello, Freya!');
    end;

end.
```

An anonymous implementation section automatically generates an internal static class, so the above code is equivalent to this other one:

```
using System;
namespace Freya.Hello.World;

public
    _InternalClass = static class
    private
        method Main(Args: Array[String]);
    end;

implementation for _InternalClass is

    method Main(Args: Array[String]);
    begin
        Console.WriteLine('Hello, Freya!');
    end;

end.
```

Ok, this is a little longer than its C# or Pascal equivalents, but this is not the code you'll be writing more frequently. Anyway, this is an almost literal translation of the empty console application template generated by Visual C# 2005.

Though we considered several alternatives to the *Main* function's protocol, we found most of them functionally equivalent to ours, or plainly bizarre. Eiffel applications, for instance, starts their execution by constructing an instance of a class marked as the *system root* from outside of the code. The practical consequences of this design are a bunch of ad hoc classes with no further goal than serving as the starting point of an application.

Namespaces

Delphi units do not fit well in the .NET framework. Units were designed as the smallest compilable language module, but there's nothing alike in .NET. When a .NET project is compiled, all its source files are merged together before proceeding. There's a compiled entity below assemblies: the .NET module. But there is a performance penalty when an assembly is divided in physically independent modules and, in any case, modules are neither a good equivalent for Pascal's compiled units.

In other words:

1. Delphi identifies units and files.
2. Delphi.NET v8 identifies namespaces with units. That's plainly wrong. A modest size component library written in Delphi 8 would contain a namespace for each single unit in the project. It's amazing that such a flaw in design were featured by a commercial product.
3. Delphi.NET v2005 tried to ameliorate this situation, synthesizing the namespace from dotted unit names by dropping the last identifier. If you compile a unit named *X.Y.Z* in Delphi 2005, all its content will belong to a *X.Y* namespace. This is better, but it is still just a clever trick to disguise a wrong design decision.

It's funny that the Delphi 2005 approach is gentler with a C# programmer who uses a Delphi component, since he only needs to know about the namespace, than with a Delphi programmer, who's forced to remember both the namespace *and* the unit name.

Freya throws away Delphi's unit structure and conforms to the more natural model so successfully featured by C#: namespaces are logical entities not bound to any physical model. One file may contain more than a namespace, and namespace features can be defined across more than one file, and even across several assemblies.

A basic Freya source file could be like this:

```
namespace Freya.Containers;
    ⋮
    // Features for the Freya.Containers namespace…
    ⋮
end.
```

When more than one namespace is defined in the same file, their closing ***end***'s are consolidated:

```
namespace Freya.Containers;
    ⋮
namespace Freya.Containers.HashTables;
    ⋮
end.
```

You could even reopen an already closed namespace in the same file.

Let's focus now in namespace structure. Although Wirth's Pascal didn't support units, the syntax for this feature is almost the same in all Pascal inspired languages. They all divide modules in two rigidly shaped parts: an interface and an implementation section. The interface section hosts all public declarations. Among those declarations, we can find class declarations. Inside a class, you will find private, protected and public sections... Now we have detected a minor inconsistency: why do we use ***interface***/***implementation*** for units, and ***public***/***protected***/***private*** for classes? The explanation has to do with the way these lan-

guages evolved. Unit structure was defined back in the era of Structured Programming. Visibility sections in classes, as a matter of fact, imitated the corresponding syntax constructions from C++.

That's why, in the sake of uniformity, we propose to abandon ***interface***/***implementation*** and to substitute them with ***public***/***private***: public namespace sections in Freya hosts public declarations, and ***private*** is the Freya equivalent of C#'s ***internal*** declarations. This is the general layout of a namespace in Freya:

```
uses System, System.Data, Freya.DataStructures;

namespace UnitName;

public

    MyClass = class
        ⋮
    end;

implementation for MyClass is

    ⋮

end.
```

This example shows also a more radical departure from the Pascal syntax: visibility sections inside a Freya namespace can be repeated. This is consistent with the syntax for visibility sections in a class declaration. One use for this feature could be grouping related public or private functionality in different sections:

```
namespace RocketScience;
public
    // Here go all the classes for
    // monitoring the ignition subsystem

public
    // The communication subsystem

private
    // Any needed private classes or types

implementation for RocketEngine is
    // An implementation section for each defined class

    ⋮
end.
```

There are neither ***initialization*** nor ***finalization*** sections in a Freya namespace. On one hand, there are no global variables to be initialised. On the other, static fields from a class can be initialised by a static constructor.

Type declarations

All direct namespace members in .NET languages are type declarations, contrasting with classic Delphi Pascal, which allows constants, type declarations, global variables routines and even labels. There's no need in Freya to prefix type declarations with a ***type*** keyword, as in Delphi Pascal. There are six kinds of type declarations: enumerations, delegates, interfaces, records, classes and class references. The following excerpt shows how to declare a simple enumerative type:

```
// Freya
public
    DOW = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

The open parenthesis after the equal sign tells the compiler it is handling an enumerative type declaration. A more sophisticated declaration could be like this:

```
// Freya
public
    BasicColor = (Red = $0000FF, Green = $00FF00, Blue = $FF0000): Cardinal;
```

This time, there are explicit values assigned to enumerative constants, and there's also a base type hint at the end of the declaration. This feature resembles an already existing C# construction:

```
// C#
public enum BasicColor: ulong {
    Red = 0x0000ff, Green = 0x00ff00, Blue = 0xff0000 };
```

Declaring a delegate type is easier in Freya than in Delphi:

```
// Freya
public
    NotifyDelegate = method (Sender: object);
```

There's no ***of object*** suffix in the delegate declaration, because there are no first-level procedures. Of course, you could declare delegates based on functions rather than on procedures:

```
// Freya
public
    AnotherDelegate = method (Sender: object): Boolean;
```

Finally, we allow type synonyms, a feature not yet supported by C#:

```
// Freya
public
    StringDictionary = System.Collections.Generic.Dictionary[String, String];
```

The nearest equivalent in C# is a variant of the **using** clause:

```
// C#
using StrDict = System.Collections.Generic.Dictionary<string,string>;
```

But this definition is only valid inside the file it is included. Freya synonyms, in despite of the lack of direct support by the CLR, are defined once and can be used as regular types by any code that references the Freya assembly that includes the definition.

