

PHP/CLR

Extensions to the PHP Language in Phalanger 2.0

Tomas Matousek, Ladislav Prosek

Draft version 1.0, July 2006

1. Overview

PHP/CLR (PHP for Common Language Runtime) is a language introduced by version 2.0 of Phalanger – the PHP Language Compiler for .NET Framework [2]. It is virtually a superset of PHP4, PHP5 and PHP6 languages supplemented with extensions targeting *Common Language Runtime (CLR)* interoperability, making the PHP/CLR a first class member of the CLR language family [1].

The PHP language [4] evolves very quickly in time often without precise specification of the features it provides. The current version is 5.2 and the next major one, version 6.0, is on the way. It implements Unicode strings, goto statement, 64-bit integer, etc. Besides, an experimental branch of the version 5 that supports namespaces is available.

Tracking the proposals and announcements of the PHP interpreter developers, we design the PHP/CLR language to be as compatible with the existing and future versions of PHP as possible. The ultimate acceptability criterion for the PHP/CLR language and its implementation in Phalanger is an ability to run vast array of existing PHP code without significant modifications (ideally without any modifications) unless it relies on apparent misbehavior of the interpreter or deprecated features.

All novel features of the PHP/CLR language were carefully considered before introduced with the aim to keep the language consistent, to preserve its dynamical nature and to keep the number of novelties at minimum while providing all constructs necessary to be able to consume and produce first class CLR libraries.

This document describes the features already implemented in Phalanger version 2.0 Beta 1 and also proposes a few more which are to be implemented in the next Beta releases. As the Phalanger is still in Beta status, it is possible that some of them could change in details. This document should serve as a basis for a discussion over the proposed solutions, not as a definite and final specification of the PHP/CLR language. However, the overall design of the language extensions should be more or less stable. The extensions include:

- *Namespaces.* The syntax and semantics in PHP/CLR is almost the same as the one proposed and implemented in PHP5 namespace experimental patch by Jessie Hernandez [REF]. Although the PHP community disputes over usefulness of namespaces, they are apparently essential for the cooperation with the CLR. The support for namespaces in PHP/CLR includes not only referring to the existing CLR namespaces but also an ability for grouping PHP classes, functions and *global constants* (see the next bullet) into the namespaces (including *anonymous* ones). The technique of *namespace importing* is also available to make using namespaces convenient.
- *Global constants.* Global constants are syntactically similar to class constants, however, they are declared in a *global scope* (i.e. outside of any declaration). Global constants are introduced by PHP namespace patch. To make the language consistent with respect to global declarations, PHP/CLR supplements the functionality of the patch by an ability to import a global constant under an alias, which is indeed possible for classes and functions as well.
- *Partial type declarations.* To allow the type declarations to be scattered across multiple source files, PHP/CLR introduces *partial type declarations* inspired by the C# language version 2.0

[REF]. The main advantage of partial declarations reveals when some parts of the class are manually written and the other parts are tool generated, e.g. via *CodeDom* [REF].

- *Generics*. As of version 2.0, CLR types and methods can define generic type parameters. The PHP/CLR language, as a first class citizen, enables to consume generic types and methods as well as to produce them to a large extent (in some aspects, even to the larger than C# version 2.0). Since the PHP language is loosely typed, generics do not bring anything revolutionary to the PHP code itself, however, the notion of generic parameters and arguments is essential for seamless interoperability with CLR. The type arguments need to be specified, for example, when extending a generic CLR class, implementing a generic CLR interface, calling a CLR generic method etc.
- *Custom attributes*. Custom attributes are well known means for attaching additional information to various language elements (classes, properties, methods, formal parameters, generic parameters etc.). A custom attribute may drive the compiler to treat the language element in some special way or be consumed by third party tools reflecting the compiled code (the attributes are persisted to the resulting assembly). PHP/CLR defines a few special purpose attributes, allows to use attributes defined in CLR libraries and also to define new ones.
- *Property accessors*. The PHP language defines special (“magic”) methods `__get` and `__set`, which are called when an undeclared property (field in CLR terminology) is read and written, respectively. It doesn’t, however, provide syntax for declaring property accessors for a single property. PHP/CLR introduces these accessors in order to enable custom overriding of virtual CLR properties. Besides, PHP/CLR allows to override a CLR property by so called *field-backed* property, which simply reads and writes the field. The field-backed property is declared as an ordinary PHP class variable, the compiler takes care of the overriding formalities.
- *Parent constructor calls*. In CLR, a constructor of a superclass is required to be called in the constructor of the subclass (if there is any). In the PHP language, there is no such requirement. Hence, when deriving from the CLR class with a constructor but without a parameterless default constructor, the subclass is required to call the parent constructor. Such requirement is expressed by mandatory parent class constructor call.
- *Long integer*. PHP/CLR introduces 64-bit signed integer type, called *int64*. The PHP integer type is 32-bit. Both widths are independent of the platform. If an operation on integer overflows in PHP the type is changed to *double*. In PHP/CLR the *integer* overflows to *long integer* and then to *double* (with precision loss). The difference is only in the types and not in the values so the existing applications should be unaffected unless they count on the exact type.
- *Extended type hints*. In PHP, it is possible to attach a type hint to a formal parameter of a function or method. The type hint can be an arbitrary class name or *array*. The types are automatically checked at the entry to the function/method. Besides of these, PHP/CLR enables any *primitive type* (i.e. *bool*, *int*, *int64*, *double*, *string*, *resource*, *object* and *array*) to be specified as a type hint.
- *Extended type casts*. CLR allows a method to be overloaded by the types of the formal parameters. Since the types of the arguments needn’t to be known at compile-time, PHP/CLR defines an algorithm how to resolve the overload to be called at run-time according to the actual values of the arguments. In usual cases, the automatic selection is sufficient. However, it may be the case that the programmer wishes to call a different one. For such cases, PHP/CLR provides a set of type casts (one for each CLR primitive type) that provide a hint to the compiler, which overload to choose.
- *Quoted identifiers*. The interoperability among various languages targeting CLR requires one language to be able to consume elements of the other (types, methods, etc.) even if their names collide with keywords of the former language. For example, the *List* is a keyword in the PHP language (note that keywords are case-insensitive) and also *List* is a name of the class implementing a non-generic collection of items in *Base Class Library* (BCL). To be able to use names that are not valid identifiers in PHP language, the PHP/CLR provides syntax of *quoted identifiers*. For example, the *List* class can be referred to by `i'List'` quoted identifier.

