**CopilotBuilder Yarr Language Reference**

**Ray Comas**

**CopilotBuilder**

[Yarr language reference 5](#_Toc197507950)

[Data types 5](#_Toc197507951)

[Comments 6](#_Toc197507952)

[Symbols 6](#_Toc197507953)

[Expressions 7](#_Toc197507954)

[Exception handling 8](#_Toc197507955)

[Special forms 8](#_Toc197507956)

[And 8](#_Toc197507957)

[Backquote 8](#_Toc197507958)

[Bind-setf-method 9](#_Toc197507959)

[Call 9](#_Toc197507960)

[Case 9](#_Toc197507961)

[Cond 10](#_Toc197507962)

[Decf 10](#_Toc197507963)

[Declare 10](#_Toc197507964)

[Defmethod 11](#_Toc197507965)

[Defparameter 11](#_Toc197507966)

[Defstruct 11](#_Toc197507967)

[Defvar 11](#_Toc197507968)

[Do 12](#_Toc197507969)

[Each 12](#_Toc197507970)

[Fn 12](#_Toc197507971)

[For 14](#_Toc197507972)

[Foreach 14](#_Toc197507973)

[Function 14](#_Toc197507974)

[If 14](#_Toc197507975)

[Incf 15](#_Toc197507976)

[Lambda 15](#_Toc197507977)

[Let 15](#_Toc197507978)

[Let\* 16](#_Toc197507979)

[Letv 16](#_Toc197507980)

[Lock 17](#_Toc197507981)

[Lock-if 17](#_Toc197507982)

[Macro 17](#_Toc197507983)

[Nullp 18](#_Toc197507984)

[Or 18](#_Toc197507985)

[Quote 18](#_Toc197507986)

[Return 18](#_Toc197507987)

[= / Setq / Setf 18](#_Toc197507988)

[Symbol-name 19](#_Toc197507989)

[Symbol-value 19](#_Toc197507990)

[To 19](#_Toc197507991)

[Try 19](#_Toc197507992)

[Unbind 20](#_Toc197507993)

[Unbind-setf 20](#_Toc197507994)

[While 20](#_Toc197507995)

[With 20](#_Toc197507996)

[Functions 21](#_Toc197507997)

[Append 21](#_Toc197507998)

[Apply 21](#_Toc197507999)

[Aref 21](#_Toc197508000)

[Arithmetic Functions + - \* / % 21](#_Toc197508001)

[Array 22](#_Toc197508002)

[Assoc 22](#_Toc197508003)

[Atomp 22](#_Toc197508004)

[Bitwise Functions & | ^ ~ 22](#_Toc197508005)

[Car 23](#_Toc197508006)

[Cdr 23](#_Toc197508007)

[C*xxx*r 23](#_Toc197508008)

[Comparison Functions < <= >= > 23](#_Toc197508009)

[Cons 24](#_Toc197508010)

[Copylist 24](#_Toc197508011)

[Elt 24](#_Toc197508012)

[Eq 24](#_Toc197508013)

[Eql / == 24](#_Toc197508014)

[Eval 24](#_Toc197508015)

[Evalstring 25](#_Toc197508016)

[Evenp 25](#_Toc197508017)

[Every 25](#_Toc197508018)

[Exit 25](#_Toc197508019)

[First 25](#_Toc197508020)

[Get 25](#_Toc197508021)

[Gensym 25](#_Toc197508022)

[Identity 26](#_Toc197508023)

[In-package 26](#_Toc197508024)

[Is 26](#_Toc197508025)

[Length 26](#_Toc197508026)

[List 26](#_Toc197508027)

[Listp 26](#_Toc197508028)

[Load 26](#_Toc197508029)

[Macroexpand, Macroexpand-1 27](#_Toc197508030)

[Make-package 27](#_Toc197508031)

[Map 27](#_Toc197508032)

[Mapc 27](#_Toc197508033)

[Mapcar 27](#_Toc197508034)

[Mapl 28](#_Toc197508035)

[Maplist 28](#_Toc197508036)

[Max 28](#_Toc197508037)

[Md-array 28](#_Toc197508038)

[Member 28](#_Toc197508039)

[Min 28](#_Toc197508040)

[Minusp 28](#_Toc197508041)

[Nconc 29](#_Toc197508042)

[New 29](#_Toc197508043)

[Not 29](#_Toc197508044)

[!= 29](#_Toc197508045)

[Notany 29](#_Toc197508046)

[Notevery 29](#_Toc197508047)

[Nreverse 30](#_Toc197508048)

[Nth 30](#_Toc197508049)

[Oddp 30](#_Toc197508050)

[Plusp 30](#_Toc197508051)

[Pr 30](#_Toc197508052)

[Prl 30](#_Toc197508053)

[Read 30](#_Toc197508054)

[Reduce 31](#_Toc197508055)

[Reference 31](#_Toc197508056)

[Reset 31](#_Toc197508057)

[Rest 32](#_Toc197508058)

[Reverse 32](#_Toc197508059)

[Second 32](#_Toc197508060)

[Shadow 32](#_Toc197508061)

[Some 32](#_Toc197508062)

[The 32](#_Toc197508063)

[Third 33](#_Toc197508064)

[Throw 33](#_Toc197508065)

[Type 33](#_Toc197508066)

[Typeof 34](#_Toc197508067)

[Use-package 34](#_Toc197508068)

[Using 34](#_Toc197508069)

[Zerop 34](#_Toc197508070)

[Macros 35](#_Toc197508071)

[-> and ->> (thread first/thread last) 35](#_Toc197508072)

[Bind-disptch-character 35](#_Toc197508073)

[Bind-macro-character 35](#_Toc197508074)

[Defun 36](#_Toc197508075)

[Defmacro 36](#_Toc197508076)

[Defsetf 36](#_Toc197508077)

[Make-array 36](#_Toc197508078)

[Mapcan 36](#_Toc197508079)

[Mapcon 37](#_Toc197508080)

[Unbind-dispatch-character 37](#_Toc197508081)

[Unbind-macro-character 37](#_Toc197508082)

[When 37](#_Toc197508083)

[System symbols 38](#_Toc197508084)

[\*environment\* 38](#_Toc197508085)

[\*last-exception\* 38](#_Toc197508086)

[\*print-circle\* 38](#_Toc197508087)

[\*print-length\* 38](#_Toc197508088)

[\*print-level\* 38](#_Toc197508089)

[\*read-eval\* 38](#_Toc197508090)

# Yarr language reference

## Data types

Yarr supports all .NET data types. New types cannot be created in Yarr, but any types and classes defined in .NET assemblies can be used in Yarr by referencing the assembly.

