**CopilotBuilder Scripting Guide**

**A Developer’s Guide to Yarr**

**Ray Comas**

**CopilotBuilder**

[Introduction 4](#_Toc198229008)

[What is Yarr? 4](#_Toc198229009)

[The Yarr syntax (LISP 101) 4](#_Toc198229010)

[A Yarr tutorial 5](#_Toc198229011)

[Overview 6](#_Toc198229012)

[Creating and manipulating lists 6](#_Toc198229013)

[Symbols and variables 8](#_Toc198229014)

[Arrays 9](#_Toc198229015)

[Closures, functions, and special forms 12](#_Toc198229016)

[Parameter types 16](#_Toc198229017)

[Optional parameters 16](#_Toc198229018)

[Keyword parameters 17](#_Toc198229019)

[Variable arguments 17](#_Toc198229020)

[Return types 18](#_Toc198229021)

[Lexical and Dynamic Scope 18](#_Toc198229022)

[Multiple Dispatch Methods 19](#_Toc198229023)

[Before and After Methods 21](#_Toc198229024)

[Around Methods 23](#_Toc198229025)

[Extension Methods 25](#_Toc198229026)

[Flow control 26](#_Toc198229027)

[Do, Let, Let\* 26](#_Toc198229028)

[Cond, If, When 26](#_Toc198229029)

[Equality and Identity 27](#_Toc198229030)

[And, or 28](#_Toc198229031)

[While 29](#_Toc198229032)

[For 29](#_Toc198229033)

[To 29](#_Toc198229034)

[Foreach 29](#_Toc198229035)

[Case 30](#_Toc198229036)

[Try, Throw 31](#_Toc198229037)

[Map and Reduce 31](#_Toc198229038)

[Map, Mapcar 32](#_Toc198229039)

[Mapc 33](#_Toc198229040)

[MapList 33](#_Toc198229041)

[Mapl 33](#_Toc198229042)

[Reduce 34](#_Toc198229043)

[Types 35](#_Toc198229044)

[Structures 35](#_Toc198229045)

[Local declarations 38](#_Toc198229046)

[Symbol Packages 39](#_Toc198229047)

[The keyword package 42](#_Toc198229048)

[Property lists 43](#_Toc198229049)

[Meta-Programming 44](#_Toc198229050)

[Macros 44](#_Toc198229051)

[Setf methods 45](#_Toc198229052)

[Reader macros 46](#_Toc198229053)

[Dispatch reader macros 48](#_Toc198229054)

[Scripting and Workflows 50](#_Toc198229055)

[Script Input/Output 50](#_Toc198229056)

[Sharing Environments 51](#_Toc198229057)

[Referencing Other Scripts 52](#_Toc198229058)

# Introduction

The CopilotBuilder platform allows users to write scripts that can be included as workflow steps. A script will receive the output of the previous step (or the input into the workflow, if it’s the first step), and the output of the script will be passed on to the next workflow step.

This permits users to examine and transform data as they make their way through the workflow, for example, adding additional data from outside sources or transforming data to/from a certain format.

# What is Yarr?

The scripting language used by CopilotBulder is a subset of Common LISP. It was developed by the CopilotBuilder team for use as a scripting language during workflow processing. We built Yarr to be able to seamlessly interoperate with Microsoft .NET libraries, allowing Yarr scripts to natively invoke and reference Microsoft .NET assemblies.

# The Yarr syntax (LISP 101)

The basis of all LISP-like languages is the *s-expression* (“symbolic expression”)*,* or *list.* This is a list of *atoms* enclosed in parenthesis, for example:

(1 2 3)

(alpha beta Charlie delta)

(an s-expression (with a nested) s-expression)

Notice in the 3rd example that a list can have lists nested within.

Yarr expressions are written as lists, using prefix notation: the first element of a list is assumed to be a method or function, and the rest of the list is assumed to contain the arguments.

This reliance on lists gives the language great flexibility. Because Yarr functions are themselves lists, they can be processed exactly like data. This permits writing scripts which create or manipulate other scripts. Yarr provides several features that facilitate this kind of meta-programming.

Many modern LISP implementations encourage a functional programming style, and Yarr is no exception. Most language elements in Yarr are first-class objects, including types and functions, which encourages good programmers to write concise, recursive code. Yarr scripts can potentially do more work with less written code than programs written in a procedural language like C#.

Unlike many other scripting languages, it is possible to alter the Yarr syntax at runtime through the use of *reader macros*, which are described later in this document. This allows users to customize the “look and feel” of the Yarr scripting code to meet domain-specific requirements.

# A Yarr tutorial

Users can create and execute Yarr scripts from within CopilotBuilder. Select “Scripts” from the left navigation bar:

A screenshot of a computer

AI-generated content may be incorrect.

Then click the “Add Script” button in the upper right:

A screenshot of a computer

AI-generated content may be incorrect.

The resulting form has two boxes: a “Script Code” text box where the user enters their script code, and a “Script Output” text box where CopilotBuilder shows the result of executing the script by clicking the “Run Script” button.

Note that all CopilotBuilder scripts run server-side, so if a script is to make use of external resources such as databases, web endpoints, etc., those resources must be accessible from the CopilotBuilder server, not just from the user’s web browser.

Code in this tutorial can be run by copying it into the “Script Code” text box, and then clicking “Run Script” to see the results. In this document, text before the 🡪 is script code, and text after the 🡪 is the result of evaluating the script code.