- *Pragmas*. To enable a programmer or a tool to provide the compiler additional information anywhere in the source code, PHP/CLR comes with *pragmas*. The pragma is a single line comment opened by '#' character (which is a regular single line comment in PHP) followed immediately by '*pragma*' token. If such comment is found in the source code, its content is interpreted as a *directive* to the compiler. Currently supported directives include *line*, *default line*, *file* and *default file* altering the current source file location reported in compile-time and run-time errors and warnings.
- *LINQ*. [3]

2. Compiler Modes

Traditional PHP application comprises of a set of source files, which refer to each other via inclusions. The source file contains global code and declarations. The interpreter starts processing the statements of the global code one by one stepping into the global code of the included file when it encounters an inclusion and back when the global code of the included file returns. The inclusion targets can also be unknown at compile-time, which is very often case due to the way how PHP searches for them.

This approach is inconvenient for building component based applications and libraries. The main issue is the presence of global code as it is unclear when to run global code of the scripts constituting the library. The inclusions are also making things more difficult as the entire code of the library should be known at the compile-time.

Therefore, PHP/CLR distinguishes two modes of compilation. In the *standard mode*, all PHP features including the global code and inclusions are enabled. The standard mode is provided for backward compatibility with existing PHP applications. It is well suited to the PHP web applications and executable programs.

In the *pure mode*, the application comprises of a set of source files free of global code and inclusions. All declarations stated in any source file are visible from any other source file. The same effect would be reached by concatenation of all source files into a single file before the compilation. Note that the C# compiler works this way. Pure executable applications have to declare an entry point – a static parameterless function or method called *Main*. There can be at most one entry point in the source code of the application.

Most of the PHP/CLR features are available in both modes. The only one that is bound to the pure mode is the concept of partial type declarations as the compiler must know all parts of the declarations at compile-time.

3. PHP/CLR Syntax and Semantics

3.1. Identifiers and Names

Micro Syntax

identifier:

simple-identifier

quoted-identifier

simple-identifier:

[A-Za-z_][0-9A-Za-z_]*

quoted-identifier:

i'([\^\\<>`#\r\n] | (\\[\^<>`#\r\n]))+'

qualified-name:

qualified-name:::*simple-identifier*

simple-identifier

Semantics

1. *Simple identifier*. Non-empty sequence of English alphabet letters, underscores and digits not starting with a digit.
2. *Quoted identifier*. Non-empty sequence of arbitrary characters except for '<', '>', '#', back-quote '`', carriage-return and new-line characters. The identifier is enclosed in single quotes preceded by letter 'i'. The identifier can contain backslash escaped single quote character and backslash escaped backslash character.
3. *Qualified name*. Contiguous sequence of simple identifiers separated by the namespace separator ':::'.

Examples

1. *Quoted identifier*. The *array* is a PHP/CLR keyword referring to a PHP array primitive type (see [REF]). There are two possibilities, how to refer to the CLR array class (*System:::Array*) directly. The first uses the quoted identifier *i'Array'*:

```
<?
// Compilation Mode: standard
// References: PhpNetClassLibrary, mscorlib

import namespace System;
$a = i'Array'::CreateInstance(Type::GetType("System.Int32"), 10);
var_dump($a);
?>
```

The first line of the example imports the content of the CLR *System* namespace into the current naming context so that the *System:::Array* and *System:::Type* CLR classes become visible under the names “Array” and “Type”, respectively. The *CreateInstance* static method of the *System:::Array* class creates a new CLR vector of a specified type and length. The *GetType* static method of the *System:::Type* class takes a CLR name and returns the type object representing the type of that name. Note that in CLR type names, the namespace separator is a single dot while in PHP/CLR it is a triple-colon ':::'.

2. *Qualified name*. The other possibility to refer to the *System:::Array* type is simply to use a fully qualified name:

```
<?
```

```

// Compilation Mode: standard
// References: PhpNetClassLibrary, mscorlib

$a = System::Array::CreateInstance(System::Type::GetType("System.Int32"), 10);
var_dump($a);
?>

```

This time, we refer to all used CLR types by their fully qualified names so we needn't to import the *System* namespace.

3.2. Pragmas

Micro Syntax

```

pragma:
#pragma line positive-integer end-of-line
#pragma default line end-of-line
#pragma file path end-of-line
#pragma default file end-of-line

```

Semantics

1. *Pragma*. The *pragma* is a single line comment whose content is interpreted as a *directive* for the compiler.
2. *Line and file mappings*. The mapping is defined by pragmas with line mapping directives (*line* and *default line*) and file mapping directives (*file* and *default file*). When an error is reported (at compile-time or run-time) the immediately preceding line and file directives are determined.

If the line directive immediately preceding the error occurrence is the *line* directive, the line reported in the error is calculated as a difference of the actual position of the error occurrence and the actual position of the directive added to the line number specified by the directive. Hence, the number of the line where the directive appears is considered to be the number stated in the directive.

If the line directive immediately preceding the error occurrence is the *default line* directive or no line directive precedes the error occurrence at all, the line reported in the error is the actual line of the error occurrence.

If the file directive immediately preceding the error occurrence is the *file* directive, the source file path reported in the error is the verbatim *path* value stated in the directive. The value can contain any characters except for the end-of-line characters. Neither correctness nor existence of the file is checked.

If the file directive immediately preceding the error occurrence is the *default file* directive or no file directive precedes the error occurrence, the source file path reported in the error is the actual source file path of the error occurrence.

Example

```

<?
#pragma file parser.y
function DoAction(Stack $stack, $action)
{
    switch ($action)
    {
        case 1: #pragma line 20      E -> E + E
                $stack->Push($stack->Peek(1) + $stack->Peek(2));
                break;
        case 2: #pragma line 22      E -> E - E
                $stack->Push($stack->Peek(1) - $stack->Peek(2));
                break;
    }
}

```

```

1  program.php
2  parser.y
3  parser.y
4  parser.y
5  parser.y
6  parser.y
20 parser.y
21 parser.y
22 parser.y
23 parser.y
22 parser.y
23 parser.y
24 parser.y
25 parser.y

```

```

    }
    #pragma file default
    #pragma line default
?>

```

```

26 parser.y
27 parser.y
17 program.php
18 program.php

```

3.3. Custom Attributes

Syntax

```

custom-attributes:
    custom-attributes custom-attribute-section
    custom-attribute-section

custom-attribute-section:
    [ attribute-target-selectoropt custom-attribute-list ]

attribute-target-selector:
    attribute-target :

attribute-target:
    assembly
    module

custom-attribute-list:
    custom-attribute-list , custom-attribute
    custom-attribute

custom-attribute:
    qualified-name
    qualified-name ( positional-attribute-argument-list , named-attribute-argument-list )
    qualified-name ( positional-attribute-argument-list )
    qualified-name ( named-attribute-argument-list )

positional-attribute-argument-list:
    positional-attribute-argument-list , positional-attribute-argument
    positional-attribute-argument

positional-attribute-argument:
    constant-expression

named-attribute-argument-list:
    named-attribute-argument-list , named-attribute-argument
    named-attribute-argument

named-attribute-argument:
    variable => constant-expression