Classes and class members

The most important namespace direct members are class declarations. Class can be declared in any visibility section, public or private, but they must be always implemented in a private section. This is the syntax for a class declaration:

```
[abstract | sealed | static] [partial] class [ formal-generics ][ inheritance ]
    class-members
end;
```

where the optional *inheritance* clause uses the same syntax as in Delphi. Genericity is discussed below.

This example features an overly simplified integer stack class, and it shows how methods are always implemented in a separate **implementation** section:

```
namespace Stacks;

public

    IntStack = class
    private
        Items: Array[Integer];
        Count: Integer;
    public
        ⋮
        method Push(Value: Integer);
        method Pop;
    end;

implementation for IntStack is

    method Push(Value: Integer);
    begin
        // Ignore array capacity and stack overflow
        Items[Count] := Value;
        Count++;
    end;

    method Pop;
    begin
        // Ignore stack underflow
        Count--;
```

```
    end;
    ⋮
end.
```

Since *Items* and *Count* are just implementation details that affect neither the contract between the class and its users, nor between the class and its inheritors, they can be moved to the implementation section, as an alternative:

```
namespace Stacks;

public

    IntStack = class
    public
        ⋮
        method Push(Value: Integer);
        method Pop;
    end;

implementation for IntStack is

    var
        Items: Array[Integer];
        Count: Integer := 0;

    method Push(Value: Integer);
    begin
        // Ignore array capacity and stack overflow
        Items[Count] := Value;
        Count++;
    end;

    method Pop;
    begin
        // Ignore stack underflow
        Count--;
    end;

    ⋮
end.
```

Thanks to this syntax, private items are declared closer to the place they are used. You don't need private members to understand how a class interacts with its clients. And, as a welcomed side effect, the "contract form" of the class gets more compact, and more readable.

You can declare inner types inside a class. For nested classes, records and interfaces you could declare the whole nested type inside the outer class declaration, or you could provide a forward declaration and declare the inner type below. Take a look at this example:

```
namespace Freya.LinkedLists;

public

    LinkedList = class[X]
    protected
        LinkedNode = class
        public
            Value: X;
            Next: LinkedNode;
        end;
    end;
    ⋮
end.
```

Things can mess when the inner type declaration is long enough. As an alternative, you could declare the inner class as follows:

```
namespace Freya.LinkedLists;

public
    LinkedList = class[X]
    protected
        LinkedNode = class;
    end;
```

```
    LinkedList[X].LinkedNode = class
    public
        Value: X;
        Next: LinkedList[X].LinkedNode;
    end;

    ⋮

end.
```

Though the full declaration of *LinkedNode* has been written inside a public section of the namespace, you can access this class only inside methods from *LinkedList* and any possible derived classes.

Visibility

There are several reasons against keeping Delphi accessibility levels. The most important: Delphi assigns private and protected a totally different meaning than the one assigned by most important .NET languages. That's an accident waiting to happen. One of the authors, moreover, finds the "strict" jargon a joke:

"Ok, guys, we'll make DontTouchThisButton a private method… well, yes… I mean, we'll be strict on this. It'll be a strictly private resource… and this time I really mean it…"

Inside a type declaration, we allow six different visibility sections: **public**, **protected**, **private**, **internal**, **internal protected** and **implementation protected**. The first five of them are interpreted as in C#, and then we introduce an additional modifier:

```
implementation protected
```

The order of keywords can be inverted, and their meaning would still be the same:

```
protected implementation
```

The new visibility level corresponds to the CLR's *family and assembly*: your code is required to be part of the same assembly and located in a derived class if you want to access an implementation protected resource. On the other hand, **internal protected** means *family or assembly*, which is a more relaxed requirement.

Instance and static members

.NET classes can have "normal" *instance members*, which are allocated and access in a per instance manner, and *static members*, which can only be managed using the class name. Although Delphi already has a very idiosyncratic *class method*, which could be regarded as a special kind of instance members to some extent, it didn't allow for static fields. This is the solution adopted by Delphi.NET for declaring static members:

```
// Delphi.NET
type
    AClass = class
    private
        var
            // Normal instance fields
            Field1, Field2: Integer;
            Field3, Field4: string;
        class var
            // Static fields
            Field5, Field6: Double;
            Field7: Boolean;

    private
        // The old style is still accepted
        Field8: Integer;
    protected
        // Static method declarations conforms to totally different rules!
        class function DoThat(Value: Integer): string; static;
        ⋮
    end;
```

Static methods and static fields are declared through totally unrelated constructions. There's another detail: ***static*** implies ***class*** for method declarations, so we have a little redundancy here.

Static members could be seen as normal members of another class, implemented after the singleton pattern, closely associated to the original class. In this light, when we declare a class with both static and non-static members, we are really declaring members for two related classes. This interpretation, by the way, is consistent with the implementation of other class features as class methods, virtual constructors and class references. If we were to design a hierarchy around member attributes, the most natural grouping would be similar to this one:

```
Class declaration
    Instance members
        Visibility sections
    Static members
        Visibility sections
```

We would separate first instance members from static members, and only then, we would proceed with categorising by member visibility... i.e., a totally different schema compared with Delphi.NET schema: visibility first, and only then, the static or instance attribute.

C# and Java solves this problem by explicitly including all modifiers along with the declared class member. There are no special sections inside a class. Freya uses an intermediate style, as shown in this example:

```
MyClass = class(MyAncestor)
private
    FNextInstance: MyClass;
    FReserved: Double;

public
    method PublicInstanceStuff;

static internal protected
    FNumberOfInstances: Integer;
    FFirstInstance: MyClass;

static public
    method NumberOfInstances: Integer;
    property Wow: WhoCares;
end;
```

One of the advantages of this syntax is that you can still mimic the C#/Java's way. This class declaration is valid Freya code:

```
MyClass = class(MyAncestor)
    private FNextInstance: MyClass;
    private FReserved: Double;

    public method PublicInstanceStuff;

    static internal protected FNumberOfInstances: Integer;
    static internal protected FFirstInstance: MyClass;

    static public method NumberOfInstances: Integer;
    static public property Wow: WhoCares;
end;
```

This way, you'll have an easier time if you ever need to translate Java or C# code to Freya. You could start leaving one explicit visibility section for each class member, and leave grouping until you find the time.

Methods

Despite the removal of the ***static*** directive from method declarations, there remain some inconsistencies with the rest of method directives in Delphi. For instance, Delphi requires an abstract method to be declared both ***virtual*** and ***abstract*** (in this order!). Freya's binding modifiers for methods are mutually exclusive, and only one of them, at most, is allowed for a given method declaration. Recognised binding modifiers are:

1. ***virtual***: It's used the first time a virtual method is introduced in a class hierarchy.