When processing numbers, the Yarr reader automatically detects the type of the number and tries to parse it in the following order:

1. If the number starts with “0x” it is assumed to be hexadecimal, otherwise it is assumed to be decimal.
2. If the number ends in “b” attempts to parse the number as a **System.Byte**
3. If the number ends in “s” attempts to parse the number as a **System.Int16**
4. If the number ends in “us” attempts to parse the number as a **System.UInt16**
5. If the number ends in “u” attempts to parse the number as a **System.UInt32**
6. If the number ends in “l” attempts to parse the number as a **System.Int64**
7. If the number ends in “ul” attempts to parse the number as a **System.UInt64**
8. If the number ends with “m” attempts to parse the number as a **System.Decimal**
9. If the number ends with “z” attempts to parse the number as a **System.Numerics.BigInteger**
10. If the number ends with “f” attempts to parse the number as a **System.Single**
11. Attempts to parse the number as a **System.Int32**. If this fails …
12. Attempts to parse the number as a **System.Int64**. If this fails …
13. Attempts to parse the number as a **System.Numerics.BigInteger**
14. Attempts to parse the number as a **System.Double**.
15. If all the above fail, then the “number” is considered to be a symbol

Numbers of type **System.Numerics.Complex** are entered using the LISP complex number notation:

#C(*real-part imaginary-part*)

Strings are enclosed in double quotes:

"this is a string"

Individual characters are entered using the syntax #\*char*:

(typeof #\c) 🡪 System.Char

Lists are represented internally as instances of the **Yarr.Cons** class. The reader parses sequences enclosed in parentheses and constructs the necessary **Yarr.Cons** object for them. The reader also supports LISP’s “dotted pair” notation:

(car '(a . b)) 🡪 a

(cdr '(a . b)) 🡪 b

(cdr '(a b)) 🡪 (b)

The object to the left of the dot is the **car** of the cons node, and the object to the right of the dot is the **cdr** of the cons node.

The reader can directly parse arrays. Arrays are delimited with square brackets. A list prefixed with a pound-sign (#) is considered to be an array, and is parsed accordingly. Optionally, a rank can be specified for an array by prefixing the list with “#*nn*a” or “#*nn*A”:

(typeof [1 2 3 4]) 🡪 System.Int32[]

(typeof #(1 2 3 4)) 🡪 System.Int32[]

(typeof [[[1 2] [3 4]]) 🡪 System.Int32[][]

(typeof #2a((1 2) (3 4))) 🡪 System.Int32[,]

Yarr supports **System.Boolean** directly. In the context of conditional operators, **false**, **null**, any numeric zero value, **DateTime.MinValue**, and empty strings are all considered *false*. Everything else is considered *true*.

Types self-evaluate, and are first-class objects, so they can be bound to symbols and passed as parameters:

(= a-type System.Int32)

(typeof a-type) 🡪 #<System.RuntimeType>

a-type 🡪 #<System.Int32>

## Comments

The semi-colon (;) is a comment character, and marks everything in the input stream up to the next new-line character as a comment. Multi-line comments are enclosed between #**|** and **|#**:

(= foo 42) ; this is a comment

(= bar 3) #| start of a multi-line comment

(this is ignored) and now the comment ends … |#

## Symbols

Symbols in Yarr can be any sequence of characters. Symbols can be either *bound* or *unbound*. A symbol is bound to an object with the **=/setq/setf** special form. A bound symbol evaluates to the object bound to it. An unbound symbol evaluates to itself, which is an instance of the class **Yarr.Symbol**, except:

* If the symbol name is the same as a type name, and the symbol is unbound, it will evaluate to the type
* If the symbol can be parsed as a number, it will be considered a number, not a symbol

Symbols are grouped into symbol packages, and new packages can be created at any time. A symbol specification of the form **package-name:symbol-name** references a symbol named **symbol-name** in the package named **package-name**. Packages may inherit symbols from other packages.

The **quote** function causes a symbol to evaluate to itself, instead of the object that is bound to it:

(= foo '(a b c))

(typeof foo) 🡪 #<Yarr.Cons>

(typeof (quote foo)) 🡪 #<Yarr.Symbol>

(typeof 'foo) 🡪 #<Yarr.Symbol>

Quoting a number still produces the number. A reader macro is required to create symbols with “numeric” names. Note that the quote character (') is a short-hand for the function **quote**.

## Expressions

As was previously mentioned, numbers, types, and unbound symbols evaluate to themselves. Bound symbols evaluate to the object they are bound to. Lists are evaluated as follows:

1. The **car** of the list is evaluated, and is expected to be a macro, special form, function, closure, or symbol
2. If the **car** of the list is a macro:
   1. The remaining elements of the list are left unevaluated. They are considered arguments, and are bound to the macro’s parameters
   2. The macro is expanded
   3. The result of the macro expansion is evaluated
3. If the **car** of the list is a special form:
   1. The remaining elements of the list are left unevaluated. They are considered arguments, and bound to the special form’s parameters
   2. The special form is invoked
4. If the **car** of the list is a function (either a built-in function, or a Yarr closure):
   1. The remaining elements of the list are evaluated in order, producing a list of values
   2. The list of values is considered to contain arguments for the closure or function, and the values are bound to the parameters in the local environment
   3. The closure or function is invoked
5. If the **car** of the list is a symbol:
   1. The symbol’s name is considered to be the name of a .NET delegate, method, property, or field
   2. If it is a delegate, the delegate is invoked; the remaining elements of the list are evaluated in order, and the resulting values are passed as arguments to the delegate
   3. If it is not a delegate:
      1. If the symbol is of the form “foo.bar” it is transformed internally to “bar foo” so that (foo.bar a b c …) will be evaluated as (bar foo a b c …)
      2. The second element of the list is evaluated. The result of the evaluation is considered to be an instance of a .NET class. If it is a **System.Type**, then the method, property, or field is considered to be a static member of that type
      3. The remaining elements of the list are evaluated in order, producing a list of values. These are considered to be arguments for the .NET method. The data types of these values are used to determine which overloaded method should be called
      4. The .NET method (determined by the first element of the list) is invoked on the .NET object (determined by the second element of the list) and is passed the evaluated values as its parameters. If any of the values is a **Yarr.Symbol**, then the corresponding parameter is assumed to be a **ref** or **out** parameter, and the result will be bound to the specified symbol when the method returns

## Exception handling

Any exception thrown during the evaluation of Yarr expressions is caught by the Yarr runtime’s top-level evaluator. The exception object is bound to the symbol **\*last-exception\*** and, if the exception is not of type **Yarr.YarrException**, it is wrapped in a **Yarr.YarrException**. After being bound (and possibly wrapped), the exception is rethrown. Callers into the Yarr runtime therefore only have to catch exceptions of type **Yarr.YarrException**, and if they require more information, they can obtain the actual exception object from the Yarr environment.

There is also a built-in Yarr **try** special form that can be used to catch and handle exceptions from within Yarr code.

## Special forms

The Yarr runtime contains the following built-in special forms:

### And

(and

*expression1*

*expression2*

…

*expressionN*)

Evaluates each *expression* in turn. If an expression is *false*, no further expressions are evaluated, and the **and** special form evaluates to **false**. If all expressions evaluate to *true*, the **and** special form evaluates to the value of *expressionN*.

### Backquote

(backquote *template*)

`*template*

Performs substitution on *template*. The *template* is examined for symbols prefixed with either an *unquote* tag (,) or a *splice* (,@) tag. Unquoted symbols are replaced by the object bound to them. Spliced symbols are assumed to be bound to lists, and are replaced by the list elements:

(= foo ‘(a b c))

Foo 🡪 (a b c)

`(x ,foo y ,@foo z) 🡪 (x (a b c) y a b c z)

The **backqouote** special form evaluates to the expanded template.