```

Semantics

1. *Custom attribute class.* *Custom attribute class* is a PHP or CLR class that derives (directly or indirectly) from *System.Attribute* CLR class (*System::Attribute* using PHP/CLR namespace syntax) and is not generic nor constructed. Besides, the custom attribute class must be resolvable at compile time. It is an error to use classes defined at run-time, *incomplete classes* (those deriving from compile time unknown parent or implementing unknown interface), or classes declared conditionally (so called *run-time activated classes*) as custom attribute types. Only a type meeting all these criteria can be referred to by its *qualified-name* in the custom attribute usage.

The name of the attribute class usually has the *Attribute* suffix. When resolving the custom attribute type in a custom attribute usage, the compiler first searches for the *qualified-name* with the *Attribute* suffix and if not found it searches for the *qualified-name* itself. For example, if the name of the custom attribute class is *ExportAttribute*, the *qualified-name* stated in the attribute usage can be *Export* or *ExportAttribute*.

2. *Custom attribute targets.* In general, a custom attribute can be applied on an assembly, module, global constant, function, class, interface, method, property, class constant, return value of a function or method, parameter and generic parameter. An attribute type can restrict the range of possible targets and multiplicity of the attribute usage on a single target. The restriction is defined by application of the *System.AttributeUsageAttribute* CLR custom attribute on the custom attribute type.
3. *Built-in custom attributes.* PHP/CLR defines a couple of built-in custom attributes known to the compiler that influence the compilation results. The first is the *ExportAttribute* and the next is the *AppStaticAttribute*.

The *ExportAttribute* can be applied on an assembly, class, interface, function, method and property. It cannot be applied multiple times. When applied on a function or a class member, it instructs the compiler to emit a set of additional methods/properties to the type that allows to access the member from other CLR languages more conveniently, hiding the specific features of the PHP language. The member becomes *exported*. If the attribute is applied on an entire class or interface the type itself and all its members become exported. If the attribute is applied on a module or assembly all types contained within the assembly become exported.

The *ExportAttribute* is available only in the pure mode and is especially useful for writing class libraries that are to be used by other CLR languages.

The *AppStaticAttribute* is applicable only on field-backed properties, i.e. for properties backed by storage. The PHP language allows to define instance properties and static properties. However, the semantics of a static property is in fact thread-static, which means the storage of the property is associated with the current thread (HTTP request). There is no way in PHP how to define a storage whose content is available during the entire application lifetime. The *AppStaticAttribute* allows to define such storage.

4. *Custom attribute arguments.* A custom attribute can be parameterized by positional and/or named parameters. The values of the parameters have to be constant expressions, i.e. compile time evaluable expressions. The number of positional parameters is determined by the definition of custom attribute's class public constructor(s).

If the custom attribute class is a CLR class, it can define multiple public constructors with various number and types of parameters. The PHP/CLR compiler statically resolves the constructor that fits the best the number and static types of the positional arguments specified in the attribute usage.

Although a PHP custom attribute class can define at most one public constructor, the PHP language enables the method (and constructor as well) to define some of the arguments optional. Therefore, the PHP custom attribute class also provides the feature of optional positional arguments in attribute usage. If the constructor defines any type-hints, the static types of the attribute arguments must be compatible with the hints. If the attribute class defines no constructor at all the default parameterless constructor is picked.

A custom attribute usage can optionally specify named parameters, which corresponds to the public writable properties/fields of the custom attribute class. The static type of the argument value must be compatible with the type of the property/field for CLR properties and fields.

Example

1. *AppStatic attribute usage.* The example of a multithreaded application illustrates the difference between a static field-backed property with and without *AppStatic* attribute. The *Data::\$shared* property is shared among all threads, as the *AppStatic* attribute is applied, while *Data::\$local* property is local to each thread.

```
<?
// Compilation mode: standard
// References: PhpNetClassLibrary, mscorlib

class Data
```

```

{
    [AppStatic]
    public static $shared = 0;

    [AppStatic]
    public static $mutex = 0;

    public static $local = 0;
}

function Worker($id, $iterations, $sleep)
{
    for ($i = 0; $i < $iterations; $i++)
    {
        System::Threading::Monitor::Enter(Data::$mutex);
        $local = Data::$local++;
        $shared = Data::$shared++;
        System::Threading::Monitor::i'Exit'(Data::$mutex);

        echo "$id/$i local=$local shared=$shared\n";
        usleep($sleep);
    }
}

for ($i = 0; $i < 3; $i++)
    clr_create_thread("Worker", $i, rand(2, 5), rand(100, 100000))->Start();
?>

```

The `clr_create_thread` function of the *PhpNetClassLibrary* creates a new CLR thread using a given routine (specified by PHP callback, i.e. by name „Worker“) and arguments passed to the routine. It returns an instance of CLR class `System::Threading::Thread` that represents the created thread. A new thread should always be created by this function and not by a direct instantiation of the `Thread` class as the PHP run-time context (so called *script context*) has to be adjusted when a new thread is created.

One of the possible outputs of the program could be:

```

0/0 local=0 shared=0
1/0 local=0 shared=1
2/0 local=0 shared=2
1/1 local=1 shared=3
1/2 local=2 shared=4
2/1 local=1 shared=5
0/1 local=1 shared=6
0/2 local=2 shared=7
0/3 local=3 shared=8

```

2. *Custom attribute type definition.* In PHP/CLR, the programmer can also define a custom attribute type. This examples shows how:

```

<?
// Compilation mode: pure or standard
// References: mscorlib

import namespace System;

[AttributeUsage(AttributeTargets::All, $AllowMultiple => true)]
class MyAttribute extends Attribute
{
    private $a, $b;
    public $c;

    function __construct($a, $b = "Default")
    {
        $this->a = $a;
        $this->b = $b;
        $this->c = "Default";
    }
}