## Overview

When Yarr encounters a list, it expects the first element of the list to be a function, and the rest of the list contains the parameters to the function. Yarr is a *symbolic language*. Other languages such as JavaScript or C# have variables, and values are assigned to variables. Yarr has *symbols*, which are first class objects. We can assign values to symbols (called *binding* the symbol), and treat them as variables, but symbols can exist and be used without having values bound to them.

Scripts produce output by calling the **pr** and **prl** functions, which convert their parameters to strings and write them out. The **prl** function is the same as the **pr** function, but terminates the existing line so that the next output will be on a new line. If neither function is called in the script, then the script output will be the result of the last statement executed.

## Creating and manipulating lists

One way to create lists is with the **list** function:

(list 1 2 3)

🡪 (1 2 3)

(list a b (list c d) e)

🡪 (a b (c d) e)

In the second example, we created a nested list. The Yarr interpreter evaluates a function’s parameters before calling the function, so the expression (list c d) is evaluated first, producing (c d), and then the outer expression is evaluated. In the second expression, the symbols **a, b, c, d,** and **e**don’t have any values bound to them, so they evaluate to themselves.

It’s also possible to create a list using the **quote** function:

(quote (1 2 3))

🡪 (1 2 3)

'(1 2 3)

🡪 (1 2 3)

(quote (a b (c d) e))

🡪 (a b (c d) e)

'(a b (c d) e)

🡪 (a b (c d) e)

**quote**, which can be abbreviated to a single quote character, tells Yarr to not process the list that follows it, but to simply return it as-is.

Internally, a list is represented as a pair, called a *cons* (for “construct”) *node*. The first member of the pair is the first element of the list, and the second member is the rest of the list. For historical reasons, the first element of the cons node is called the **car**, and the second element is called the **cdr**. The functions **first** and **rest** are synonyms for **car** and **cdr**:

(car '(a b c))

🡪 a

(first '(a b c))

🡪 a

(cdr '(a b c))

🡪 (b c)

(rest '(a b c))

🡪 (b c)

Notice the use of **quote** (') in the above examples. Normally, a function’s parameters are evaluated before calling the function, so without the quote:

(car (a b c))

The Yarr runtime would try to evaluate (a b c) as a call to some function **a** (or a call to a method named **a** on the .NET object **b**). The quote prevents this evaluation.

The **car** and **cdr** functions can be nested to get particular list elements, and there is also an **nth** function to get an element at a particular list position (zero-based):

(car (cdr '(a b c)))

🡪 b

(cadr '(a b c))

🡪 b

(nth 1 '(a b c))

🡪 b

In the second example, the **cadr** function provides a short-hand for (car (cdr …)). There are similar functions **caar, cdar, cddr, caaar, caadr, cadar, caddr, cdaar, cdadr, cddar,** and **cdddr,** familiar to LISP and Scheme programmers, that do what is expected.

If we have two objects, and want to create a cons node, we use the **cons** function:

(cons a '(b c))

🡪 (a b c)

(cons a b)

🡪 (a . b)

In the second example, we create a cons node that is not a list: it’s just a pair of objects, called a *dotted pair* because of how it is represented. A list has a specific internal representation using nested cons nodes. In other words, all lists are (nested) cons nodes, but not all cons nodes are lists.

Lists can be appended together with the **append** function, which takes any number of list parameters, appends them all together:

(append '(a b c) '(1 2 3) '(d e f))

🡪 (a b c 1 2 3 d e f)

## Symbols and variables

Now that we can create lists, we need a way of storing them for later use. In Yarr, we do this by binding them to symbols. This is equivalent to assigning an object to a variable in other programming languages. However, in Yarr symbols are more than just references to memory locations, and don’t necessarily need to have values bound to them, which is why we can write expressions such as (list a b) without having to define what **a** and **b** are.

The assignment function in Yarr is **=,** which for historical reasons has synonyms **setq** and **setf**:

(= bar '(a b c))

🡪 (a b c)

(= foo (+ 3 4 5))

🡪 12

(= foo (+ 3 4 5))

(\* FOO 5)

🡪 60

Note that, as in the first example, the assignment function returns the value that was assigned.

In the second and third examples, we introduce the arithmetic functions **+** and **\***. Most Yarr arithmetic functions are *n-ary*, that is, they can take an arbitrary number of parameters, as shown in the example with the addition function. Note also that Yarr symbols are case-insensitive, so **foo, Foo, fOo, foO**, etc. are all the same symbol.

It’s possible to make multiple assignments at once:

(= foo '(a b c) bar '(1 2 3))

(prl foo)

(prl bar)

🡪

(a b c)

(1 2 3)

When making multiple assignments, the value returned is the last one assigned.

The assignment function has more capability than simple value binding, for example:

(= foo '(a b (c d) e))

(prl "initial foo = " foo)

(= (car foo) f)

(prl "after setting (car foo) = " foo)

(= (nth 2 foo) g)

(prl "after setting (nth 2 foo) = " foo)

(= (rest foo) '(x y z))

(prl "after setting (rest foo) = " foo)

🡪

initial foo = (a b (c d) e)

after setting (car foo) = (f b (c d) e)

after setting (nth 2 foo) = (f b g e)

after setting (rest foo) = (f x y z)

In most cases, if a Yarr expression retrieves a value, then assigning to that expression will set the value.