### Bind-setf-method

(bind-setf-method *symbol expression*)

Binds a setf method to a symbol. The first argument is the symbol to which the setf method will be bound. The second argument is an expression, which is evaluated and is expected to produce a closure. This closure is bound to the symbol as a setf method.

The **bind-setf-method** special form evaluates to the closure that was bound.

### Call

(call *method-name* *object* *expression1* *expression2* … *expressionN*)

Calls a .NET method. The first argument is an unbound symbol or string. This specifies the method that is to be called. The second argument is evaluated, and is expected to produce a .NET object instance. If this is an instance of **System.Type**, then the method is considered to be a static method on that type.

The remaining expressions are evaluated in order, producing a list of values. These are considered to be arguments for the .NET method. The data types of these values are used to determine which overloaded method should be called. The .NET method (determined by the first argument) is invoked on the .NET object (determined by the second argument), and is passed the evaluated values as its parameters. If any of the values is a **Yarr.Symbol**, then the corresponding parameter is assumed to be a **ref** or **out** parameter, and the result will be bound to the specified symbol when the method returns.

The **call** special form evaluates to the return value of the .NET method. If it is a **void** method, the **call** special form evaluates to **null**.

### Case

(case *test-expr*

*expr1 result1*

*expr2 result2*

*…*

*exprN resultN*

*[default-expr]*)

First, *test-expr* is evaluated. Then, starting with, *expr1* the first expression in each *expr-result* pair is evaluated.

If the result is a single object, and that object is **eql** to the result of *test-expr*, then corresponding *result* expression is evaluated and returned.

If the result of the *expr* is **IEnumerabe**, and any of the elements of the **IEnumerable** is **eql** to the result of *test-expr*, then the corresponding *result* expression is evaluated and returned.

If there is no match in either case, the *result* expression is not evaluated, and the next *expr* is checked. If none of the *expr* match, then the optional *default-expr* is evaluated and returned. If there is no *default-expr* then null is returned.

### Cond

(cond

*test1 expression1*

*test2 expression2*

*test3 expression3*

*…*

*testN expressionN*

*else-expression*)

Performs conditional execution. Each *test* is evaluated in turn. If it evaluates to *false*, the corresponding *expression* is not evaluated. If it evaluates to *true*, the corresponding expression is evaluated, becomes the result of evaluating the **cond** special form, and no further *tests* or *expressions* are evaluated.

If the final *else-expression* is reached (this expression is optional), it is evaluated and its value becomes the result of evaluating the **cond** special form. If no *else-expression* exists, and all *tests* are *false*, then the **cond** special form evaluates to **null**.

### Decf

(decf *symbol [amount]*)

(-- *symbol*)

Decrements the value bound to *symbol* by *amount*. The *amount* parameter is optional, and defaults to 1. If *symbol* is bound to an object, then **decf** will try to invoke the appropriate **op\_subtraction** method. The (-- *symbol*) special form behaves like post-fix -- in C# or C++: it decrements the value bound to *symbol* but returns the pre-decremented value. If symbol is bound to an object, then -- will attempt to invoke the appropriate **op\_decrement** method.

### Declare

(declare (*symbol1 type1 [value1]*)

(*symbol2 type2 [value2]*)

…

(*symbolN typeN [valueN]*))

Declares each *symbol* in the current environment to be of a specific *type*, and optionally bind them to an initial *value*. If no *value* is specified, the *symbol* is bound to **null** (if *type* is a reference type) or the type’s default value (if *type* is a value type). Evaluates to the final *value* that was set.

### Defmethod

(defmethod *name* [:around | :before | :after] (*parameters*)

*expression1*

*expression2*

*…*

*expressionN*)

Defines a mutlimethod. The *parameters* of the multimethod can be any parameter list usable in the **fn** special form (described below). The multimethod can dispatch on the non-optional, non-key parameters based on either the type listed in the argument list (which defaults to **object** if not provided), or a value if the dispatch specifier is of the form *(eql x)*. See the tutorial in the *CopilotBuilder Yarr Scripting Guide* for more information and examples. For Common LISP users: the Yarr implementation of multimethods is a simplified version of generic methods in the Common LISP Object System.

### Defparameter

(defparameter *symbol [value]*)

Declares *symbol* to be dynamically scoped. If *value* is provided, the symbol is bound to that value, otherwise it is bound to **null**. The **defparameter** special form evaluates to the value to which *symbol* was bound.

### Defstruct

(defstruct *struct-name* (*parent-list*) *slot-def1 slot-def2 …*)

Defines a structure named *struct-name* in the current environment, with slots *slot-def1, slot-def2, …*, and which also inherits the slots of the structures in the *parent-list*. Each *slot-def* can be a *symbol*, or a list of the form (*symbol* *init-value*). The slot will be identified by the symbol *symbol* in the keyword package. If no value is provided for the slot when an instance is created, the slot will have the value *init-value* by default. The slots of the structure will be the union of the *slot-defs* and the slots of all structures in *parent-list*.

### Defvar

(defvar *symbol type* &rest *arguments*)

Declares *symbol* in the current environment to be of the specific *type*. If *arguments* are provided, the **new** function is then invoked, with *type* and *arguments* passed as parameters, and the result of the call to **new** is bound to *symbol*. Otherwise symbol is bound to **null** (if *type* is a reference type) or to the type’s default value (if *type* is a value type). The **defvar** special form evaluates to *symbol*.

### Do

(do

*expression1*

*expression2*

*…*

*expressionN*)

Evaluates each *expression* in turn. The **do** special form evaluates to the value of the last *expression*.

### Each

(each *symbol IEnumeralbe-object*

*expression1*

*expression2*

*…*

*expressionN*)

Iterates over an **IEnumerable** collection. The first argument is expected to be a symbol. The second argument is evaluated, and is expected to produce an instance of an **IEnumerable** class. The special form will create a local environment, and will bind *symbol* in the local environment to each element of the **IEnumerable** class in turn. For each element, it will evaluate the *expressions* one at a time in order, in the local environment.

The **each** special form evaluates to the value of the last *expression*.

### Fn

(fn (*parameters*)

*expression1*

*expression2*

*…*

*expressionN*)

Creates a Yarr closure. The first argument is a parameter list for the closure. When the closure is subsequently invoked, the arguments will be bound to these parameters, and the *expressions* will be evaluated in order. The result of invoking the closure will be the value of the last *expression* evaluated. When the closure is invoked, the symbol **self** will be bound to the closure, allowing the closure to recursively invoke itself.

The **fn** special form evaluates to the closure it creates.

*Parameter Lists*

Parameter lists contain one symbol for each argument that will be passed. They are bound to the arguments in order. The special tags **&optional**, **&key**, and **&rest** result in special handling of the arguments.

The **&optional** tag denotes that all subsequent parameters are optional:

(fn (a &optional b) …)

defines a closure that takes one or two arguments. If a second argument is provided, it is bound to **b** otherwise **b** is **null**. Default values may be specified for the optional parameters:

(fn (a &optional b (c *default-value*)) …)

defines a function that takes one, two, or three arguments. If only one argument is provided, **b** is **null** and **c** is set to *default-value*. If two are provided, they are bound to **a** and **b**, and **c** is set to 5. If three are provided, they are bound to **a**, **b**, and **c**.