```

?>

3. *Custom attribute type usage.* The custom attribute class defined above can be subsequently used as follows:

```
<?
[My("on class C", "optional positional arg", $c => "optional named arg")]
class C
{
    [My("on function")]
    function [My("on return value")] f([My("on parameter")] $arg)
    {
    }
}

[assembly: My("on declaring CLR assembly")]
[module: My("on declaring CLR module"), My("once more on declaring module")]
[My("on class D")]
class D
{
}

class G<: [My("on generic parameter S")]S, T:>
{
    function f<:P, [My("on generic parameter Q")] Q:>()
    {
    }
}
?>
```

3.4. Primitive Types

Syntax

primitive-type:
bool
int
int64
double
string
resource
object
array

Semantics

1. *Long integer.* PHP/CLR introduces 64-bit signed integer type, called *int64*. The signed integer type *int* is of 32-bits in PHP/CLR. Both widths are independent of the platform.
2. *Silent overflows.* If an operation on integer overflows in PHP the type is silently changed to *double*. In PHP/CLR the *integer* overflows to *long integer* and then to *double* (with precision loss). The difference is only in the types and not in the values so the existing applications should be unaffected unless they count on the exact type.
3. *Coercions.* Primitive types are coercible as follows:
 - *bool* is coerced to *integer* (*false* to 0, *true* to 1),
 - *integer* is coerced to *long integer*,
 - *long integer* is coerced to *double*,
 - *bool* is coerced to *string* (*false* to an empty string, *true* to "1"),
 - *integer*, *long integer* and *double* are coerced to *string* (using invariant culture formatting),

- *resource* is coerced to *integer* (id assigned to the resource),
 - *string* is coerced to *integer*, *long integer*, and *double*.
4. *Numeric coercion of the string type*. A string value can be automatically coerced to *integer*, *long integer*, or *double* if the operation it takes part defines. The string is said to be *numeric* if it exactly matches one of the following patterns:
- `[:white:]*[+-]?[0-9]*[.]?[0-9]*([dDeE][+-]?[0-9]+)?` (floating point number)
 - `[:white:]*[+-]?[0-9]*` (decimal integer number)
 - `[:white:]*[+-]?0(x|X)[0-9A-Fa-f]*` (hexadecimal integer number)

and if the corresponding numeric value fits the range of the *integer*, *long integer*, or *double* type. The numeric value of the string is of type *double* if it matches the a) format and doesn't match the b) or matches b) yet the value doesn't fit to the *long integer*. The numeric value of the string is of type *long integer* if it matches format b) or c) and doesn't fit into 32-bit *integer*. Otherwise, the numeric value of string is of type *integer*.

5. *CLR primitive types*. A PHP variable (local variable, global variable, PHP fields, keys and values of a PHP array) can only store PHP/CLR types. Any value of a CLR primitive type is converted to the corresponding PHP/CLR primitive type if it is to be stored in PHP variable. Non-primitive types are wrapped into a designated PHP/CLR type. A value of any CLR integer primitive type is coerced either to the *integer* or to the *long integer* type, which fits the best. If even *long integer* is too small, the value is converted to the *double* type with a possible loss of information.
6. *Array*. The PHP *array* type is a hash table mapping *integer* and/or *string* keys to an arbitrary PHP/CLR type. Entries with keys of both types can be intermixed. The order of entries addition is preserved and the array used for iteration.
7. *Object*. When *CLR Semantics* is enabled, the *object* primitive type is a superclass of any PHP/CLR type including wrapped CLR types and all PHP/CLR primitive types (which can be treated as objects having fields and methods). Under *CLR Semantics*, the *object* primitive type is equivalent to the *System.Object* CLR type). The result of the `$x instanceof object` expression is always *true* in this setting, regardless of the value of `$x`.

When CLR Semantics is disabled (the default, when compiling is standard mode, for backward compatibility), the *object* primitive type is a superclass of any PHP class that doesn't (indirectly) derive from a CLR class. There are two inheritance hierarchies, each having its root. The first is the hierarchy of PHP classes not derived from a CLR class with the root *object* (which itself cannot be instantiated). The second is the hierarchy of CLR classes and PHP classes deriving from a CLR class with the root *System.Object*. In this setting, a primitive type cannot be treated as object.

8. *Methods of primitive types*. Under CLR Semantics, each primitive type can be used as an object with methods as follows:
- *bool*
 - *integer*
 - *long integer*
 - *double*
 - *string*
 - *resource*
 - *object*
 - *array*

3.5. Generics

Syntax

static-type-reference:
primitive-type
qualified-static-type-reference

qualified-static-type-reference:
qualified-name static-type-arguments_{opt}

static-type-arguments:
<: static-type-reference-list :>

static-type-reference-list:
static-type-reference-list , static-type-reference
static-type-reference

qualified-static-type-reference-list:
qualified-static-type-reference-list , qualified-static-type-reference
qualified-static-type-reference-list

dynamic-type-reference:
qualified-static-type-reference
dynamic-name dynamic-type-arguments_{opt}

dynamic-type-arguments:
<: dynamic-type-reference-list :>

dynamic-type-reference-list:
dynamic-type-reference-list , dynamic-type-reference
dynamic-type-reference

type-parameters:
<: type-parameter-list :>
<: optional-type-parameter-list :>
<: type-parameter-list , optional-type-parameter-list :>

type-parameter:
custom-attributes_{opt} identifier

optional-type-parameter:
custom-attributes_{opt} identifier = static-type-reference

type-parameter-list:
type-parameter-list , type-parameter
type-parameter-list

optional-type-parameter-list:
optional-type-parameter-list , optional-type-parameter
optional-type-parameter-list

Semantics

1. *Generics.* As of version 2.0, CLR types and methods can define generic type parameters. The PHP/CLR language, as a first class citizen, enables to consume generic types and methods as well as to produce them to a large extent (in some aspects, even to the larger than the C# version 2.0). Since the PHP language is loosely typed, the generics do not bring something revolutionary to the PHP code itself, however, the notion of generic parameters and arguments is essential for seamless

interoperability with CLR. The type arguments need to be specified, for example, when extending a generic CLR class, implementing a generic CLR interface, calling a CLR generic method etc.

2. *Generic declarations.* Classes, interfaces, methods, and functions can be declared with generic type parameters (see type declaration and function declaration).
3. *Type parameters.* Type parameter is a type name visible within the class, interface, method or function declaration. Type parameters are not visible in the declarations nested into the generic class/method/function declaration. The name of the type parameter cannot be the same as the name of the declaring generic type; however, it can be the same as the name of the declaring method. Moreover, it is possible (yet not recommended) for the type parameter of the method to be of the same name as the type parameter of the declaring class/interface. In such case, the method's one hides the class' one within the scope of the method declaration starting just *after* the declaration of the type parameters.