## Arrays

Yarr has comprehensive support for .NET arrays. The easiest way to define an array is using the special array syntax:

(= foo [1 2 3])

🡪 [1 2 3]

For those familiar with Common LISP, Yarr also supports the Common LISP array syntax:

(= foo #(1 2 3))

🡪 [1 2 3]

The pound-sign (#) in the example above indicates that (1 2 3) is an array, not a list. Continuing the example, we can use the **typeof** function to see that this is an actual .NET array:

(= foo [1 2 3])

(typeof foo)

🡪 #<System.Int32[]>

We can create arrays of objects as well:

(= foo [1 a "string" 3.0])

(typeof foo)

🡪

#<System.Object[]>

If all of the elements of an array are of the same type, then Yarr will create a .NET array of that type. Otherwise, it will create an array of objects, as shown in the previous example.

There is also support for nested arrays:

(= foo [[1 2 3] [4 5 6] [7 8 9]])

(typeof foo)

🡪

#<System.Int32[][]>

The […] and #(…) syntax are just a short-hand for calling the **array** function, so the following are equivalent:

(prl [1 2 3])

🡪

[1 2 3]

(prl #(1 2 3))

🡪

[1 2 3]

(prl (array 1 2 3))

🡪

[1 2 3]

We can also create multidimensional arrays, using the syntax borrowed from Common LISP:

(= foo #2a((1 2 3)(4 5 6)(7 8 9)))

(typeof foo)

🡪

#<System.Int32[,]>

The syntax is #*nn*a(…) where *nn* specifies the *rank* of the array. The one-dimensional case is so common that Yarr assumes the “1a,” but it can still be supplied:

(= foo #1a(1 2 3 4))

🡪

[1 2 3 4]

For rank > 1, the #*nn*a(…) syntax is really a short-hand for calling the **md-array** function, so the following are equivalent:

#2a((1 2)(3 4))

🡪

#2a((1 2)(3 4))

(md-array 2 '((1 2)(3 4)))

🡪

#2a((1 2)(3 4))

Note that unlike the **array** function, which puts all of its arguments into a one-dimensional array, the **md-array** function explicitly takes a list as an argument and constructs an array out of it.

Since these are .NET arrays, we can call their various .NET array methods:

(= foo [1 2 3 4])

(prl (length foo))

(prl "foo.length = " foo.length)

🡪

4

foo.length = 4

Note that .NET fields and properties can be accessed directly, so we can use foo.length as well as (foo.length) or (length foo) to get the value.

We can access arrays using the .NET **GetValue/SetValue** methods, but an easier way is to use the built-in **aref** function:

(= foo [1 2 3 4])

(pr (aref foo 0))

(pr (aref foo 2))

🡪

1 3

(= bar #2a((1 2)(3 4)))

(pr (aref bar 0 0))

(pr (aref bar 0 1))

(pr (aref bar 1 0))

(prl (aref bar 1 1))

(= (aref bar 0 1) 10)

(prl bar)

🡪

1 2 3 4

#2a((1 10)(3 4))

The last example shows how to assign values to array elements.

Just as we can reference .NET types directly by name, for example using the symbol **int32** to refer to the type **System.Int32**, we can also refer to array types using symbol names. The convention used is that, for a given type **T**, an array of **T** is referenced as **T\*** and a multi-dimensional array of **T** is referenced as **T\**n*** where ***n*** is the rank. For example:

Int32\* ; Array of Int32

🡪

#<System.Int32[]>

Int32\*2 ; Rank 2 array of Int32

🡪

#<System.Int32[,]>

Int32\*\* ; Array of array of Int32

🡪

#<System.Int32[][]>

Int32\*2\* ; Array of rank 2 array of Int32

🡪

#<System.Int32[,][]>

This gives us a way to create arrays, using the **new** function:

(new int32\* 5)

🡪

[0 0 0 0 0]

(new int32\*2 3 2)

🡪

#2a((0 0) (0 0) (0 0))

There is also a **make-array**, which is more efficient than **new** at making arrays, since it invokes **System.Array.CreateInstance** directly instead of going through **System.Activator** as **new** does:

(make-array int32 3 2)

🡪

#2a((0 0) (0 0) (0 0))

(make-array int32 5)

🡪

[0 0 0 0 0]

## Closures, functions, and special forms

Yarr is a *lexically scoped* language. When a script executes, Yarr provides it with its own *environment*. This environment provides, among other things, a symbol table where symbols are defined, and the lexical scope where symbols are bound to objects. When a Yarr function is defined, Yarr ensures that the function *encloses* the environment in which it is defined. When the function is later invoked, it is invoked in this enclosed environment. This pair—the function, and the associated environment—is called a *closure*. We will often use the words *function* and *closure* interchangeably. We’ll elaborate on this later in this document.

Flow control is Yarr is handled with *special forms*. A special form is similar to a closure, except that it does not have an enclosed environment (it always executes in the same environment as its caller), and the arguments are not evaluated before the special form is called. Special forms are the Yarr analog of language keywords, such as **if**, **while**, **for**, etc. in other languages.

Functions in Yarr are defined using the special form **fn**. For historical reasons, **lambda** can be used as a synonym for **fn**.