2. ***abstract***: It's an alternative to ***virtual***, and it means that the method has no implementation yet. Of course, an abstract method is always a virtual method.
3. ***override***: It's used when redefining a virtual method in a derived class.
4. ***sealed***: A virtual method introduced in an ancestor is redefined for the last time, and it cannot be overridden in a class derived from the current one.

Another independent method modifier is ***new***, which can be used alone or combined with ***virtual*** or ***abstract***. It has the same meaning as in C#, and it's used when a new method declaration masks a similar declaration inherited from the ancestor. This is the full syntax for a method declaration in Freya:

```
method method-identifier
    [parameters][: return-type]; [modifiers ;]
```

Since binding modifiers are mutually exclusive, you can only have two modifiers at most for a single method. The order of the modifiers is irrelevant: ***new virtual*** is exactly the same as ***virtual new***.

There is no overload directive in Freya, and it's important to understand why. Delphi 4 was forced to introduce an explicit overload directive to keep backward compatibility with code and tools. If we use a traditional linker, overloaded methods require *name mangling*, a technique that adds the encoded method signature to the original method name. Freya has no compatibility issues to address, but even more important: the own CLR supports directly method overloading, without resorting to name mangling.

Properties and events

Freya is a component-oriented language, and it must provide first-class support for properties and events. Our goal in this area is to simplify the declaration and implementation of these features. This simple example shows how properties are declared in Freya:

```
Button = class(Control)
public
    ⋮
    property Caption: string;
end;
```

Delphi, unlike C#, requires the programmer to provide explicit accesors for each property. If you want to declare a *Caption* property in Delphi, you'll have to declare separate methods with names like *GetCaption* and *SetCaption*, at least in the more general case. This requirement implies extra typing and unneeded complexity. Freya, on the other hand, assumes that properties allows reading and writing by default, and avoids the explicit declaration of read and write accesors.

Nevertheless, you have to provide those special methods later, in the class implementation:

```
implementation for Button is

    property Caption: string;
    begin
        ⋮
    end;

    property Caption(Value: string);
    begin
        ⋮
    end;
```

Although both "methods" share their names, the former is easily identified as a function, and it provides the implementation of readings. The latter has no return type, so it necessarily implements the write access for the property.

We could also declare read only properties, by including the **readonly** attribute:

```
IntStack = class
public
    ⋮
    property Top: Integer; readonly;
```

```
    property IsEmpty: Boolean; readonly;
end;
```

This time, Freya only allows a read accesor implementation:

```
property IsEmpty: Boolean;
begin
    Result := Count = 0;
end;
```

Again, since the accesor returns a value, it should be evident we're dealing with the read accesor. Write only properties are not allowed in Freya, since this restriction would defeat the very concept of property.

Another useful feature is the automatic implementation of properties as fields. If you declare a property and you don't provide access methods for the property, Freya assumes you want an automatic implementation, using a private hidden field:

```
public

    MyClass = class
    public
        constructor MyClass;
        property A: Integer; readonly;
        property B: String;
    end;

implementation for MyClass is

    constructor MyClass;
    begin
        A := 1111;
        B := '1111';
    end;
```

Since there are not explicit access methods for properties *A* and *B*, they are internally implemented via fields. Property *A* is declared **readonly**. However, when you access a property with no explicit access methods from inside the declaring class, the compiler translate property references as field references. That's how you can still initialize *A* inside the constructor: actually, the assignment is done on the field. By the way, this is how events with automatic implementations are handled both by C# and Freya.

In contrast with C#, Freya allows both named and anonymous indexers. This is how an anonymous indexer property is declared:

```
Dictionary = class[Key, Value]
public
    property Dictionary[K: Key]: Value;
    ⋮
end;
```

Note that we use the class name for the default class indexer. The implementation of these properties is predictable, once you know how scalar properties are implemented in Freya:

```
implementation for Dictionary[Key,Value] is

    property Dictionary(K: Key): Value;
    begin
        ⋮
    end;

    property Dictionary(K: Key; V: Value);
    begin
        ⋮
    end;
```

Events are even easier to declare:

```
StackEvent = method (Sender: object; E: EventArgs);

IntStack = class
public
    ⋮
    event Pushing: StackEvent;
```

```
    event Pushed: StackEvent;
end;
```

If we don't provide accesors, Freya generates its own internal implementation. But we can provide our own accesors:

```
implementation for IntStack is

    event Pushing.Add(Value: StackEvent);
    begin
        ⋮
    end;

    event Pushing.Remove(Value: StackEvent);
    begin
        ⋮
    end;
```

As all .NET languages rule, you have to provide both an *Add* and a *Remove* accesor when you decide to explicitly implement an event.

Constructors

Delphi Pascal is a language with named constructors. On the other hand, the CLR only allows anonymous constructors: all constructor methods should name *.ctor*, and they must be distinguished only by their signatures. If we had to allow named constructors in Freya, we should add dummy hidden parameters to their declarations, in order to enforce distinct signatures for them all, if needed. That would translate into an unacceptable toll at runtime, so Freya constructors have no names.

This code fragment shows a class with two overloaded constructors:

```
IntStack = class
    constructor;
    constructor(Capacity: Integer);          // No overload directive!
end;
```

In their shortest forms, neither constructor declarations nor implementations require a name. However, you can use the class name as an acceptable alternative:

```
IntStack = class
    constructor IntStack;
    constructor IntStack(Capacity: Integer);
end;
```

Both alternatives are equivalent. In long code listings, it may be useful to repeat the class name in constructors. We're following a general principle, already found in other popular languages as C++: whenever there's a need for an anonymous feature at the class level, the feature is actually represented using the same name as the class. C#/C++ cannot simply drop the identifier because they have no syntactic markers such as **constructor**, **iterator** or **method**.

There's a difference between C++ and Freya, regarding the implementation of constructors. In C++, the class name is repeated in the constructor signature:

```
IntStack::IntStack(int capacity)             // C++
```

But this is a consequence of C++'s hybrid nature. Since there are no first-level procedures in Freya, and also as a consequence of having a distinctive marker for constructors, we can avoid repeating the class name:

```
constructor IntStack(Capacity: Integer);     // Freya
begin
    Items := new array[Integer](Capacity);   // More on this soon…
end;

constructor: Self(128);                      // Droping the class name
begin
end;
```

The second constructor is implemented with a call to another constructor from the same class. The syntax for calling a *parallel constructor*, as before, resembles the syntax used in

C++ for this task. Of course, a constructor can call explicitly an inherited constructor, like this one:

```
constructor TForm(AOwner: TComponent): Inherited(AOwner);
begin
end;
```

Please note that *Inherited* is no longer a keyword. *Inherited* and *Self* play a similar syntactic role and both of them are object references.