The **&key** tag denotes that all subsequent parameters are keyword parameters:

(fn (&key from to) …)

defines a function that takes two keyword parameters. This function can be invoked in either of the following equivalent ways:

(*(closure)* :from a :to b)

(*(closure)* :to b :from a)

The Yarr interpreter automatically assigns the arguments to the correct parameters. Keyword parameters are automatically optional, and can have default values specified:

(fn (&key (from *default-value*) to) …)

If the **:from** keyword is not provided, **from** is set to *default-value*.

The **&rest** tag causes all arguments not already bound to parameters to be collected into a list, and bound to the parameter immediately following the **&rest** tag.

When specifying parameters, an optional type may be provided using the following syntax:

* *Symbol* – specifying a symbol, as in all of the examples above, indicates that an argument of any type can be bound to the symbol
* (*symbol type)* – for parameters that are not optional or keyword parameters, indicates that *symbol* will be bound to the argument, and that the argument must be of type *type*
* (*symbol default*) – for optional and keyword parameters, indicates that an argument of any type can be bound to the symbol, and if no argument is specified, the symbol will be bound to the value of *default*
* ((*symbol type*) *[default]*) – for optional and keyword parameters, indicates that the argument passed must be of type *type*, and optionally defaults to *default* if no argument is provided

### For

(for *init-expression test-expression iter-expression*

*Expression1*

*Expression2*

*…*

*expressionN*)

First evaluates *init-expression*, which is evaluated only once. Then, if *test-expression* evaluates to *true*, loops over each *expression* in turn, and evaluates *iter-expression*. Repeats this process until *test-expression* evaluates to *false*, at which point the **for** loop evaluates to the value of the last *expression*.

### Foreach

(foreach *symbol IEnumeralbe-object*

*expression1*

*expression2*

*…*

*expressionN*)

This is a synonym for **each**.

### Function

(function *function-reference*)

If *function-reference* is a closure or delegate, evaluates to that closure or delegate. Otherwise *function-reference* is assumed to be a reference to a .NET method, creates a closure wrapping that method, and evaluates to that closure. The There is a reader macro bound to **#'** that converts input of the form **#'***foo* to (function *foo*).

Example:

(apply #’Console.Writeline '("{0} {1}" "hello" "world"))

🡪 hello world

(apply #’+ '(1 2 3)) 🡪 6

(eq + #’+) 🡪 True

### If

(if

*test1 expression1*

*test2 expression2*

*test3 expression3*

*…*

*testN expressionN*

*else-expression*)

This is a synonym for **cond**.

### Incf

(incf *symbol [amount]*)

(++ *symbol*)

Increments the value bound to *symbol* by *amount*. The *amount* parameter is optional, and defaults to 1. If *symbol* is bound to an object, then **incf** will try to invoke the appropriate **op\_addition** method. The (++ *symbol*) special form behaves like post-fix ++ in C# or C++: it increments the value bound to *symbol* but returns the pre-incremented value. If *symbol* is bound to an object, then **++** will attempt to invoke the appropriate **op\_increment** method.

### Lambda

(lambda (*parameters*)

*expression1*

*expression2*

*…*

*expression*)

This is a synonym for **fn**, and exists for historical reasons.

### Let

First form:

(let *var value*

*expression1*

*expression2*

*…*

*expressionN*)

Second form:

(let (*local-declarations*)

*expression1*

*expression2*

*…*

*expressionN*)

Creates a local lexical environment, with local symbols bound, and executes each *expression* in the local environment. The **let** special form evaluates to the value of the last *expression*.

There are two forms of **let**. The first form creates a single local symbol, and binds it to the result of evaluating *value*.

The second form permits multiple local symbols to be declared. A *local-declaration* can be just a symbol, or it can be a list with two elements. If it is a list, the first element is expected to be a either a symbol or a list of the form (*symbol type*). The second element is evaluated, and the result is bound to the symbol. If a *type* is specified, the symbol is declared to be of that *type* in the local environment. The various initialization expressions for the local symbols are evaluated in the caller’s environment.

### Let\*

First form:

(let\* *var value*

*expression1*

*expression2*

*…*

*expressionN*)

Second form:

(let\* (*local-declarations*)

*expression1*

*expression2*

*…*

*expressionN*)

Creates a local lexical environment, with local symbols bound, and executes each *expression* in the local environment. The result of evaluating the special form is the value of the last expression evaluated.

Like the **let** special form, except that **let\*** evaluates the initialization expressions for the local symbols in order, in the local environment.

### Letv

(letv (*variables-list*) (*values-list*)

*expression1*

*expression2*

*…*

*expressionN*)

Creates a local lexical environment, binding each symbol in *variables-list* to the corresponding value in *values-list*, and then executes each *expression* in the local environment. The result is the value of the last expression.

### Lock

(lock *lock-obj*

*expression1*

*expression2*

*…*

*expressionN*)

Locks the object *lock-obj*, and then executes each *expression* in the local environment. Any other thread attempting to lock *lock-obj* will block until **lock** returns. The result is the value of the last expression. This analogous to the **lock (obj) { … }** construct in C#. Note that symbols are objects, so it can be convenient to use a keyword symbol as the *lock-obj*.

### Lock-if

(lock-if *lock-obj bool-expr*

*expression1*

*expression2*

*…*

*expressionN*)

If *bool-expr* evaluates to true, locks the object *lock-obj*, and then evaluates *bool-expr* again. If it is still true, executes each *expression* in the local environment. The result is the value of the last expression. If *bool-expr* is false either before or after obtaining the lock, returns false. If *bool-expr* is false on the initial check, then no lock is obtained and returns false.

### Macro

(macro (*argument-list*)

*expression1*

*expression2*

*…*

*expressionN*)

Creates a Yarr macro. The first argument is a parameter list for the macro. When the macro is subsequently expanded, the arguments will be bound to these parameters, and the *expressions* will be evaluated in order. The result of expanding the macro will be the value of the last *expression* evaluated.

*Parameter lists*

Macro parameter lists are similar to function parameter lists, and support **&optional**, **&key**, and **&rest**. However, parameter types cannot be specified. The macro parameter list supports **&body**, which is identical to **&rest**, and can be used to improve readability of code if used to denote a parameter that is a collection of expressions that will used as the body of a definition (for example, see the definitions of the **defun** or **defmacro** macros). Parameters that are not **&key**, **&optional**, or **&body**/**&rest** support destructuring.

The **macro** special form evaluates to the value of macro that it creates, *i.e.* the result of the macro-expansion is evaluated.

### Nullp

(null *object*)

Returns **true** if *object* is **null**, *false* otherwise.

### Or

(or

*expression1*

*expression2*

…

*expressionN*)

Evaluates each *expression* in turn. If an expression is *true*, no further expressions are evaluated, and the **or** special form returns that expression’s value. If all expressions evaluate to *false*, the **or** special form evaluates to *false*.

### Quote

(quote *object*)

'*object*

The **quote** special form evaluates to its (unevaluated) argument.