The **fn** special form creates a closure:

(fn (x y) (+ x y))

Creates a closure with two parameters, **x** and **y**. The symbols **x** and **y** are bound to the function’s arguments in an environment local to the function. The result of evaluating this function is the sum of the two arguments:

(= x 10 y 20) ;; sets x = 10, y = 20

(prl (+ x y))

(prl ((fn (x y) (+ x y)) 5 6))

🡪

30

11

Note the third line of the example: when Yarr evaluates ((fn (x y) (+ x y)) 5 6) it evaluates the first element of the list, which is (fn (x y) (+ x y)). This evaluates to a closure taking two arguments. Next, Yarr evaluates (*anonymous-closure* 5 6) where *anonymous-closure* is the closure returned by **fn,** and the function is invoked with **x** bound to 5 and **y** bound to 6 *in the enclosed environment*. So the addition adds 5 and 6, not 10 and 20.

More generally, the syntax for **fn** is:

(fn (*argument-list*) *statements*)

Closures return the value of the last statement executed. It is possible to return from a closure (or any special form that executes a sequence of statements) at any time by using the **return** special form:

((fn (x y) (if (< x y) (return x)) (+ x y)) 3 4)

🡪

3

((fn (x y) (if (< x y) (return x)) (+ x y)) 4 3)

🡪

7

If we want to re-use a closure, we can bind it to a symbol:

(= my-function (fn (x y) (+ x y)))

(my-function 3 4)

🡪

7

The **defun** macro (macros will be covered later) simplifies the syntax, so that we can write the equivalent:

(defun my-function (x y) (+ x y))

Up until now, every Yarr statement we’ve evaluated has been evaluated in the *root environment*, which is the environment where all of Yarr’s built-in functions are defined. It’s possible to create *local environments*, which are the Yarr analogue to creating local scopes with {…} in JavaScript or C#.

A local environment is automatically created when a Yarr closure is called, which contains the arguments of the function call. It is also possible to explicitly create a local environment using the **let** special form:

(= a 3 b 4) ;; set a =3, b = 4 “outside” the let

(let ( (a 10) (b 20) ) ;; set a = 10, b = 20 “inside” the let

(prl "a+b inside let = " (+ a b)))

(prl "a+b outside let = " (+ a b))

🡪

a+b inside let = 30

a+b outside let = 7

The syntax for **let** is:

(let ( (*var1 val1*) (*var2 val2*) … )

*Statements*)

When the *statements* execute, any references to *var1, var2, …, varN* are resolved in the “inner” environment. References to any other symbols are resolved in the “outer” environment. The result of evaluating the **let** is the value of the last statement executed.

If only one local variable is required, the **let** syntax can be simplified a bit:

(let temp 10 (+ temp temp))

🡪

20

Now for a more complicated example, let’s examine what it means for something to be a *closure*:

(= a 3 b 4)

(let ( (a 10) (b 20) )

(defun foo (x) (+ a b x)))

The first statement sets **a** to 3 and **b** to 4 in the “outer” environment. Then the **let** creates a new environment which sets local copies of **a** to 10 and **b** to 20. Finally, we define a closure **foo.**

Where is **foo** defined? Since **foo** is not listed as a local symbol for the **let**, **foo** is bound in the outer environment. However, the function that is bound to **foo** is defined in (and references symbols **a** and **b** in) the “local” environment of the **let**. So the closure encloses the “inner” environment, and any invocation of **foo** will happen in the “inner” environment, even if we’re no longer “inside” the **let**:

(= a 3 b 4) 🡪 4

(let ( (a 10) (b 20) )

(defun foo (x) (+ a b x)))

;; foo defined within the let

(prl (+ a b)) ;; we’re no longer inside the let

(prl (foo 5)) ;; but foo has enclosed the environment

;; that the let created, and executes there

🡪

7

35

This is more than a syntactic trick: **foo,** or other functions defined in the same environment, can modify that environment at any time:

(= a 3 b 4)

(let ( (a 10) (b 20) )

(defun foo (x) (+ a b x))

(defun bar (x) (= b x)))

(prl "(+ a b) <before> = " (+ a b))

(prl "(foo 5) <before> = " (foo 5))

(bar 1) ;; sets b = 1, but where?

(prl "(+ a b) <after> = " (+ a b)) ;; b wasn’t changed “outside”

(prl "(foo 5) <after> = " (foo 5))

;; b was changed in the “enclosed” environment

🡪

(+ a b) <before> = 7

(foo 5) <before> = 35

(+ a b) <after> = 7

(foo 5) <after> = 16

One important aspect of Yarr programming is recursion. To facilitate this, every Yarr closure binds the symbol **self** to itself in its local environment. This allows us to define anonymous recursive closures:

((fn (x) (if (== x 1) 1 (\* x (self (- x 1))))) 5)

🡪

120

### Parameter types

Yarr functions have support for optional parameters, keyword parameters, and variable number of arguments. These are declared using the **&optional**, **&key,** and **&rest** symbols in the parameter list definitions.

When specifying parameters, it’s also possible to specify that the corresponding argument must be of a specific type, for example:

(defun foo ((a int32) (b int32)) (+ a b))

defines a function *foo* that takes two **System.Int32** arguments. If invoked with arguments that are not **System.Int32**, an exception is thrown. In general, if the type specified for an argument has an appropriate **op\_implicit** methoddefined, then Yarr will try to invoke the **op\_implicit** method to convert the parameter to the correct type and will throw an exception if this fails or if no appropriate **op\_implicit** method exists.