All these constructors initialise new instances from the class. Freya also supports *class constructors*, for initialising static fields. Class constructors cannot be called explicitly, and you can have only one class constructor per class, with no parameters. Since they are an implementation feature, they are not declared with the class declaration, but are directly implemented inside the implementation section of the class.:

```
namespace MathTools;
public

    MathTools = class
    static public
        Pi: Double; readonly;
    end;

implementation for MathTools is

    class constructor MathTools;
    begin
        ⋮
    end;

end.
```

You can also drop the class name when implementing a class constructor:

```
implementation for MathTools is

    class constructor;
    begin
        ⋮
    end;

end.
```

Instantiation

As you have already seen, Freya uses the ***new*** operator in order to create class instances, in a very C# fashion. That's a consequence of having unnamed constructors:

```
var L: Resistor;
begin
    L := new Resistor(45000);
    ⋮
end;
```

Freya provides extended instantiation syntax to simplify instantiation and initialization. Frequently, we need to create an object instance and perform some assignments to its properties and fields in order to complete the initialization:

```
var DS: DataSet;
begin
    DS := new DataSet('MyDataSet');
    DS.CaseSensitive := false;
   ⋮
end;
```

We can merge the whole initialization pattern in a single instruction:

```
var DS: DataSet;
begin
    DS := new DataSet('MyDataSet', CaseSensitive := false);
   ⋮
end;
```

Field and property initializers may follow proper parameters in a constructor call, resembling the named parameters feature found in some other languages. In the following example, a call to a parameterless constructor is followed by two property initializers:

```
var P := new Point(X: 0, Y: 0);
```

As the above example shows, you can also use a semicolon to separate the field or property name from the initialisation expression.

Local variable declarations

Freya allows inline local variable declarations, in true C++/C# fashion. Consider this:

```
var L: Resistor;
begin
    L := new Resistor(45000);
    ⋮
end;
```

First, we declare a variable using a type descriptor. When it's time to create an instance for this class, the type name must be repeated. Freya accepts this alternative:

```
begin
    var L := new Resistor(45000);
    var C: ElectricComponent := new Resistor(1000);
    ⋮
end;
```

The variable declaration has been merged with the initialization. In the first instruction, the variable's type is inferred from the right expression's type, as in C# 3.0. You could also provide an explicit type for the local declaration, as in the second instruction. In this case, we assume *ElectricComponent* is an ancestor of *Resistor*, or an interface type implemented by the latter.

Of course, the new **var** statement can use any kind of expression for its initializer expression:

```
begin
    var I := 0;
    var S: String := I.ToString;
    ⋮
end;
```

Local variables declared by a **var** statement are only accessible inside the same block they are declared, and from the declaration point on:

```
begin
    begin
    ⋮
    // Access to I is not allowed here.
    var I := 0;
    // Access to I is allowed from here on.
    ⋮
    end;
    // Access to I is not allowed here.
end;
```

We can even merge block declaration and local variable declaration:

```
begin
    // Variable type is inferred:
    with var I := 0 do
    begin
        ⋮
    end;
    // Variable type is explicitly declared:
    with J: Integer := 34 do
    begin
        ⋮
    end;
```

```
    // Several variables declared in the same block:
    with I: Integer := 0, var J := I + 1 do
    begin
        ⋮
    end;
end;
```

Freya's **with** is by no means related to Pascal **with**.

Genericity

Generic types are supported by the CLR in .NET v2.0. Our specification runs along with C#'s in most aspects. We have only changed some minor syntactic elements. For instance, Freya uses square brackets for enclosing both formal and real type parameters, instead of angular brackets. Consider this C# type reference:

```
Stack<Stack<int>>                    // C#
```

The two closing angular brackets can be confused by the lexical analyser with a C/C++/C# right shift operator. Of course, Freya could use ***shr*** and ***shl*** for bit shifting, as in Delphi, but even then, angular brackets have no other use in Pascal. Fortunately, there are other languages from the Algol/Pascal family which support genericity, and most of them use square brackets for this purpose:

```
Stack[Stack[Integer]]                // Freya
```

Another syntactic enhancement can be applied when a generic class depends on just one parametric type. When instantiating a type with real parameters, you could use the *X **of** Y* syntax, like in old Pascal's *array **of** Y* or *set **of** Y*:

```
var
    S1: Stack of Integer;
    S2: Stack[Integer];
    A1: Array of Integer;
    A2: Array[Integer];
```

This variant cannot be used when there are two or more type parameters, as this example shows:

```
A[X[Y], Z]
A[X of Y, Z]                         // Ok
```

If we attempt to substitute the outer brackets in the previous example with our alternative syntax, we end up with an ambiguous construction:

```
A of X of Y, Z                       // Error!
```

The ***of*** construction has right associativity. The following type reference:

```
Array of Array of System.Integer
```

is interpreted as:

```
Array[Array[System.Integer]]
```

You cannot use ***of*** in association with the ***new*** operator:

```
A := new Array of X(35);             // Error!!!
A := new Array[X](35);               // Ok
A := new Array[Set of Byte](2);      // Ok again
```

Freya implements the same feature set as C# and Visual Basic.NET. For instance, it supports constrained genericity:

```
IHashable = interface
    method Hash: Integer;
end;

HashTable = class[X(IHashable)]
    ⋮
end;
```

Note that we avoid a separate constraint section, as the **where** clause from C#. You could require a real type parameter either to have a given class as its ancestor, or to implement

one or more interface types, or to have a default public constructor, or a combination of these requirements:

```
MyGenericClass = class[class X(IMyIntf, new)]
    ⋮
end;
```

The above example shows a generic class that can only be instantiated with class types implementing the *IMyIntf* interface and having a default constructor.