### Return

(return *value*)

When encountered as a top-level form in a flow-control special form: **and**, **do**, **for**, **each/foreach**, **let**, **let\***, **or**, **to**, **while**; or at the top-level in a closure or macro, causes that special form, closure, or macro to stop processing and return *value*. If evaluated as part of a **cond** or **if** special form, it is considered to be part of the enclosing form. Otherwise (for example, if encountered as a parameter to a function), simply evaluates to *value*.

### = / Setq / Setf

(= *lvalue1 expression1*

*lvalue2 expression2*

*…*

*lvalueN expressionN*)

Assigns values. Each *lvalue* is examined in turn, and is expected to be either a symbol or a list.

* If it is a symbol, the corresponding *expression* is evaluated and the symbol is bound to its value
* If it is a list
  + The first element of the list is examined, and is expected to be a symbol.
  + If there is a setf method bound to the symbol, it is invoked, and is passed as arguments:
    - The value of the corresponding *expression*
    - The second element of the *lvalue* list
    - The value of any remaining elements in the *lvalue* list, in order
  + If there is no setf method bound to the symbol, it is assumed to be a property or field of a .NET class instance, which is expected to be the second element of the *lvalue* list. The Yarr runtime attempts to set this field or property to the value of the corresponding *expression*

**Setq**, **Setf**, and **=** are synonymous. The preferred syntax is to use **=**; the other two exist for historical reasons. The **=/Setq/Setf** special form evaluates to the value of the last *expression*.

### Symbol-name

(symbol-name *symbol*)

Returns the name of the symbol *symbol* as a string.

### Symbol-value

(= (symbol-value *symbol*) *value*)

Returns the value bound to a symbol. A setf method **symbol-value** is also available that can be used to dereference a symbol, and set the resulting symbol’s value. For example:

(= foo bar)

(= (symbol-value foo) 10)

foo 🡪 bar

bar 🡪 10

### To

(to *var end-value*

*expression1*

*expression2*

*…*

*expressionN*)

Creates a local environment, binds the symbol *var* to zero (*var* declared in the local environment as a **System.Int32**), then evaluates each *expression* in turn. After the last *expression* is evaluated, the value of *var* is incremented, and if it is less than *end-value*, the loop is repeated. When the value of *var* reaches *end-value*, the **to** special form evaluates to the value of the last *expression*.

### Try

(try *expression catch-expression [finally-expresssion]*)

Evaluates *expression* with exception handling. The *expression* is evaluated in a local environment. If an exception is thrown during the evaluation of *expression*, it is caught, bound to the symbol **it** in the local environment, and the **try** special form evaluates the *catch-expression*. If the *finally-expression* exists, it will be evaluated last, regardless of whether or not an exception is thrown. If no exception is thrown, the **try** special form evaluates to the value of *expression*. If an exception is thrown, the **try** special form evaluates to the value of *catch-expression*.

### Unbind

(unbind *symbol1 symbol2 … symbolN*)

Unbinds symbols from the Yarr environment. If the objects bound to these symbols are no longer referenced, they can be garbage collected by the .NET runtime. The **unbind** special form evaluates to the last symbol that was unbound.

### Unbind-setf

(unbind-setf *symbol1 symbol2 … symbolN*)

Unbinds setf methods from symbols. A setf method must be explicitly unbound from a symbol before a new setf method can be bound to that symbol. The **unbind-setf** special form evaluates to the last symbol that had its setf method unbound.

### While

(while *condition*

*expression1*

*expression2*

*…*

*expressionN*)

Loops while a condition is *true*. The *condition* is evaluated, and if it evaluates to *true*, each *expression* is evaluated in turn. This repeats until the *condition* evaluates to false, at which point the **while** special form evaluates to the value of the last *expression.*

### With

(with ([*local-symbol*] *disposable-expr*)

*expression1*

*expression2*

*…*

*expressionN*)

Executes *disposable-expr*, which should return an **IDisposable** object. If *local-symbol* is provided, the result of *disposable-expr* is bound to *local-symbol* in a local environment. Then remaining expressions are evaluated in turn. After the last expression is evaluated, the IDisposable object is disposed, analogous to the **using (obj) {…}** construct in C#. The **with** special form evaluates to the value of the last *expression.*

## Functions

The Yarr runtime contains the following built-in functions:

### Append

(append *list1 list2 … listN*)

Appends *list1 list2 … listN* together and returns the result. **Append** is non-destructive, and calls **copylist** to create shallow copies of each *list*.

### Apply

(apply *function list*)

Applies the function, closure, special form, or macro *function* to the list *list*, and returns the result. The *list* can be “loosely” defined, for example:

(apply *function obj1 obj2 … objN* (*list-items*))

is equivalent to:

(apply *function* (*obj1 obj2 … objN list-items*))

### Aref

(aref *array index1 index2 … indexN*)

Returns the element *array*[*index1,index2,…,indexN*] of a rank *N* array. There is also a built-in **aref** setf method that sets the value of *array*[*index1,index2,…,indexN*].

### Arithmetic Functions + - \* / %

(+ *number1 number2 … numberN*)

Adds all of the *numbers* and returns the sum. For object types, Yarr will attempt to call an appropriately-defined **op\_addition**  or **op\_unaryplus** method.

(- *number1 number2 … numberN*)

Subtracts *number2 … numberN* from *number1* and returns the result. For objects types, will attempt to call an appropriately-defined **op\_subtraction** or **op\_unaryminus** method.

(\* *number1 number2 … numberN*)

Multiplies the *numbers* and returns the product. For object types, will attempt to call an appropriately-defined **op\_multiply** method.

(/ *number1 number2 … numberN*)

Divides *number1* by *number2 … numberN* and retuns the quotient. For object types, will attempt to invoke an appropriately-defined **op\_division** method.

(% *number1 number2*)

(mod *number1 number2*)

Computes and returns *number1* (mod *number2*). **Mod** and **%** are synonyms. The two numbers must be integer types. For object types, will attempt to invoke an appropriately-defined **op\_modulus** method.

The arithmetic functions will return the most general type of their arguments, for example if adding **System.Int32** numbers, the result will be a **System.Int32**, but if one of the numbers is a **System.Double**, the result will be a **System.Double**. In particular, the division function will do integer division if all of its arguments are integer types.

### Array

(array *obj1 obj2 … objN*)

[ *obj1 obj2 … objN* ]

#(*obj1 obj2 … objN*)

Constructs a one-dimensional array containing *obj1 obj2 … objN*. If the objects are all of the same type, the array will be of that type, otherwise it will be a **System.Object[]**.

### Assoc

(assoc *key a-list*)

Searches an *association list* for the member with the specified *key*. An *association list* is a list of cons nodes, and the **car** of each element is assumed to be a key value. The **assoc** function searches the association list until it finds an element whose key matches *key*, and then returns that element. If no element is found, **assoc** returns **null**.

### Atomp

(atomp *object*)

Returns **true** if *object* is an atom, *false* otherwise.

### Bitwise Functions & | ^ ~

(& *number1 number2 … numberN*)

Computes and returns the bitwise and of *number1 number2 … numberN*. For object types, will attempt to invoke an appropriately-defined **op\_bitwiseand** method.