Symbols that have the same name as an existing .NET type will resolve to that type. Recall that to specify an array type, we use the following notation:

int32\* ; Array of Int32

🡪

#<System.Int32[]>

int32\*\* ; Array of Array of Int32

🡪

#<System.Int32[][]>

int32\*2 ; Rank 2 Array of Int32

🡪

#<System.Int32[,]>

int32\*\*2 ; Array of Rank 2 Array of Int32

🡪

#<System.Int32[][,]>

So a function that expects an array of strings as its argument can be defined as:

(defun bar ((s string\*)) ... )

### Optional parameters

Optional parameters are declared using the **&optional** symbol:

(defun foo (a &optional b) …)

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

(defun foo (a &optional b (c 5)) …)

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

It’s also possible to specify a type for optional parameters:

(defun foo (a &optional ((b int32)) ((c int32) 5)) …)

specifies that **b** and **c** must be **System.Int32**. Since no default value is specified for **b**, it will default to zero if not specified.

### Keyword parameters

Keyword parameters are declared using the **&key** symbol:

(defun foo (&key from to) …)

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

(foo :from a :to b)

(foo :to b :from a)

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

(defun foo (&key (from 0) to) …)

If the **:from** keyword is not provided, **from** is set to the default value of 0. Parameter types can also be specified:

(defun foo (&key ((from Int32)) ((to Int32) -1)) …)

This declares the **from** and **to** parameters as integers and sets the value of the **to** parameter to -1 if it’s not supplied.

### Variable arguments

The **&rest** symbol is used to specify that all remaining unbound arguments are to be collected into a list:

(defun foo (a b &rest c) …)

The first two arguments are bound to parameters **a** and **b**; any remaining arguments are collected into a list and bound to parameter **c**.

### Return types

To specify that a Yarr function will return a particular type, we use the special **:returns** keyword after the parameter list:

(fn (a b c) :returns int32 …)

defines a closure that returns a **System.Int32**. This can be used in a **defun** as well:

(defun foo (a b c) :returns int32 …)

The **:returns** and the following type must appear immediately after the parameter definitions. If not provided, the closure is assumed to return **System.Object**.

If the closure attempts to return a value other than the specified return type, an attempt will be made to implicitly cast the value (by looking for a suitable **op\_implicit**), and if this fails, an exception is thrown.

## Lexical and Dynamic Scope

As mentioned previously, Yarr symbols are *lexically scoped* by default. However, it is sometimes useful to have symbols that are *dynamically scoped*. Yarr supports this with the **defparam**special form, which declares that a particular symbol should be dynamically scoped instead of lexically scoped:

(setf lex "outer lexical")

(defparam \*dyn\* "outer dynamic")

(defun show-lex () lex)) ; this lex is the one defined above

(defun show-dyn () \*dyn\*) ; this \*dyn\* is dynamically scoped

(let ((lex "local lexical")

(\*dyn\* "local dynamic"))

(prl (show-lex))

(prl (show-dyn)))

🡪

outer lexical

local dynamic

Since **lex** is lexically scoped, its value in the **show-lex** function is “outer lexical,” since that’s its value in the environment where **show-lex** is defined. But since **\*dyn\*** is dynamically scoped, its value in the **show-dyn** function will be the value of that symbol at the time that **show-dyn** is called, which in the above example is “local dynamic.”

It’s a LISP convention to name dynamically scoped symbols with asterisks before and after the name, as in **\*dyn\*** to distinguish them from lexically scoped symbols. This is optional, there are no restrictions on what dynamically scoped symbols are named.

## Multiple Dispatch Methods

In addition to closures and functions, the Yarr runtime also supports multiple dispatch methods, or *multimethods.* A multimethod is a method that is dispatched at runtime based on the value or data type of one or more if its arguments. This feature is commonly used to call methods on .NET classes, when Yarr uses the argument types to invoke a particular overloaded method in the .NET class. But it is also possible to define Yarr methods that are dynamically dispatched.

In Yarr, multimethods are defined using the **defmethod** special form:

(defmethod *name* (argument-list) … )

The definition is exactly like the definition of a function using **defun**. However, multiple methods can be defined, and the type specifiers in the argument list are used to determine how the method will be dispatched. For example:

(defmethod my-method (a b)

(prl "object a=" a " object b=" b))

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b))

(defun call-my-method (a b) (my-method a b))

(call-my-method 3 4)

(call-my-method "foo" 4)

(call-my-method 3 "bar")

🡪

object a=3 object b=4

string a=foo object b=4

object a=3 string b=bar

We can also dispatch on multiple parameters. Continuing the example:

(defmethod my-method (a b)

(prl "object a=" a " object b=" b))

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b))

(defmethod my-method ((a :int32) (b :string))

(prl "int32 a=" a " string b=" b))

(call-my-method 3 "foo")

(call-my-method "foo" "bar")

🡪

int32 a=3 string b=foo

string a=foo object b=bar

Note the last example, where there is a choice of which **my-method** can be called. We have a method with parameter list: *(a (b :string))* and another with parameter list: *((a :string) b)*, so which should be matched when both are strings? The Yarr runtime resolves this by matching left-to-right, so it matches *((a :string) b)* since that’s the most specific match to the first (left-most) argument. But what if we wanted both methods to be called in this case?