Another important feature related to genericity is the support for *nullable types*. Nullable types are value types constructed from non nullable value types. They add one more value to the base type domain: the **nil** constant. They are useful for emulating the SQL expressions semantic:

```
var age: Integer? := 30;
if age.HasValue then
    Console.WriteLine(age.Value);
age := nil;
if not age.HasValue then
    Console.WriteLine('Ok!');
```

Freya provides a binary coalescence operator:

```
Console.WriteLine(age ?? 0);
```

This operator returns its first operand when it does not contain a null; otherwise, it returns the value from its second operand. Through this operator was designed along with nullable types, you can use it with traditional reference types, as in this example:

```
var String: s := nil;
var String: t := s ?? String.Empty;
```

Freya also implements lifted operators for handling expressions involving nullable types:

```
var x: Double? := nil, y: Double? := 1;
var z: Double? := x + y;
Console.WriteLine(z);
```

Any expression containing at least one null value, automatically evaluates as null.

Iterators

Iterators were added to C# in version 2.0, for simplifying the design and implementation of the *IEnumerable* interface type. This interface helps to define *open iterators*, which can be used directly, by calling the individual methods from this interface, or indirectly, by using the ***foreach*** statement in C#. In C# version 1.0 and 1.1, you had to create a class which implemented *IEnumerator* and provided state for iteration.

This is an elementary example of a C# 2.0 iterator:

```
// C# v2.0
public static IEnumerable<int> Range(int lo, int hi)
{
    while (lo <= hi)
        yield return lo++;
}
```

And this is how we would use this iterator using a ***foreach*** statement:

```
// C#
foreach (int i in MyClass.Range(0, 10)) {
    ⋮
}
```

The Freya equivalent for the previous iterator is implemented this way:

```
iterator MyClass.Range(Lo, Hi: Integer): Integer;
begin
    for Result := Lo to Hi do
        Yield;
end;
```

Freya use a variant of the ***for*** statement for consuming the iterator:

```
for I in MyClass.Range(0, 9) do
begin
    ⋮
end;
```

There is enough syntactic information to tell between a traditional ***for*** statement and a loop based on an iterator.

Freya iterators have several advantages when compared to C# iterators:

1. We have added a new executable feature type: ***iterator***, as we already have ***method***, ***constructor*** and ***destructor***. Thanks to this, iterators can be easily identified by the programmer. On the contrary, you have to watch ***yield*** statements and method return types in order to locate iterators in C#.
2. Freya iterators are not linked to a specific implementation technique, as C# iterators are. However, since there is an exact translation from a Freya to a C# iterator, the former could be, at least, as efficient as the latter. But you could event implement a Freya iterator as a closed iterator, if needed.
3. There's no need for a ***yield break*** feature, as in C#. You could end iteration with an *Exit* procedure call, as in a normal procedure.

Another possibility is to declare an anonymous iterator for a class. For instance:

```
LinkedList = class[X]
public
    // Constructor declaration included for comparison
    constructor LinkedList;
    // This is the anonymous iterator
    iterator LinkedList: X;
    ⋮
end;
```

As with other similar features, like constructors, an anonymous iterator is declared using the class name. When it comes to its implementation, the class name is included only once:

```
iterator LinkedList: X;
begin
    ⋮
end;
```

Freya even allows you to declare a static anonymous iterator for a class. As a matter of fact, enumerative types already have a predefined static iterator in Freya:

```
for var day in DayOfWeeks do
    // ... whatever ...
```

For a *dense* enumerative type, the previous statement is translated this way:

```
for var day := DayOfWeeks.First to DayOfWeeks.Last do
    // ... whatever ...
```

But even for a sparse type, we could still use the static iterator. The internal implementation uses an internal table created by the compiler for this purpose.

Exceptions

There are no great differences between Freya and Delphi on exception support. Freya introduces a new ***try***/***fault*** statement:

```
try
    DoWhatever;
fault
    CleanIfError;
end;
```

The previous statement is semantically equivalent to this one:

```
try
    DoWhatever;
except
    CleanIfError;
```

```
    raise;
end;
```

These statements could be used with common patterns like explicit transaction processing:

```
trans := connection.BeginTransaction;
try
    ModifyDatabase;
    trans.Commit;
fault
    trans.Rollback;
end;
```

The rationale behind ***try/fault*** is the fact that, in a well written application, most ***try/except*** statements already include a ***raise*** as the last statement of the ***except*** clause. This fact is also recognised by .NET, which supports a ***fault*** block in its Intermediate Language.

There's another little difference in how exceptions traps are declared:

```
try
    Statement1;
    Statement2;
except E: CryptoException do
    Statement3;
    Statement4;
except E: IOExceptions do
    Statement5;
except
    MoreStatements;
end;
```

The original Delphi syntax is absurd: the ***try/except*** statement follows the modern block pattern, but the old style is restored when you have to write an embedded ***on/do*** clause for trapping exceptions by class.

Freya allows ***try/except/finally*** blocks:

```
try
    FirstAttempt;
except
    SecondAttempt;
finally
    CommonCleaning;
end;
```

This compound exception block will be seldom used, but it's kept in order to make code translation from other .NET languages easier.

Exceptions in Freya are raised as in Delphi:

```
raise new ApplicationException('Uh-oh');
raise alreadyCreatedInstance;
```

The first instruction raises an exception and creates a new instance for holding information on why this exception has been fired. The second instruction uses an already existing instance as the exception object.

Deterministic destruction and block declarations

.NET languages heavily rely on the *IDisposable* pattern in order to achieve deterministic disposal of resources other than memory. C# introduces the ***using*** statement for coding this pattern. We have also added a ***using*** statement, with two variants. Most of the times, **using** declares and initializes a local variable

```
using X: MyClass := new MyClass do
    WhatEver;
// X is not available here
```

The translation depends whether *MyClass* implements the *IDisposable* interface or not. When the answer is yes, the translation looks like this:

```
begin
    var X: MyClass := new MyClass;
    try
        WhatEver;
    finally
        IDisposable(X).Dispose;
    end;
end;
// X is not available here
```

This is how C# interprets a **using** statement, but Freya also allows this statement even if *MyClass* does not implement *IDisposable*. In that case, no **try**/**finally** block is generated, and the statement just introduces a block with a local variable declaration:

```
begin
    var X: MyClass := new MyClass;
    WhatEver;
end;
// X is not available here
```

We can also drop the explicit type declaration, to allow the compiler to infer the type of the local variable:

```
using var X := new MyClass do
    WhatEver;
```

If the code enclosed by **using** does not need to access the declared variable, we could drop the variable declaration:

```
using new MyClass do
    WhatEver;
```