(| *number1 number2 … numberN*)

Computes and returns the bitwise or of *number1 number2 … numberN.* For object types, will attempt to invoke an appropriately-defined **op\_bitwiseor** method.

(^ *number1 number2 … numberN*)

Computes and returns the bitwise exclusive-or of *number1 number2 … numberN.* For object types, will attempt to invoke an appropriately-defined **op\_bitwiseexclusiveor** method.

(~ *number*)

Computes and returns the bitwise inversion of *number.* For object types, will attempt to invoke an appropriately-defined **op\_onescomplement** method.

### Car

(car *cons*)

Returns the first element of the cons node. The built-in setf method **car** sets the first element of a cons node.

### Cdr

(cdr *cons*)

Returns the second element of the cons node. The built-in setf method **cdr** sets the second element of the cons node.

### C*xxx*r

The functions **caar**, **cadr**, **cdar**, **cddr**, **caaar**, **caadr**, **cadar**, **caddr**, **cdaar**, **cdadr**, **cddar**,and **cdddr** are short-hand for (car(car …)),(car(cdr …)),…,(car(car(car …))),…,(cdr(cdr(cdr …))). Each has a corresponding built-in setf method.

### Comparison Functions < <= >= >

(< *number1 number2 … numberN*)

Returns **true** if *number1 < number2 < … < numberN*, otherwise returns **false.** For object types, will attempt to invoke an appropriately-defined **op\_lessthan** method.

(<= *number1 number2 … numberN*)

Returns **true** if *number1 <= number2 <= … <= numberN*, otherwise returns **false.** For object types, will attempt to invoke an appropriately-defined **op\_lessthanequal** method.

(> *number1 number2 … numberN*)

Returns **true** if *number1 > number2 > … > numberN*, otherwise returns **false.** For object types, will attempt to invoke an appropriately-defined **op\_greaterthan** method.

(>= *number1 number2 … numberN*)

Returns **true** if *number1 >= number2 >= … >= numberN*, otherwise returns **false.** For object types, will attempt to invoke an appropriately-defined **op\_greaterthanequal** method.

### Cons

(cons *car-value cdr-value*)

Creates and returns a cons node from the given *car-value* and *cdr-value*.

### Copylist

(copylist *cons*)

Returns a *shallow copy* of the cons node *cons*.

### Elt

(elt *object index*)

Returns the element at index *index* on the object *object*. This is useful for obtaining items from .NET collections, and there is also a corresponding setf method. For example:

(= foo (new System.Collections.Hashtable))

(= (elt foo "a") 10)

(elt foo "a") 🡪 10

### Eq

(eq *obj1 obj2 … objN*)

Returns **true** if all of the *obj* reference the same object (reference equality) or have the same value type. Otherwise returns **false**.

### Eql / ==

(eql *obj1 obj2 … objN*)

(== *obj1 obj2 … objN*)

If *obj1****.*Equals(***obj2***)**, *obj2***.Equals(***obj3***)**, … , **.Equals(***objN***)** (value equality) returns **true**, otherwise returns **false**. **Eql** and **==** are synonyms.

### Eval

(eval *form*)

Evaluates its argument (assumed to be a Yarr expression) and returns the result of the evaluation.

### Evalstring

(evalstring *string*)

Evaluates its argument (assumed to be a string containing Yarr code) and returns the result of the evaluation.

### Evenp

(evenp *number*)

Returns *true* if *number* is an even number, *false* otherwise.

### Every

(every *function* *list1 list2 … listN*)

Evaluates *function*, passing it the first element of each *list* as an argument (so *function* should expect as many arguments as there are *lists*). If this value is *true*, continues evaluating *function*, passing it the next element of each *list*, until one of the *lists* runs out of elements or an evaluation is *false*. Returns **true** if every evaluation of function was *true*; returns **false** if any evaluation of *function* was *false*. Stops evaluating *function* once a *false* value is encountered.

### Exit

(exit *exit-code*)

Exits the current process. If an *exit-code* is provided, it is the exit code of the process. If the symbol **\*no-exit\*** is bound in the current environment, then **exit** does nothing.

### First

(first *list*)

A synonym for **car**. There is also a built-in setf method **first**.

### Get

(get *symbol property*)

Gets the value for a *symbol*’s specified *property*. There is also a setf method **get** for setting a symbol’s property value.

### Gensym

(gensym)

Generates and returns a unique symbol. Each call to **gensym** will generate a new unique symbol.

### Identity

(identity *obj*)

The identity function; returns its argument. Intended for use with functions that require a function as an argument, but for which a no-op is desired.

### In-package

(in-package *package-name*)

Sets the current package to the package named *package-name*. If no such package exists, an exception is thrown.

### Is

(is *type object*)

If *object* is of type *type*, returns **true** otherwise returns **false**.

### Length

(length *object*)

If *object* is null, returns zero. Otherwise, if *object* is an instance of a .NET class that implements a **Length** method or property, it invokes that method and returns the value. If *object* is an instance of a .NET class that does not implement a **Length** method or property, then **length** returns **null**.

### List

(list *obj1 obj2 … objN*)

Creates and returns the list (*obj1 obj2 … objN*)

### Listp

(listp *object*)

Returns **true** if *object* is a cons node, *false* otherwise.

### Load

(load *script-name*)

Loads the previously defined CopilotBuilder script with name *script-name* and evaluates every expression in the script in an implied **do** block. Evaluates to result of the last expression in the script. If no script with name *script-name* is found, an exception is thrown.

### Macroexpand, Macroexpand-1

(macroexpand '(*macro obj1 obj2 … objN*))

(macroexpand-1 '(*macro obj1 obj2 … objN*))

Expands the Yarr macro *macro*, using *obj1 obj2 … objN* as the macro’s arguments, and returns the resulting macro expansion without evaluating it. The **macroexpand-1** function will expand the macro once; the **macroexpand** function will iteratively expand the macro until a non-macro results:

(defmacro macro-1 (x) `(macro-2 ,x))

(defmacro macro-2 (x) `(\* ,x ,x))

(macroexpand-1 '(macro-1 3)) 🡪 (macro-2 3)

(macroexpand '(macro-1 3)) 🡪 (\* 3 3)

### Make-package

(make-package *package-name* &key *use shadow)*

Creates or modifies a symbol package. If the package named *package-name* does not exist, it is created. If the *use* parameter is present, it is assumed to be a list of existing packages and these symbols are inherited by the package. If the *shadow* parameter is present, it is assumed to be a list of symbol names, and these are created in the package.

### Map

(map *function list1 list2 … listN*)

Invokes function *function*, providing it each element in the lists in turn as its arguments, collects the return values into a list of values, and returns that list of values. For example:

(map + ’(1 2 3) ’(4 5 6) ’(7 8 9)) 🡪 (12 15 18)

### Mapc

(mapc *function list1 list2 … listN*)

Invokes function *function*, providing it each element in the lists in turn as its arguments. Like **map**, except that **mapc** does not collect the return values. Instead, it returns *list1*.

### Mapcar

(mapcar *function list1 list2 … listN*)

This is a synonym for **map** and exists for historical reasons.

### Mapl

(maplist *function list1 list2 … listN*)

Like **maplist**, but does not collect the return values. Instead, it returns *list1*.