Type parameters can be declared as mandatory or optional. The optional declaration follows the mandatory ones and requires a default type to be specified. This type *cannot* be the type parameter of the generic type/method/function whose type parameters are currently being defined. The scope of visibility of the type parameters starts just after the declaration of *all* the type parameters (i.e. behind the closing ':>')¹. When referring to a generic type/function/method, the default type of the optional type parameter is used if the corresponding argument is missing. Arguments corresponding to the mandatory parameters must be specified.

Generic CLR types and methods are treated as if all of their type parameters were declared optional all with default type *System.Object*, unless constraints prevent such substitution. If some constraint on a type parameter rules out the substitution, the parameter itself and all proceeding ones are treated as mandatory.

PHP classes and interfaces as well as functions and methods cannot be overloaded over the number of type arguments. Constraints on generic type parameters are not supported in declarations of PHP classes and interfaces. They are however checked when instantiating CLR types.

PHP classes and interfaces should be emitted as generic CLR types with the same number of generic type parameters as declared in (mandatory + optional) in order to allow inheriting from generic CLR types. To comply with CLR convention, the name of the generic type should be suffixed by the back-quote followed by the string representation of the number of generic arguments. On the other hand, the generic PHP methods and functions needn't to be emitted as generic CLR methods. The type arguments may be passed by regular hidden parameters.

4. *Type arguments.* Type arguments are specified using either *static-type-arguments* or *dynamic-type-arguments*. The former can only list type names, while the later can use variables for specifying the type arguments. Static type references are used in class/interface declaration for specifying base class and interfaces, in function/method declaration for specifying type hints, and in static field access and static method invocation for specifying the class. Dynamic type references can be used in *instanceof* and *new* operators and generic methods and functions invocations. Type arguments can be omitted if the corresponding type parameters are optional. If all parameters are optional, the type/method can be referred to without specifying any type arguments at all.
5. *Dynamically specified constructed types.* Possibility to specify variables in constructed types using *dynamic-type-reference* allows some extent of dynamicity. To allow fully dynamic use of a constructed type, PHP/CLR enables to encode the constructed type into a string. If the *dynamic-name* used without type arguments yields to a string value with a correct format, the value is

¹ This rule prevents from type parameters to refer to each other, which would cause complications for the compiler and the run-time. Besides, it would be necessary to check for cycles.

decoded to a constructed type. The format of the string, lets denote it *{constructed-type}* is following:

{qualified-type-name}[<[:]?*{type-argument}*](,*{type-argument}*)*[:]?[>]

The *{qualified-type-name}* is a (qualified) name of a generic type and *{type-argument}* is

- a. *{qualified-type-name}*
- b. *{constructed-type}*
- c. [!]*{type-parameter-name}*
- d. [@]*{primitive-type-name}*

The *{type-parameter-name}* following an exclamation mark can refer to a type parameter of the current generic type or generic method/function.

The *{primitive-type-name}* refers to a primitive type name as defined in Section 3.4.

Example

1. *Declaring a generic type.*

```
<?
// Compilation mode: pure or standard
// References: mscorlib

import interface System::Collections::Generic::IComparer;

// Generic class with two generic parameters:
class C1<:T, S:>
{
    public static $x = 0;
}

// Generic class with one generic parameter:
class C2<:T:> extends C1<:T, bool:>
{
}

// Non-generic class:
class C3 extends C1<:string, double:>
{
}

// Generic class implementing BCL generic interface.
// The generic parameter has a default value,
// which means that in PHP the generic parameter can be omitted.
class MyGenericComparer<:T = array:> implements IComparer<:T:>
{
    function Compare($x, $y)
    {
        if ($x < $y) return -1;
        if ($x > $y) return 1;
        return 0;
    }
}
?>
```

2. *Creating an instance of a generic type.* Using above declarations, we can create instances of the classes *C1*, *C2*, *C3* and *MyGenericComparer* with various type arguments.

```
<?
// Compilation mode: pure or standard
// References: PhpNetClassLibrary, mscorlib

function Main()
{
    $c1 = new C1<:int, string:>;
    $c2 = new C2<:C1<:C3, C3:>>;
}
```

```

$c3 = new C3;

// generic parameter T is optional so we needn't to specify any type argument:
$cmp = new MyGenericComparer;

$t1 = "C1";
$t2 = "C2";
$t3 = "C3";
$i = "@int";
$s = "@string";

// pass type arguments dynamically:
$dc1 = new C1<: $i, $s :>;
$dc2 = new C2<: C1 <: $t3, $t3 :> :>;
$dc3 = new $t3;

// state both the generic types and type arguments dynamically:
$ddc1 = new $t1<: $i, $s :>;
$ddc2 = new $t2<: $t1 <: $t3, $t3 :> :>;

// let the entire constructed names be parsed at run-time:
$c1_name = "C1<:@int,@string:>";
$c2_name = "C2<:C1<:C3,C3:>:>";
$c3_name = "C3";
$c4_name = "MyGenericComparer";

$sc1 = new $c1_name;
$sc2 = new $c2_name;
$sc3 = new $c3_name;
$sc4 = new $c4_name;

// Although the following constructed name is corrent, the type won't be
// resolved as may be expected -- the second parameter's name includes
// a leading space and type "@string" wasn't declared.
$unknown_name = "C1<:@int, @string:>";
$sc_error = new $unknown_name;
}

```

3. *Declaring a generic function/method.*
4. *Invoking a generic method.*
5. *Accessing a static property.* Using above declarations, we can access a static property `$x` if the type `C1`. Note that the static storage is associated with an instantiation of the generic type as illustrated by the following example.

```

<?
// Compilation mode: pure or standard
// References: PhpNetClassLibrary, mscorlib

function Main()
{
    C1<:int, string:>::$x = 1;
    C1<:int, int:>::$x = 2;

    // prints 1, 2:
    var_dump(C1<:int, string:>::$x, C1<:int, int:>::$x);
}

?>

```

3.6. Source Unit

Syntax

```

source-unit:
    empty-statementsopt

```

*import-statement-list*_{opt}
*top-statements*_{opt}

empty-statements:
empty-statements empty-statement

top-statements:
top-statements top-statement
top-statement

top-statement:
global-constant-declaration
declarative-statement
non-declarative-statement permitted only in standard mode
empty-statement

declarative-statement:
type-declaration-statement
function-declaration-statement

Semantics

1. *Source unit*. The source unit corresponds to a file or a piece of code to be evaluated at run-time. Each source unit has its own set of imported namespaces and aliases for classes, interfaces, functions and constants listed in *import-statement-list*. The scope of the imported declarations is exactly the source unit. Note that the imports defined in the source unit are not visible to the code evaluated inside the unit, for example by the *eval* construct. The evaluated code need to use either fully qualified names or to declare the imports by itself.