Inside the definition of a method, the symbol **next-method-p** is bound to a function that returns true if there is another matching method that can be invoked on the arguments, and the symbol **call-next-method** is bound to that next method to call.

Let’s redefine the two methods in question to use **call-next-method**:

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b)

(call-next-method))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b)

(call-next-method))

Now we can invoke **my-method** with two strings to see what happens:

(defmethod my-method (a b)

(prl "object a=" a " object b=" b))

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b)

(call-next-method))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b)

(call-next-method))

(call-my-method "foo" "bar")

🡪

string a=foo object b=bar

object a=foo string b=bar

object a=foo object b=bar

Note how using **call-next-method**, each method calls the next one in the chain, and the methods are called from *most specific* to *least specific*. To make this a little bit more clear:

(defmethod my-method (a b)

(prl "object a=" a " object b=" b))

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b)

(call-next-method))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b)

(call-next-method))

(defmethod my-method ((a :string) (b :string))

(prl "string a=" a " string b=" b)

(call-next-method))

(call-my-method "foo" "bar")

🡪

string a=foo string b=bar

string a=foo object b=bar

object a=foo string b=bar

object a=foo object b=bar

### Before and After Methods

Sometimes it’s useful to perform some initialization before a method chain is invoked, or to perform some clean-up afterwards. This can be done by defining methods with the **:before** and **:after** specifiers. Continuing our example above:

(defmethod my-method (a b)

(prl "object a=" a " object b=" b))

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b)

(call-next-method))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b)

(call-next-method))

(defmethod my-method ((a :string) (b :string))

(prl "string a=" a " string b=" b)

(call-next-method))

(defmethod my-method :before (a b)

(prl "Before object-object"))

(defmethod my-method :after (a b)

(prl "After object-object"))

(call-my-method "foo" "bar")

🡪

Before object-object

string a=foo string b=bar

string a=foo object b=bar

object a=foo string b=bar

object a=foo object b=bar

After object-object

Note that there is no **call-next-method** call in the **:before** or **:after** definition. It is implied that the primary method chain will be invoked after **:before**, and that **:after** will be invoked after the last primary method (note that there is no **call-next-method** in the default **my-method** definition, either).

As with primary methods, the **:before** and **:after** methods can be chained, but the chaining happens automatically without having to call **call-next-method**:

(defmethod my-method (a b)

(prl "object a=" a " object b=" b))

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b)

(call-next-method))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b)

(call-next-method))

(defmethod my-method ((a :string) (b :string))

(prl "string a=" a " string b=" b)

(call-next-method))

(defmethod my-method :before (a b)

(prl "Before object-object"))

(defmethod my-method :after (a b)

(prl "After object-object"))

(defmethod my-method :before ((a :string) (b :string))

(prl "Before string-string"))

(defmethod my-method :after ((a :string) (b :string))

(prl "After string-string"))

(call-my-method "foo" "bar")

🡪

Before string-string

Before object-object

string a=foo string b=bar

string a=foo object b=bar

object a=foo string b=bar

object a=foo object b=bar

After object-object

After string-string

The **:before** methods are invoked, like primary methods, from most specific to least specific. However, the **:after** methods are invoked in the *reverse order*, from least-specific to most-specific.

### Around Methods

An **:around** method is invoked before any **:before** method. However, if an **:around** method is invoked, the rest of the method chain is *not* invoked unless the **:around** method explicitly invokes **call-next-method**:

(defmethod my-method (a b)

(prl "object a=" a " object b=" b))

(defmethod my-method (a (b :string))

(prl "object a=" a " string b=" b)

(call-next-method))

(defmethod my-method ((a :string) b)

(prl "string a=" a " object b=" b)

(call-next-method))

(defmethod my-method ((a :string) (b :string))

(prl "string a=" a " string b=" b)

(call-next-method))

(defmethod my-method :before (a b)

(prl "Before object-object"))

(defmethod my-method :after (a b)

(prl "After object-object"))

(defmethod my-method :before ((a :string) (b :string))

(prl "Before string-string"))

(defmethod my-method :after ((a :string) (b :string))

(prl "After string-string"))

(defmethod my-method :around ((a :int32) (b :int32))

(prl "Around a=" a " b=" b))

(call-my-method 3 4)

🡪 Around a=3 b=4

;; redefine the last defmethod to invoke call-next-method:

(defmethod my-method :around ((a :int32) (b :int32))

(prl "Around a=" a " b=" b)

(call-next-method))

(call-my-method 3 4)

🡪

Around a=3 b=4

Before object-object

object a=3 object b=4

After object-object

As with other methods, **:around** methods can be chained, from more specific to less specific (just like primary methods), in which case **call-next-method** will call the next **:around** methodin the chain. If there are no more **:around** methods, **call-next-method** will invoke the primary method chain starting with the most specific **:before** method.

EQL Dispatch

In addition to dispatching on the *type* of a parameter, it is also possible to dispatch on the *value* of a parameter. Dispatching on a value is considered more specific than dispatching on a parameter type:

;; include all defmethods from previous example here

(defmethod my-method ((a (eql 42)) b)

(prl "a is 42, b=" b))

(call-my-method 42 "bar")

🡪

Before object-object

a is 42, b=bar

After object-object

;; redefine the last defmethod to invoke call-next-method

(defmethod my-method ((a (eql 42)) (b :int32))

(prl "a is 42, b=" b)

(call-next-method))

(call-my-method 42 "bar")

🡪

Before object-object

a is 42, b=bar

object a=42 string b=bar

object a=42 object b=bar

After object-object

In addition to primary methods, the **:before,** **:after,** and **:around** methods can also be dispatched by parameter value. The reader is invited to experiment.