This variant, also accepted by C#, only makes sense if *MyClass* implements *IDisposable*. In that case, the compiler declares a hidden variable for disposing the new instance at the end of the block:

```
begin
    var _hidden: MyClass := new MyClass;
    try
        WhatEver;
    finally
        IDisposable(_hidden).Dispose;
    end;
end;
```

Once we have accepted a limited form of nested scope for local variables, it makes sense to add a similar feature to ***for*** statements:

```
for N: TreeNode in ATree.PreOrder do
begin
    ⋮
end;

for I: Integer := 0 to Count - 1 do
begin
    ⋮
end;
```

Type inference is also permitted here:

```
for var N in ATree.PreOrder do
begin
    ⋮
end;

for var I := 0 to Count - 1 do
begin
    ⋮
end;
```

Operators and compound assignments

The C programming language, and C++ later, was designed with performance as one of their primary goals. If performance is vital for a language which compiles to native code,

it's even important when you have a virtual machine as the target, no matter how optimised it might be. This way, most of the performance advantage gained by special operators and assignments is still present in C#.

On the contrary, Pascal inspired languages have always put the emphasis on readability and simplicity. Anyway, Delphi Pascal already includes special operations resembling some of the special operators from C/C++:

```
Inc(TotalQty, Qty);                         // Delphi
Include(days, DOW.Monday);
days := days + [DOW.Monday, DOW.Tuesday];
```

With .NET arrival, Delphi.NET had to extend some of these methods in order to support multicast events:

```
Include(button1.Click, button1Click);     // Delphi.NET
```

One of the problems with Delphi's approach is that *Include*, *Exclude*, *Inc* and *Dec* must be interpreted as global first-level procedures, a feature not supported by Freya. Another problem has to do with uniformity: *Inc* can be used with integer types, but not with real types.

These are some of the reasons why Freya incorporates some of the C# special operators and compound assignments. We have now increment and decrement statements, substituting the monoparametric *Inc* and *Dec* versions:

```
Count++;                                    // Freya
Remaining--;
```

Notice that we have said *statement*, not *expression*. This is not allowed in Freya:

```
if Count++ < MAX then                       // Invalid in Freya!!!
    Remaining--;
```

Another restriction is that we have only included the suffix form. Since the increment and decrement statements can be applied to object paths, which are best read from left to right, it's more natural to write the increment operator at the right, after navigating to the last path element.

If we allow post-increments and decrements, it's also natural to allow compound assignments as substitutions for *Include*/*Exclude* and the two parameters versions of *Inc* and *Dec*:

```
button1.Click += button1Click;              // Freya
days += DOW.Monday;
days += [DOW.Monday, DOW.Tuesday];
```

Note that we have two overloaded versions for the += operator, when applied to a set variable. Again, compound assignments are not considered expressions.

Once we have compound assignments for addition and subtraction, there's no reason to exclude other compound assignment operations... except for one thing: how should we write a compound assignment involving a remainder operation, which is performed in Delphi with the traditional **mod** operator? Our answer is to allow synonyms for those literal Pascal operators:

```
i := i mod 2;               // Still valid in Freya
i := i % 2;                 // Now, this is also valid
i %= 2;                     // A valid compound assignment
i := i \ 2;                 // A valid synonym for the integer division (div)
```

We have been less kind with the bit shift operators from Delphi: ***shr*** and ***shl***. C's >> and << operators are a lot more expressive, and they don't assume you are a native English speaker.

There has been a major change regarding the very unpractical operator precedence from Pascal. Now, logical operators have been moved to the appropriated level, so you can write expressions like this:

```
while Current < Input.Length and Char.IsWhiteSpace(Input, Current) do
    Current++;
```

We have also introduced two new logical operators, with lower precedence than comparisons: the logical implication and the logical equivalence.

```
requires
    Active -> RowCount >= MINROWS;
```

The new `->` operator stands for the logical implication, and the previous expression can be rephrased as *"when Active is true, RowCount should be greater or equal to MINROWS"*. The Delphi equivalents are:

```
not Active or (RowCount >= MINROWS)
Active <= (RowCount >= MINROWS)
```

The first definition is preferred, because it allows short circuit evaluation. The precise definitions for the new operators are:

```
A ->  B        ≡       not A or B
A <-> B        ≡       A = B
```

Logical equivalence doesn't allow short circuit evaluation: the operator always needs to evaluate both operands. These two operators are a must when writing *assertions*, a feature we'll cover immediately. Eiffel, the paradigmatic language with assertions, has an ***implies*** operator, but we have preferred a symbolic representation, saving two reserved words.

There are other minor enhancements in order to make Freya programming an enjoyable task. For instance, we have introduced a compound operator for the negation of the set membership test:

```
repeat
    ⋮
until Input[Current] not in ['0'..'9'];
```

It looks like SQL, but we are sure it will be welcomed by most Freya programmers.

User defined operators and conversions

Freya implements first class support for user defined operators. You can use both the method and operator markers when defining a symbolic operator:

```
// The ortodox syntax, identical to C#'s.
operator+(c1, c2: Complex): Complex;
// The substraction operator is a symbolic one, so "method-" is ok.
method-(c1, c2: Complex): Complex;
```

The **operator** marker is the only option when the operator name is an identifier, as in user defined conversions:

```
operator Explicit(C: Complex): Double;
begin
    Result := c.Re;
end;

operator Implicit(Value: Double): Complex;
begin
    Result := new Complex(Value, 0);
end;
```

Assertions

You can find a full description of the philosophy behind assertions and programming by contract in Bertrand Meyer's *Object Oriented Software Construction*. Even though most of Freya's assertions support has been borrowed directly from Eiffel, some details had been adjusted in the translation process.

The first issue has to do with the way Pascal splits declaration from implementation. Where does pre- and post-conditions belong? Let's start with preconditions. Since preconditions must be verified by the caller, at least in theory, they must be available to it. A precondition for a public method, for instance, must be declared along with the method declaration, and may only refer to other public features from the class. A precondition for a protected method may only refer to other public or protected features, and it also must be de-

clared along with the method declaration. So, this solves the location for declaring preconditions: they must be always declared inside the class declaration. For instance:

```
Stack = class
public
    method Pop;
        requires not IsEmpty;
    ⋮
end;
```

Post-conditions are more complex. You can have public or declared post-conditions included in the class declaration. These declared post-conditions may only refer to features with same or wider visibility attribute. Declared post-conditions are useful to the class clients, because they explain *how* the class does its work. Of course, if one of these post-conditions fails, it would mean a failure from the class code, and there's nothing the client could do, except sending a bug report to the technical support. This is an example of a declared post-condition:

```
Stack = class
public
    method Push(V: X);
        ensures not IsEmpty,
                Top = V: push_on_top;
    ⋮
end;
```

But there are non-declared post-conditions too, like this one:

```
method Push(Value: X);
begin
    Items[Count] := Value;
    Count++;
ensures
    Count = old Count + 1;
end;
```

This post-condition refers to some internal features, as the number of items in the stack, and they must be declared after the method implementation, as shown.