3.7. Namespaces

Syntax

import-statement-list:
import-statement-list import-statement
import-statement

import-statement:
import declaration-kind qualified-name :: identifier ;
import declaration-kind qualified-name :: identifier as identifier ;
import namespace qualified-name ;

declaration-kind:
class
interface
function
const

namespace-declaration-statement:
namespace *qualified-name*_{opt} { *namespace-statement-list*_{opt} }

namespace-statement-list:
namespace-statement
namespace-statement-list namespace-statement

namespace-statement:
global-constant-declaration
declarative-statement

Semantics

1. *Import.* The *import-statement* defines either an alias name for the fully qualified name of a class, interface, function or constant visible throughout the declaring source unit, or imports all declarations from a specified namespace. Neither the declarations nor the namespace need to exist at the time of importing. The *import-statement* only establishes a name mapping that is applied when a type/function/constant is being resolved by the compiler or at run-time.
2. *Name resolving.* Type/function/constant name resolving uses the mapping established by *import-statements* of the source unit referring to the type/function/constant. First, the name is resolved as is. If not found, the name is mapped using the aliases defined by *import-statements*. Only the aliases of the kind of the resolved name are considered (e.g. when resolving a type name, only class and interface aliases are considered). If no alias matches the name being resolved, the resolving continues by prefixing the name with imported namespaces, one by another. If the name cannot be resolved neither this way, it is considered unknown.

In compile-time, references to unknown types/functions/constants are allowed as the entity may be created at run-time. The same resolving process takes place at run-time for the compile-time unknown types, now considering run-time activated or created types as well. Besides, the feature of user defined class/interface loading is also available as defined by PHP language (see “magic” function `__autoload`). If the type or function is still not resolved a run-time error occurs.

3. *Namespace declaration.* Namespaces can be declared only as top-level statements, i.e. cannot be declared conditionally. The namespace declaration comprises of an optional name and a list of class, interface, function and global constant declarations. If the name is not specified, the namespace is said to be *anonymous*.
4. A class/function/constant declared in a namespace can be referred to only via a fully qualified name or using an import. The declarations are not accessible by their unqualified names even within the namespace itself. The declaration in an anonymous namespace is imported implicitly with an alias equal to its unqualified name (it would be impossible to access the declaration otherwise). As it is not possible to import declarations of an anonymous namespace from other source unit, the anonymous namespace effectively makes the declarations visible only within the declaring source unit.
5. *Namespace-private declarations.*

Examples

```
<?
// Compilation mode: pure
// References: mscorlib, System, System.Drawing, System.Windows.Forms

import namespace System;
import namespace System::Windows::Forms;

import class WinApp::MainForm as MyForm;

namespace WinApp
{
    class MainForm extends Form
    {
        private function InitializeComponent()
        {
            $this->tooltip = new Tooltip;
            $this->tooltip->ToolTipIcon = ToolTipIcon::Info;
            $this->tooltip->ToolTipTitle = "Click the image to zoom in and out";

            // ... //
        }
    }
}

function Main()
{
    Application::Run(new MyForm);
}
```

?>

3.8. Type Declaration

Syntax

type-declaration-statement:

class-attributes class-declarator

interface-attributes interface-declarator

class-attributes:

custom-attributes_{opt} declaration-visibility_{opt} partial_{opt} class-modifier_{opt}

class-declarator:

class identifier type-parameters_{opt} extends_{opt} implements_{opt} { member-declaration-list }

interface-attributes:

custom-attributes_{opt} declaration-visibility_{opt} partial_{opt}

interface-declarator:

interface identifier type-parameters_{opt} extends-multiple_{opt} { member-declaration-list }

declaration-visibility:

private

class-modifier:

abstract

final

partial:

partial

extends:

extends qualified-static-type-reference

extends-multiple:

extends qualified-static-type-reference-list

implements:

implements qualified-static-type-reference-list

Semantics

1. *Partial modifier.* A new *partial* modifier is added to grammar productions of class and interface declarations. When this modifier is applied on multiple type declarations of the same name, all of them constitute a single type (class or interface). The resulting type declaration is an aggregate of all the declarations within the compilation unit having the same name. The concept of partial classes applies only at compile-time, there is no run-time type merging. Therefore, the partial modifier is available only in the pure compilation mode, a partial declaration must be unconditional, all partial declarations of a class must extend the same base class (if any), all must have the same *class-modifiers* and the same generic parameter declarations.

3.9. Function Declaration

Syntax

function-declaration-statement:

function-attributes function-declarator

function-attributes:

custom-attributes_{opt} declaration-visibility_{opt}

function-declarator:
function *return-value* *function-name* *type-parameters*_{opt} (*formal-parameters*_{opt}) *block*

function-name:
identifier
__autoload

return-value:
*custom-attributes*_{opt} *reference*_{opt}

reference:
&

formal-parameters:
formal-parameter-list
optional-formal-parameter-list
formal-parameter-list , *optional-formal-parameter-list*

formal-parameter:
*custom-attribute*_{opt} *type-hint*_{opt} *reference*_{opt} *variable*

optional-formal-parameter:
*custom-attribute*_{opt} *type-hint*_{opt} *reference*_{opt} *variable* = *constant-initializer*

type-hint:
static-type-reference

formal-parameter-list:
formal-parameter-list *formal-parameter*
formal-parameter

optional-formal-parameter-list:
optional-formal-parameter-list *optional-formal-parameter*
optional-formal-parameter

Semantics

1. *Custom attributes.*
2. *Type hints.*
3. *Optional parameters* (“overloading” by number of parameters)

3.10. Global Constant Declaration

Syntax

global-constant-declaration:
*custom-attributes*_{opt} const *global-constant-declarator-list* ;

global-constant-declarator:
identifier = *constant-initializer*

global-constant-declarator-list:
global-constant-declarator-list *global-constant-declarator*
global-constant-declarator

Semantics

1. *Global constants*. Global constants are similar to class constants; the major difference is that they are declared in a global scope and can be declared inside a namespace.