## Extension Methods

The Yarr runtime knows about extension methods, allowing us to use them seamlessly:

(= foo [1 2 3 4 5])

(prl "sum = " (foo.sum) " avg = " (average foo))

🡪

sum = 15 avg = 3

The Yarr runtime will automatically resolve extension methods, and in particular will resolve LINQ calls. Many LINQ methods accept **Func**, **Action**, and **Predicate** delegates, and it’s often useful to define these in-line as part of the LINQ method call (this is also true of other, non-LINQ methods as well).

Yarr provides built-in **make-func**, **make-action**, and **make-predicate** that can be used to directly construct **Func**, **Action**, and **Predicate** delegates, without having to first explicitly create a closure. For existing functions, there are **action**, **func**, and **predicate** properties that return the corresponding delegate.

For example:

(= foo [1 2 3 4 5])

(system.array.FindAll foo

(make-predicate ((x int32)) (> x 3)))

🡪

[4 5]

(defun bar ((x int32)) (if (> x 3) (pr x)))

(system.array.foreach [1 2 3 4 5] bar.action)

;; bar.action provides an Action<T> delegate

🡪

4 5

An example with LINQ:

(= foo [1 2 3 4 5 6 7 8])

(ToArray (foo.Where ;; LINQ Where(…).ToArray()

(make-func ((x int32))

:returns Boolean

(== (mod x 2) 1))))

🡪 [1 3 5 7]

## Flow control

### Do, Let, Let\*

Yarr has several special forms for flow control. We have already seen one, **let**¸ which creates lexical environments. A simpler one is **do**, which executes each statement in its list, and returns the value of the last one executed:

(do

*Statement1*

*Statement2*

*…*

*StatementN*)

This is the Yarr analogue to putting C# statements between { …}.

The **let** special form is just a **do** with a local environment. There are actually two versions of **let**, the other being **let\*** (pronounced *let-star*). The difference is in how the local symbols are initialized. **Let** initializes them in the outer environment, while **let\*** initializes them in order, in the inner environment:

(= a 10)

(let ((a 2) (b (+ a 1))) ;; sets b = 11, “a” is the outer “a”

(pr (+ a b)))

(let\* ((a 2) (b (+ a 1))) ;; sets b = 3, “a” is the inner “a”

(pr (+ a b)))

🡪

13 5

### Cond, If, When

The **if** special form handles conditionals. Its syntax is:

(if *test1 statement1*

*test2 statement2*

*…*

*testN statementN*

*else-statement*)

the **if** special form checks each *test* in order, starting with *test1* until it finds one that is *true*. Then it evaluates and returns the result of the matching *statement*. The other statements are not evaluated. If there is a lone, unpaired statement at the end of the list, it is considered the “else” statement and is executed if all the tests are *false*:

(= foo 4)

(if

(== foo 1) (prl "foo is 1")

(== foo 2) (prl "foo is 2")

(prl "foo is neither 1 nor 2"))

🡪

foo is neither 1 nor 2

The **if** special form is analogous to nested **if … else if … else** statements in C#. For historical reasons, **cond** can be used as a synonym for **if**, although the syntax is different from the **cond** syntax in Common LISP.

The various *test* forms do not have to explicitly return a **Boolean**. Yarr will convert other data types to **Boolean** as follows:

* Empty strings, numeric zero values (in general, the default value of a value type), and any **null** value is considered to be **false**
* Any other value is considered to be **true**

This automatic conversion to **Boolean** occurs wherever Yarr expects a conditional value, such as the special forms described below.

If the intent is to just execute a collection of statements if a given condition is true, we can use the **when** special form:

(when *condition* *statements*)

If the condition is true, the statements are evaluated in order. If the condition is false, they are not. The result is the value of the last statement evaluated.

### Equality and Identity

In Yarr, a distinction is made between *equality* and *identity*:

* Identity – Checks for *reference* equality: two objects are equal if they are value types with the same value, or if they are the same object
* Equality – Checks for *value* equality: two objects are equal if they are value types with the same value, or if they are object instances and **Object.Equals** returns **true**

We test for identity with the **eq** function, and for equality with the **==** function. Note that **eql** is a synonym for **==** and (mostly) behaves as the c# “==” operator. The **eq** function is more nuanced:

;; value types

(prl (eq 1 1)) ;; same value type, same value

(prl (eq 1 1.0)) ;; different value types, but same value

;; reference types

(= str1 "a string" str2 "a string")

sb1 (new :StringBuilder "abc")

sb2 (new :StringBuilder "abc"))

(prl (eq str1 str2)) ;; Yarr treats strings as value types

(prl (eq sb1 sb2)) ;; different StringBuilder instances

🡪

True

True

True

False

### And, or

The special forms **and** and **or** provide short-circuit evaluation of their statements:

(and

*Statement1*

*Statement2*

*…*

*StatementN*)

The statements are evaluated in order, until one is found that evaluates to *false*, at which time **and** returns *false***.** If all the statements evaluate to *true***,** then **and** returns the value of the final statement.