Properties pose another problem related to assertions. Each property may have two associated methods for its implementation. This way, a read/write property could support two different preconditions and postconditions. Non public postconditions must be stated when implementing the corresponding accesor, but there's still a problem when read and write accesors needs different preconditions, and when they ensures distinct public postconditions.

```
property Caption: string;
    requires Value <> '' for write;
    ensures  Value <> '' for read;
```

Assertions may have a name, as this example shows:

```
property Top: X; readonly;
    requires not IsEmpty: not_empty;
```

Instead of an identifier, you can associate a string literal to the assertion:

```
property Top: X; readonly;
    requires not IsEmpty: 'Cannot ask for Top when the stack is empty';
```

This rather long example summarises most of the features already discussed:

```
namespace Freya.DataStructures.Stacks;

public

    Stack = class[X]
    protected
        Items: Array[X];
        Count: Integer := 0;
    public
        constructor;
        constructor(Capacity: Integer);
        iterator Stack: X;
```

```
        property Top: X; readonly;
            requires not IsEmpty;
        property IsEmpty: Boolean; readonly;
        method Pop;
            requires not IsEmpty : not_empty;
        method Push(V: X);
            ensures not IsEmpty,
                    Top = V : push_on_top;
    invariant
        Count >= 0;
    end;

implementation for Stack[X] is

    constructor(Capacity: Integer);
    begin
        new Items(Capacity);
    end;

    constructor: Self(128);
    begin
    end;

    iterator: X;
    begin
        for I: Integer := Count - 1 downto 0 do
        begin
            Result := Items[I];
            Yield;
        end;
    end;

    property Top: X;
    begin
        Result := Items[Count-1];
    end;

    property IsEmpty: X;
    begin
        Result := Count = 0;
    end;

    method Pop;
    begin
        Count--;
    end;

    method Push(Value: X);
    begin
        Items[Count] := Value;
        Count++;
    ensures
        Count = old Count + 1;
    end;

end.
```

As you can see, class invariants can be declared using a separate section inside the class declaration.

Interfaces as contracts

A very interesting interaction takes place between interface types and assertions. Interfaces are abstract contracts that don't impose any constraint regarding the physical layout of the implementers. Since assertions are a formal technique for contract specification, it makes sense to associate assertions to interface declarations:

```
    IStack = interface[X]
        property Top: X; read;
            requires not IsEmpty;

        property IsEmpty: Boolean; readonly;

        method Pop;
            requires not IsEmpty : not_empty;
```

```
        method Push(V: X);
            ensures  not IsEmpty : push_fills_stacks,
                     Top = V     : push_goes_top;
    end;
```

Every class implementing *IStack* must honour these assertions. The implementing class does not have to repeat the assertions in its declaration, since they are added automatically by Freya.

There is a small but interesting detail in how explicit interface implementations are written in Freya:

```
    ISource = interface
        property IsDone: Boolean; readonly;
        method NextChar: Char;
        event NewLine: NewLineEventHandler;
    end;

    SourceFile = class(ISource)
        ⋮
    end;
```

Despite what the ellipsis could suggest, in a complete example there would not be any hint in the class declaration about the explicit implementation of *ISource*. The interface implementation would be totally confined to the implementation block:

```
implementation for SourceFile is

    property ISource.IsDone: Boolean;
    begin
        ⋮
    end;

    method ISource.NextChar: Char;
    begin
        ⋮
    end;

    ⋮
end.
```

There are no hints in the class declaration about how *ISource* is implemented, but that's fine: these are implementation details, and the class contract should not mess with them. Something similar happens when implementing the event declared by the interface type:

```
    event ISource.NewLine.Add(Value: NewLineEventHandler);
    begin
        ⋮
    end;
```

An extreme case of this design principle has to do with implementing interfaces by delegation, a trick borrowed from Delphi, not allowed in C#:

```
    CustomerForm = class(System.Windows.Forms.Form, IPrintable)
    private
        Grid: SuperbGridView; // This class implements IPrintable
        ⋮
    end;
```

The link between the *Grid* field and the *IPrintable* interface is relegated to the class' implementation block:

```
implementation for CustomerForm is

    interface IPrintable = Grid;

    ⋮
end.
```

Once more time, there are no hints in the class declaration about how is the interface implemented.

Current state of Freya implementation

We already have a working compiler for Freya. It already implements all features found in .NET 1.0-1.1, almost all features from .NET and C# 2.0 and even some novelties from the C# 3.0 specification. The compiler is implemented with pure C# 2.0, so we still have the possibility to translate the entire source to Freya, if needed.

The parser is a LALR(1) one. Its tables are generated with GOLD Parser, a freeware tool developed by Devin Cook, and available at www.devincook.com. We use the XML export command from GOLD Parser to generate our own parsing tables for C#. These tables are interpreted by a custom parsing engine created by the authors.

The parsing engine builds an Abstract Syntax Tree from the input source, and the rest of the compiling passes are performed on this data structure. In a first semantic pass, type references are bound to external types defined by the referenced CLR assemblies. A second pass takes computes types for expressions, bind overloaded features and performs most semantic checks.

Our compiler generates code using *Reflection.Emit*. This has been a mixed blessing, since there found several problems with these classes, especially when generating generic types and multidimensional arrays. We are generating the best possible code in most cases, and we pretend to add as many sensible optimizations as possible.

Ian Marteens
ian@intsight.com

APPENDIX A: FREYA RESERVED WORDS

abstract	**fault**	**nil**	**shl**
and	**finally**	**not**	**shr**
as	**for**	**of**	**sealed**
begin	**goto**	**old**	**static**
case	**if**	**operator**	**then**
class	**implementation**	**or**	**to**
const	**in**	**out**	**try**
constructor	**interface**	**override**	**until**
destructor	**internal**	**private**	**using**
div	**invariant**	**property**	**var**
do	**is**	**protected**	**virtual**
downto	**iterator**	**public**	**while**
else	**loop**	**raise**	**with**
end	**method**	**readonly**	**xor**
ensures	**mod**	**record**	
event	**namespace**	**repeat**	
except	**new**	**requires**	

Special procedure identifiers: *Exit*, *Break*, *Continue*, *Yield*.
Special object references: *Self*, *Inherited*.
Special variable: *Value* (only allowed in a property's declared pre- or post-condition).
Special event access indicators: *Add*, *Remove* (only in explicit event implementations).