(maplist append ’(a b c) ’(1 2 3))

🡪 ((a b c 1 2 3) (b c 2 3) (c 3))

### Maplist

(maplist *function list1 list2 … listN*)

Like **map**, but applies *function* to the list of successive **cdr** of each *list*:

(maplist append ’(a b c) ’(1 2 3))

🡪 ((a b c 1 2 3) (b c 2 3) (c 3))

### Max

(max *number1 number2 … numberN*)

Returns the maximum value of *number1 number2 … numberN*

### Md-array

(md-array *rank*  *list*)

#*rank***a**(…)

Transforms *list* into a multi-dimensional array of rank *rank*, and returns the array. If every object in the list is of the same type, the array created will be of that type, otherwise the array will be a **System.Object[]**.

### Member

(member *object IEnumerable-object*)

If *object* is contained in *IEnumerable-object*, returns **true**. Otherwise returns **false**. Note that Yarr cons nodes implement **IEnumerable**.

### Min

(min *number1 number2 … numberN*)

Returns the minimum value of *number1 number2 … numberN*

### Minusp

(minusp *number*)

Returns *true* if *number* is a negative number, *false* otherwise.

### Nconc

(nconc *list1 list2 … listN*)

Destructively concatenates *list1 list2 … listN* into a new list, and returns that list. Unlike **append**, the **nconc** function directly modifies its arguments to stitch them together.

### New

(new *type arg1 arg2 … argN*)

Creates a new instance of a .NET class or Yarr structure, passing *arg1 arg2 … argN* as arguments to the class constructor, and returns the instance. If *type* is a generic .NET type, then *arg1* is expected to be a **System.Type[]** array, which is used to resolve the generic parameters of the *type.*

### Not

(not *value*)

If *value* is *false*, returns **true**, otherwise returns **false**.

### !=

(!= *obj1 obj2 … objN*)

Returns the equivalent of (not (== *obj1obj2 … objN*))

### Notany

(notany *function* *list1 list2 … listN*)

Evaluates *function*, passing it the first element of each *list* as an argument (so *function* should expect as many arguments as there are *lists*). If this value is *false*, continues evaluating *function*, passing it the next element of each *list*, until one of the *lists* runs out of elements or an evaluation is *false*. Returns **true** if every evaluation of function was ***false***; returns **false** if any evaluation of *function* was *true*. Stops evaluating *function* once a *true* value is encountered.

### Notevery

(notevery *function* *list1 list2 … listN*)

Evaluates *function*, passing it the first element of each *list* as an argument (so *function* should expect as many arguments as there are *lists*). If this value is *true*, continues evaluating *function*, passing it the next element of each *list*, until one of the *lists* runs out of elements or an evaluation is *false*. Returns **false** if every evaluation of function was ***true***; returns **true** if any evaluation of *function* was *false*. Stops evaluating *function* once a *false* value is encountered.

### Nreverse

(nreverse *list*)

Destructively rearranges *list* so that its items are in reverse order. Unlike **reverse**, the **nreverse** function does not create any new cons nodes. Instead the existing cons nodes of *list* are rearranged.

### Nth

(nth *N IEnumerable-object*)

Returns the *N*’th item in *IEnumerable-object*. There is also a built-in setf method that can set the value of this item.

### Oddp

(oddp *number*)

Returns *true* if *number* is an odd number, *false* otherwise.

### Plusp

(plusp *number*)

Returns *true* if *number* is a positive number, *false* otherwise.

### Pr

(pr *[target] obj1 obj2 … objN*)

Writes the **.ToString()** representation of each object in turn to *target*, which is expected to be a **System.IO.TextWriter**. If the first argument is not a **System.IO.Textwriter**, then **pr** will use the object bound to **\*output\*** as its target, which by default is the output of the script. If the output target is a **System.IO.TextWriter** instance, the **pr** function will write to that instance.

### Prl

(prl *[target] obj1 obj2 … objN*)

Like **pr**, but writes a newline after the last object.

### Read

(read *[textreader [eof-value]]*)

Reads, parses (but does not evaluate), and returns the next Yarr expression from the **System.IO.TextReader** *textreader*. If an *eof-value* is specified, the reader will return that value if *textreader* is already at **EOF**, otherwise it will return -1 if *textreader* is at **EOF**. If no *textreader* is specified, the **read** function will read from the **System..IO.TextReader** bound to the **\*input\*** symbol.

### Reduce

(reduce *function sequence*

&key *key start end initial-value from-end*)

Uses *function* to combine the elements of the *sequence*. The *function* must accept two arguments, which will be either elements of *sequence* or the result of previous *function* evaluations, and the *function* must also be able to accept no arguments.

The keyword parameters are:

**Key** if supplied, specifies a function that is applied to each element of *sequence* before it is processed by *function*

**Start** if supplied, starts processing *sequence* at this position. If not provided, starts processing at position 0

**End** if supplied, stops processing *sequence* at this position. If not provided, stops processing at the last element of *sequence*

**Initial-value**

if supplied, this is passed along to *function* along with the first *sequence* element to be processed

**From-end**

if *true*, then *function* is applied so that it is right-associative. Otherwise it will be left-associative

The result of **reduce** is the combined result of *function* being applied to successive pairs of elements of *sequence*. If *sequence* contains exactly one element, and no *initial-value* is given, then that element is returned and *function* is not called. If *sequence* is empty and an *initial-value* is given, then the *initial-value* is returned and *function* is not called. If *sequence* is empty and no *initial-value* is given, then *function* is called with zero arguments, and **reduce** returns whatever *function* does. This is the only case where the *function* is called with no arguments.

### Reference

(reference *assembly-name1 assembly-name2 … assembly-nameN*)

References the named assemblies, so that Yarr can access them. If any of the assemblies are not yet loaded into the current **System.AppDomain**, they are loaded at this time. Returns the **System.Assembly** instance of the last assembly in the list.

### Reset

(reset)

Resets the current lexical environment, deleting all symbol bindings that are not built-in Yarr objects. Evaluates to **null**.

### Rest

(rest *list*)

A synonym for **cdr**. There is also a built-in setf method **rest**.

### Reverse

(reverse *list*)

Returns a copy of *list*, with the elements in reverse order.

### Second

(second *list*)

A synonym for **cadr**. There is also a built-in setf method **second**.

### Shadow

(shadow *symbol-names* &optional *package*)

Creates the symbols specified by *symbol-names*, which can be either a single symbol name or a list of symbol names, in the package specified by *package*. If *package* is not provided, the symbols are created in the current package.

### Some

(some *function* *list1 list2 … listN*)

Evaluates *function*, passing it the first element of each *list* as an argument (so *function* should expect as many arguments as there are *lists*). If this value is *false*, continues evaluating *function*, passing it the next element of each *list*, until one of the *lists* runs out of elements or an evaluation is *true*. Returns **true** if some evaluation of function was *true*; returns **false** if all evaluations of *function* were *false*. Stops evaluating *function* once a *true* value is encountered.