Example

3.11. Type Member Declaration

Syntax

type-member-declaration:

```
custom-attributesopt member-modifiersopt const class-constant-declarator-list ;  
custom-attributesopt property-modifiersopt property-declarator-list ;  
custom-attributesopt member-modifiersopt method-declarator ;  
custom-attributesopt member-modifiersopt constructor-declarator ;
```

method-declarator:

```
function return-value method-name ( formal-parametersopt ) ;  
function return-value method-name ( formal-parametersopt ) block
```

constructor-declarator:

```
function return-value __construct ( formal-parametersopt ) parent-ctor-callopt block
```

parent-ctor-call:

```
: parent ( actual-argument-listopt )
```

method-name:

```
identifier  
__destruct  
__get  
__set  
__call  
__sleep  
__wakeup  
__toString
```

property-modifiers:

```
var  
member-modifiers
```

member-modifiers:

```
member-modifiers member-modifier  
member-modifier
```

member-modifier:

```
private  
protected  
public  
static  
abstract  
final
```

Semantics

3.12. Class Constants

Syntax

class-constant-declarator-list:
class-constant-declarator-list class-constant-declarator
class-constant-declarator

class-constant-declarator:
identifier = constant-initializer

Semantics

1. Same as in the PHP language version 5.

3.13. Property Accessors

Syntax

property-declarator-list:
property-declarator-list , property-declarator
property-declarator

property-declarator:
variable constant-initializer_{opt}
variable { property-getter_{opt} property-setter_{opt} }
variable { property-setter_{opt} property-getter_{opt} }

property-getter:
custom-attributes_{opt} method-modifiers_{opt} __get block

property-setter:
custom-attributes_{opt} method-modifiers_{opt} __set block

Semantics

1. *Properties and fields.* CLR distinguishes fields and properties – a field provide storage for a value associated with an object (instance field) or with a class (static field) while a property is a pair of methods (getter and setter, one may be omitted), which are often used merely to read/write the value of the field from outside the class. Such properties are called *field-backed* properties. PHP/CLR uses the same syntax for accessing PHP class variables, CLR fields and CLR properties (i.e. *\$object->\$property_or_field*, or *Class::\$property_or_field*).

3.14. Statements

Syntax

statement:
non-declarative-statement
declarative-statement

non-declarative-statement:
empty-statement
label
block
echo-statement
inline-html-statement
expression-statement
unset-statement
if-statement
switch-statement
loop-statement
branch-statement

try-catch-statement
throw-statement
variable-declaration-statement

empty-statement:
 ;

label:
identifier ;

block:
 { *statement-list*_{opt} }

echo-statement:
 echo *expression-list* ;

inline-html-statement:
inline-html

expression-statement:
expression ;

unset-statement:
 unset (*writable-chain-list*) ;

if-statement:
 if (*expression*) *non-declarative-statement* *else-if-list*_{opt} *else*_{opt}
 if (*expression*) : *statement-list*_{opt} *else-if-colon-list*_{opt} *else-colon*_{opt} endif ;

else-if-list:
else-if-list elseif (*expression*) *non-declarative-statement*

else-if-colon-list:
else-if-colon-list elseif (*expression*) : *statement-list*_{opt}

switch-statement:
 switch (*expression*) *switch-block*

switch-block:
 { *switch-cases*_{opt} }
 { ; *switch-cases*_{opt} }
 : *switch-cases*_{opt} endswitch ;
 : ; *switch-cases*_{opt} endswitch ;

switch-cases:
switch-cases *switch-case*
switch-case

switch-case:
 case *expression* : *statement-list*_{opt}
 case *expression* ; *statement-list*_{opt}
 default *expression* : *statement-list*_{opt}
 default *expression* ; *statement-list*_{opt}

loop-statement:
while-statement
do-while-statement
for-statement

foreach-statement

while-statement:
while (*expression*) *non-declarative-statement*
while (*expression*) : *statement-list*_{opt} endwhile ;

do-while-statement:
do *non-declarative-statement* while (*expression*) ;

for-statement:
for (*expression-list*_{opt} ; *expression-list*_{opt} ; *expression-list*_{opt}) *non-declarative-statement*
for (*expression-list*_{opt} ; *expression-list*_{opt} ; *expression-list*_{opt}) : *statement-list*_{opt} endfor ;

foreach-statement:
foreach (*writable-chain* as *reference*_{opt} *foreach-entry*) *non-declarative-statement*
foreach (*chain-free-expression* as *foreach-entry*) *non-declarative-statement*
foreach (*writable-chain* as *reference*_{opt} *foreach-entry*) : *statement-list*_{opt} endforeach ;
foreach (*chain-free-expression* as *foreach-entry*) : *statement-list*_{opt} endforeach ;

foreach-entry:
writable-chain
writable-chain => *reference*_{opt} *writable-chain*

branch-statement:
break *expression*_{opt} ;
continue *expression*_{opt} ;
return *expression*_{opt} ;
goto *identifier* ;

throw-statement:
throw *expression* ;

try-catch-statement:
try *block* *catch-blocks*

catch-blocks:
catch-blocks *catch-block*
catch-block

catch-block:
catch (*qualified-static-type-reference* *variable*) *block*

variable-declaration-statement:
global *global-variable-list* ;
static *static-variable-list* ;

global-variable:
variable
\$ *chain*
\$ { *expression* }

static-variable:
variable
variable = *constant-initializer*

global-variable-list:
global-variable-list , *global-variable*
global-variable

static-variable-list:
static-variable-list , *static-variable*
static-variable

statement-list:
statement-list *statement*
statement

Semantics

1. Only the statements specific to the PHP/CLR are explained further. The others are have the same syntax and semantics as in the PHP language version 5.
2. *Labels*.

3.15. Expressions

Syntax

expression-list:
expression-list , *expression*
expression

expression:
chain
expression-without-chain

expression-without-chain:
parenthesized-expression
assignment-expression
new-expression
clone-expression
instance-of-expression
eval-expression
assert-expression
exit-expression
isset-expression
empty-expression
array-match-expression
array-construction-expression
inclusion-expression
increment-expression
decrement-expression
unary-expression
binary-expression
conditional-expression
shell-command-expression
scalar-expression
linq-expression

parenthesized-expression:
(*expression*)

assignment-expression:
writable-chain *assignment-operator* *expression*
writable-chain = *reference*_{opt} *new* *dynamic-type-reference* (*actual-argument-list*_{opt})

assignment-operator: one of
= =& += -= *= /= .= %= &= |= ^= >>= <<=

new-expression:
new *dynamic-type-reference* (*actual-argument-list*_{opt})

clone-expression:
clone *expression*

instance-of-expression:
expression instanceof *dynamic-type-reference*

eval-expression:
eval (*expression*)

assert-expression:
assert (*expression*)

exit-expression:
exit
exit ()
exit (*expression*)

empty-expression:
empty (*chain*)

isset-expression:
isset (*writable-chain-list*)