APPENDIX B: A SIMPLE SORTED BINARY TREE IN FREYA

```
using System;

namespace Freya.DataStructures.Trees;

public

    BinaryTree = class[X(IComparable)]
    protected
        TreeNode = class
        public
            Value: X;
            Left, Right: TreeNode;

            constructor TreeNode(Value: X);
            method Contains(Value: X): TreeNode;
            iterator Preorder: X;
        end;

        Root: TreeNode;
    public
        method Add(Value: X);
        method Contains(Value: X): Boolean;
        iterator Preorder: X;
    end;

implementation for BinaryTree[X].TreeNode is

    constructor TreeNode(Value: X);
    begin
        Self.Value := Value;
    end;

    method Contains(Value: X): TreeNode;
    begin
        if Value.CompareTo(Self.Value) = 0 then
            Result := Self
        else if Value.CompareTo(Self.Value) < 0 then
            if Left <> nil then
                Result := Left.Contains(Value)
            else
                Result := nil
        else if Right <> nil then
            Result := Right.Contains(Value)
        else
            Result := nil;
    end;

    iterator Preorder: X;
    begin
        Result := Value;
        Yield;
        if Left <> nil then
            for Result in Left.Preorder do Yield;
        if Right <> nil then
            for Result in Right.Preorder do Yield;
    end;

implementation for BinaryTree[X] is

    method Contains(Value: X): Boolean;
    begin
        Result := Root <> nil and Root.Contains(Value) <> nil;
    end;

    method Add(Value: X);
    var
        Parent, Current: TreeNode;
    begin
        Parent := nil;
        Current := Root;
        while Current <> nil and Value.CompareTo(Current.Value) <> 0 do
        begin
            Parent := Current;
```

```
            if Value.CompareTo(Current.Value) < 0 then
                Current := Current.Left
            else
                Current := Current.Right;
        end;
        if Current = nil then
        begin
            Current := new TreeNode(Value);
            if Parent = nil then
                Root := Current
            else if Value.CompareTo(Parent.Value) < 0 then
                Parent.Left := Current
            else
                Parent.Right := Current;
        end;
    end;

    iterator Preorder: X;
    begin
        if Root <> nil then
            for Result in Root.Preorder do
                Yield;
    end;

end.
```

APPENDIX C: A TOP-DOWN PARSER FOR NUMERIC EXPRESSIONS

```
using System, System.IO;

namespace Freya.Examples.Parser;
public

    Parser = class
    protected
        Lexer: Lexer := nil;
        method Expression: Double;
        method Term: Double;
        method Factor: Double;
    public
        method Compile(Input: String): Double;
    end;

private

    Tokens = (Number, Plus, Minus, Times, Divide, LPar, RPar, Eof);

    Lexer = class
    public
        constructor Lexer(Input: String);
        property Token: Tokens; readonly;
        property Value: Double; readonly;
        method Next;
    end;

implementation for Lexer is

    var
        Input: String;
        Current: Integer;

    constructor Lexer(Input: String);
    begin
        Self.Input := Input;
        Self.Current := 0;
        Self.Next;
    end;

    method Next;
    begin
        Value := 0;
        while Current < Input.Length and
            Char.IsWhiteSpace(Input, Current) do
            Current++;
        if Current >= Input.Length then
            Token := Tokens.Eof
        else
        begin
            case Input[Current] of
                '0'..'9':
                begin
                    Token := Tokens.Number;
                    repeat
                        Value := Value * 10 + Input[Current].Ord - '0'.Ord;
                        Current++;
                    until Current >= Input.Length or
                        Input[Current] < '0' or Input[Current] > '9';
                end;
                #0 : Token := Tokens.Eof;
                '+': Token := Tokens.Plus;
                '-': Token := Tokens.Minus;
                '*': Token := Tokens.Times;
                '/': Token := Tokens.Divide;
                '(': Token := Tokens.LPar;
                ')': Token := Tokens.RPar;
            else
                raise new Exception('Invalid character: ' + Input[Current]);
            end;
            if Token not in [Tokens.Eof, Tokens.Number] then
                Current++;
```

```
        end;
    end;

implementation for Parser is

    method Compile(Input: String): Double;
    begin
        Lexer := new Lexer(Input);
        try
            Result := Expression;
            if Lexer.Token <> Tokens.Eof then
                raise new Exception('Invalid expression ending');
        finally
            Lexer := nil;
        end;
    end;

    method Expression: Double;
    begin
        Result := Term;
        var T := Lexer.Token;
        while T in [Tokens.Plus, Tokens.Minus] do
        begin
            Lexer.Next;
            if T = Tokens.Plus then
                Result += Term
            else
                Result -= Term;
            T := Lexer.Token;
        end;
    end;

    method Term: Double;
    begin
        Result := Factor;
        var T := Lexer.Token;
        while T in [Tokens.Times, Tokens.Divide] do
        begin
            Lexer.Next;
            if T = Tokens.Times then
                Result *= Factor
            else
                Result /= Factor;
            T := Lexer.Token;
        end;
    end;

    method Factor: Double;
    begin
        if Lexer.Token = Tokens.Number then
        begin
            Result := Lexer.Value;
            Lexer.Next;
        end
        else if Lexer.Token = Tokens.LPar then
        begin
            Lexer.Next;
            Result := Expression;
            if Lexer.Token <> Tokens.RPar then
                raise new Exception('Right parenthesis expected');
            Lexer.Next;
        end
        else
            raise new Exception('Number or left parenthesis expected');
    end;

implementation

    method Main;
    begin
        with var P := new Parser do
        loop
            Console.Write('? ');
            with var S := Console.ReadLine do
            try
                if String.IsNullOrEmpty(S) then Exit;
```

```
                    var d := P.Compile(S);
                    Console.Write('= ');
                    Console.WriteLine(d);
                except e: Exception do
                    Console.WriteLine(e.Message);
                end;
            end;
        end;

end.
```

CONTENTS