### The

(the *type object*)

Attempts to convert *object* to type *type*. If *type* is **Yarr.Cons**, the function can convert any of the following types of *object* to a **Yarr.Cons**:

* **System.String**
* **System.Array**
* **System.Collections.Hashtable**
* **System.Collections.SortedArray**
* **System.Data.DataRow**
* **System.Data.DataTable**
* Any class that implements **ICollection**
* Any class that implements **IEnumerable**

If *object* is a **Yarr.Cons**, the function can convert the **Yarr.Cons** instance to any of the following types:

* **System.String**
* **System.Array**
* **System.Collections.Hashtable**
* **System.Collections.Queue**
* **Systems.Collections.Stack**
* **Systems.Collections.SortedList**

The function will convert any type to **System.String** by calling the object’s **ToString()** method.

When converting any object to **System.Boolean**, any numeric zero value, null, and any empty **System.String** is converted to **false**; everything else is converted to **true**.

A **null** can be cast to any reference type (but not to any value type), and the Yarr runtime will track that **null**’s type.

If the function cannot figure out how to covert *object* to *type*, it will look for appropriately defined **op\_explicit** and **op\_implicit** methods on the specified type. If none are found, it invokes **System.Convert.ChangeType()** as a last resort.

### Third

(third *list*)

A synonym for **caddr**. There is also a built-in setf method **third**.

### Throw

(throw *exception-message*)

Throws an exception. If *exception-message* is an exception object, it is thrown. Otherwise throws a new **Yarr.YarrException**, and sets the exception’s **message** property to *exception-message***.ToString()**. Since this function throws an exception, it does not return.

### Type

(type *base-type* &rest *args*)

Creates a specific instance of a generic type. If *base-type* is not generic, evaluates to *base-type*. Otherwise, *args* are considered to be types, and these are used to resolve *base-type* into a specific type. For example:

(type 'int32) 🡪 #<System.Int32>

(type 'dictionary string string)

🡪#<System.Collections.Generic.Dictionary`2[System.String,

System.String]>

Note that the *base-type* should be quoted since it is not a valid .NET type as specified (the actual .NET type name is decorated to indicate the number of generic parameters).

### Typeof

(typeof *object*)

Returns the **System.Type** instance for the object’s type.

### Use-package

(use-package *package-name-list* &optional *package*)

Specifies that the package *package* should inherit the symbols from the packages specified by *package-name-list*, which can be either a single package name or a list of package names. If *package* is not specified, the symbols will be inherited by the current package.

### Using

(using *namespace1 namespace2 … namespaceN*)

Registers the namespaces with the Yarr runtime, so that types defined in those namespaces can be referenced in Yarr code without fully qualifying the types with their namespace.

### Zerop

(zerop *number*)

Returns *true* if *number* is zero, *false* otherwise.

## Macros

The Yarr runtime contains the following built-in macros:

### -> and ->> (thread first/thread last)

(-> obj *form1 form2 … formN*)

(->> obj *form1 form2 … formN*)

These macros provide a more readable syntax for nested function calls. The **->** macro (thread first) will insert *obj* as the first parameter in *form1*, and then inserts the result of each call as the first parameter of the next call. For example:

(-> "A String To Process" :Trim :ToLower (:Replace " " "\_"))

Is evaluated as:

(:Replace (:ToLower (:Trim "A String To Process")) " " "\_")

The **->>** macro (thread last) is similar, except that it inserts each result as the last parameter of the next call, rather than the first parameter.

### Bind-disptch-character

(bind-macro-character *dispChar char function*)

Sets the Yarr *function* as the dispatch reader macro for *dispChar* and *char*. The *function* should have the following signature:

(fn (reader readtable chr arg) *…*)

Matching the signature of the **Yarr.DispatchReaderMacro** delegate.

### Bind-macro-character

(bind-macro-character *char function*)

Sets the Yarr *function* as the reader macro for *char*. The *function* should have the following signature:

(fn (reader readtable chr) *…*)

which matches the signature of the **Yarr.ReaderMacro** delegate.

### Defun

(defun *name parameter-list*

*Expression1*

*Expression2*

*…*

*ExpressionN*)

Defines a Yarr closure using the *parameter-list* and *expressions*, and binds it to the symbol *name*.

### Defmacro

(defmacro *name parameterlist*

*Expression1*

*Expression2*

*…*

*expressionN*)

Defines a Yarr macro using the *parameter-list* and *expressions*, and binds it to the symbol *name*.

### Defsetf

(defsetf *name parameterlist*

*Expression1*

*Expression2*

*…*

*expressionN*)

Defines a Yarr setf method using the *parameter-list* and *expressions*, and binds it to the symbol *name*.

### Make-array

(make-array *type rank-array [bound-array]*)

Creates a .NET **System.Array** instance of type *type*. The *rank-array* parameter is expected to be a **System.Int32[]** or **System.Int64[]**, and specifies the size of each rank of the array. The *bound-array* is optional. If provided, it must be of the same size and type as *rank-array*, and specifies the lower bound for each rank. If *rank-array* is omitted, the resulting array will be zero-based by default.

### Mapcan

(mapcan *function* *list1 list2 … listN*)

Like **map**, but uses **nconc** to combine the results into one list.

### Mapcon

(mapcon *function* *list1 list2 … listN*)

Like **maplist**, but uses **nconc** to combine the results into one list.

### Unbind-dispatch-character

(unbind-dispatch-character *dispChar char*)

Unbinds the dispatch reader macro assigned to *dispChar* and *char*.

### Unbind-macro-character

(unbind-macro-character *char*)

Unbinds the reader macro assigned to *char*.

### When

(when *test-expression*

*expression1*

*expression2*

*…*

*expressionN*)

Evaluates the *test-expression*. If it evaluates to *true*, evaluates each *expression* in turn, and returns the value of the last *expression.* If the *test-expression* evaluates to *false*, returns **null**.

## System symbols

The Yarr runtime makes use of the following symbols. All of these symbols are dynamically scoped, unless otherwise noted:

### \*environment\*

An object representing the current environment.

### \*last-exception\*

Whenever the Yarr runtime encounters an exception it binds the exception object to **\*last-exception\***. Scripts can then obtain the exception object from the environment, if it is needed.

### \*print-circle\*

This symbol, if set to true, causes circular cons references to print using the reader abbreviation macro syntax, **#n= … #n#** which allows the printed form to be read by the Yarr reader to recreate the data structure. If false, printing of circular references is controlled by **\*print-length\*** and **\*print-level\***. This symbol is false by default.

### \*print-length\*

If **\*print-circle\*** is false, this symbol controls the print length of lists. Printing will be truncated after the number of elements specified by the value bound to this symbol. This value is not set by default, in which case the printer uses a default value of 20. This is used to prevent circular references from resulting in a non-terminating loop.

### \*print-level\*

If **\*print-circle\*** is false, this symbol controls the print level of nested lists. Printing will be truncated after reaching the number of nested levels specified by the value bound to this symbol. This value is not set by default, in which case the printer uses a default value of 6. This is used to prevent circular references from resulting in a non-terminating loop.

### \*read-eval\*

This symbol, if set to true, enables the “**#.**” reader dispatch macro. This macro causes the next form read to be evaluated immediately (at read time), and returned as the result of the reader invocation. This symbol is true by default. If false, use of the “**#.**” Dispatch reader macro will cause the runtime to throw an exception.