array-match-expression:
list (*array-match-item-list*)

array-match-item-list:
array-match-item-list , *array-match-item*_{opt}
*array-match-item*_{opt}

array-match-item:
chain
array-match-expression

array-construction-expression:
array (*array-item-list*_{opt} *comma*_{opt})

constant-array-construction-expression:
array (*constant-array-item-list*_{opt} *comma*_{opt})

array-item-list:
array-item-list *array-item*
array-item

array-item:
expression
& *writable-chain*
expression => *expression*
expression => & *writable-chain*

constant-array-item-list:
constant-array-item-list *constant-array-item*
constant-array-item

constant-array-item:
constant-initializer
constant-initializer => *constant-initializer*

inclusion-expression:
include *expression*
include_once *expression*
require *expression*
require_once *expression*

increment-expression:
writable-chain ++
++ writable-chain

decrement-expression:
writable-chain --
-- writable-chain

unary-expression:
unary-operator *expression*
type-cast-operator *expression*

unary-operator: one of
+ - * /

type-cast-operator: one of:
(bool) (double) (float) (string) (unicode) (array) (object) (unset)
(int8) (int16) (int32) (int64) (uint8) (uint16) (uint32) (uint64) (decimal)

binary-expression:
expression *binary-operator* *expression*

binary-operator: one of
| & ^ || && or and xor . + - * / % << >> == === != !== < > <= >=

conditional-expression:
expression ? *expression* : *expression*

shell-command-expression:
` *composite-string*_{opt} `

invocation-expression:
callee *dynamic-type-arguments*_{opt} (*actual-argument-list*_{opt})

callee:
qualified-name
dynamic-name
qualified-static-type-reference :: *dynamic-name*

actual-argument-list:
actual-argument-list *actual-argument*
actual-argument

actual-argument:
expression

constant-initializer:
constant
constant-array-construction-expression

+ *constant-initializer*
- *constant-initializer*

constant:
literal-constant
pseudo-constant
global-constant
class-constant

literal-constant:
integer-literal
long-integer-literal
double-literal
single-quoted-string-literal
double-quoted-string-literal

pseudo-constant:
__LINE__
__FILE__
__NAMESPACE__
__CLASS__
__METHOD__
__FUNCTION__

class-constant:
qualified-static-type-reference :: *identifier*

global-constant:
qualified-name

scalar-expression:
constant
string-embedded-variable-name
composite-binary-string
composite-double-quoted-string
composite-binary-heredoc
composite-heredoc

composite-binary-string:
b" *composite-string*_{opt} "

composite-double-quoted-string:
" *composite-string*_{opt} "

composite-binary-heredoc:
b<<<*LABEL* *composite-string*_{opt} *LABEL*

composite-heredoc:
<<<*LABEL* *composite-string*_{opt} *LABEL*

composite-string:
composite-string *string-embedded-variable*
composite-string *string-embedded-literal*
string-embedded-variable
string-embedded-literal

string-embedded-literal:
string-embedded-simple-identifier
string-embedded-integer
string-embedded-characters

```
->
[
]
{
}
```

string-embedded-variable:

```
variable
variable [ string-embedded-key ]
variable -> string-embedded-simple-identifier
${ expression }
${ string-embedded-variable-name [ expression ] }
{ chain }
```

string-embedded-key:

```
string-embedded-simple-identifier
string-embedded-integer
variable
```

Semantics

1. Only the expression specific to or modified in the PHP/CLR language are explained further. The others have the same syntax and semantics as in the PHP language version 5.
2. *Type arguments.*
3. *Type casts.*
4. *Assertion.* The *assert-expression* is treated as ordinary function call in PHP. PHP/CLR explicitly treats the assertions in a special way so that they can be evaluated more effectively and wiped off when compiled in release mode.
5. *Passing an argument by reference.*

3.16. Chains

Syntax

writable-chain:
chain

writable-chain-list:
writable-chain-list , *writable-chain*
writable-chain

chain:
chain-with-calls

dynamic-name:
chain-without-calls

chain-without-calls:
chain-base *member-chain*_{opt}

chain-with-calls:
chain-base-with-calls *member-chain-with-calls*_{opt}

member-chain:
member-chain member-chain-link
member-chain-link

member-chain-with-calls:
member-chain-with-calls member-chain-link-with-call
member-chain-link-with-call

member-chain-link:
 -> *keyed-member-name*

member-chain-link-with-call:
 -> *keyed-member-name*
 -> *keyed-member-name (actual-argument-list_{opt})*

chain-base:
qualified-static-type-reference :: keyed-variable
keyed-variable

chain-base-with-calls:
chain-base
invocation-expression

keyed-member-name:
keyed-simple-member-name
keyed-variable

keyed-simple-member-name:
keyed-simple-member-name [key_{opt}]
keyed-simple-member-name { expression }
identifier
{ expression }

keyed-variable:
 \$ *keyed-variable*
keyed-compound-variable

keyed-compound-variable:
keyed-compound-variable [key_{opt}]
keyed-compound-variable { expression }
compound-variable

compound-variable:
variable
 \$ { *expression* }

Semantics

1. Semantics of the chains is the same as in PHP language version 5.

3.17. Language Integrated Query (LINQ)

Syntax

linq-expression:
linq-from-clause linq-query-body

linq-from-clause:
from linq-generator-list

linq-where-clause:
where expression

linq-generator-list:
linq-generator-list , linq-generator
linq-generator

linq-generator:
expression as variable
expression as variable => variable

linq-query-body:
linq-from-where-clause-list_{opt}
linq-order-by-clause_{opt}
linq-select-group-by-clause
linq-into-clause_{opt}

linq-from-where-clause-list:
linq-from-where-clause-list linq-from-clause
linq-from-where-clause-list linq-where-clause
linq-from-clause
linq-where-clause

linq-order-by-clause:
orderby linq-ordering-list

linq-ordering-list:
linq-ordering-list , linq-ordering-clause
linq-ordering-clause

linq-ordering-clause:
expression
expression descending
expression ascending

linq-select-group-by-clause:
select expression
group expression by expression

linq-into-clause:
as variable in linq-query-body
as variable => variable in linq-query-body

Semantics

1. *LINQ*.

4. References

- [1] DotNetLanguages.com, <http://www.dotnetlanguages.com>
- [2] Microsoft: *.NET Framework*, MSDN, <http://msdn.microsoft.com/netframework>
- [3] Microsoft: *Language Integrated Query*, MSDN, <http://msdn.microsoft.com/data/ref/linq>
- [4] PHP Group: *PHP language manual*, <http://www.php.net/manual>