(or

*Statement1*

*Statement2*

*…*

*StatementN*)

Similarly, the statements are evaluated in order until one is found that evaluates to *true***,** at which time **or** returns its value. If all of the statements evaluate to *false*,then **or** returns *false*.

Since Yarr converts other data types to Boolean (as mentioned above), we can use **and** and **or** to perform many useful checks:

;; using **and** for null-checking

;; returns property value if the reference is not null,

;; otherwise returns null:

(and some-dotnet-reference

Some-dotnet-referece.SubOjbect

some-dotnet-reference.SubOjbect.PropertyName)

;; We can use **or** to generate errors:

(or (and some-dotnet-reference

some-dotnet-reference.SubObject

some-dotnet-reference.SubObject.PropertyName)

;; the throw only executes if **and** evaluates to false:

(throw "could not reference some-dotnet-refrence.SubOjbect"))

These can be used to selectively set values:

(= foo (or (and some-dotnet-reference

some-dotnet-reference.SubObject

some-dotnet-reference.SubObject.PropertyName)

default-value))

This will set *foo* to the property value, but if the .NET reference is null, it will set *foo* to *default-value* instead.

### While

Looping is handled by the **while** special form:

(while *condition statements*)

If the condition is true, the *statements* are evaluated in an implied **do**, and then the condition is checked again. This repeats until the condition is false, at which point while returns the result of the implied **do**.

The **while** special form is analogous to a **while (…) { … }** statement in C#.

### For

The built-in **for** special form has the syntax:

(for *initializer condition iterator statements*)

The *initializer* statement is evaluated only once, before the loop starts. If the condition is **true**, the loop will then execute: the *statements* are evaluated in order, followed by the *iterator.* The loop repeats until the *condition* is **false**. This is analogous to a **for(***initializer* ; *condition* ; *iterator***) { …}** statement in C#:

(for (= x 0) (< x 10) (++ x) (pr x))

🡪

0123456789

This does **not** create a local lexical environment, so in the above example the symbol **x** is bound in the environment where **for** executes.

### To

A simpler version of **for**, when we simply want to increment an indexer, is the **to** special form:

(to *var limit Statements*)

This does creates a local lexical environment where *var* is set to zero, then increments *var* each time through the loop. The loop exits when *var == limit*.

(to x 10 (pr x))

🡪

0123456789

### Foreach

If we have an instance *instance* of a .NET class that implements **IEnumerable**, we can use the **foreach** special form to iterate over the elements:

(foreach *obj instance Statements*)

This creates a local lexical environment, binds *obj* in that environment to each element in turn, and executes the *statements*. This is analogous to a **foreach (***obj* **in** *instance***) { …}**  statement in C#.

### Case

The **case** special form is analogous to a switch statement in C#:

(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.

For example:

(= big-10 [penn-state michigan

michigan-sate ohio-state

purdue indiana

illinois northwestern

wisconsin minnesota

nebraska iowa

rutgers maryland])

(defun classify-school (school)

(pr school " : ")

(case school

[usc ucla washington oregon]

(prl "A new Big 10 school")

big-10 (prl "An original Big 10 school")

notre-dame (prl "Not a Big 10 school")

(prl "some other school")))

(classify-school USC)

(classify-school notre-dame)

(classify-school iowa)

(classify-school Stanford)

🡪

USC : A new Big 10 school

notre-dame : Not a Big 10 school

iowa : An original Big 10 school

Stanford : some other school

### Try, Throw

The **throw** function throws an exception of type **Yarr.YarrException:**

(throw "message for the exception")

This is the only kind of **System.Exception** that makes it past the top-level evaluator. If the evaluator sees an unhandled **System.Exception** of some other type, the exception is wrapped in a **YarrException** and re-thrown. Furthermore, any exception that reaches the top-level evaluator is bound to the symbol **\*last-exception\*** in the top-level environment, so that it may be inspected by the caller.

The **try** special form is used for exception handling within Yarr. The syntax is:

(try *statement*

*Catch-statement*

*Finally-statement*)

First, *statement* is evaluated. If an exception is thrown during its execution, the exception is caught, bound to the symbol **it** in the **try**’s local environment, and then *catch-statement* is evaluated, and its value returned. Regardless of whether or not an exception is caught, the *finally-statement* is evaluated. If there was no exception, the value of *statement* is returned. The *finally-statement* is optional.

(try (pr (/ 10 0)) (pr "exception caught-") (prl "-finally"))

🡪

exception caught--finally

(try (pr (/ 10 2)) (pr "exception caught-") (prl "-finally"))

🡪

5-finally

## Map and Reduce

Yarr can map functions to sequences of parameters. For example, suppose we want to add two lists of numbers, say (1 2 3) and (10 20 30), and we would like to produce (11 22 23) as the result:

(map + '(1 2 3) '(10 20 30))

🡪

(11 22 33)

The **map** function repeatedly applies the **+** function in turn, first executing (+ 1 10), then (+ 2 20), and finally (+ 3 30), and collects the results into a list for us. The **map** function can work with any number of parameters, for example:

(map + '(1 2 3) '(10 20 30) '(100 200 300))

🡪

(111 222 333)

The **map** function also works with anonymous closures:

(map (fn (x y) (+ (\* x x) (\* y y)))

'(1 2 3) '(4 5 6))

🡪

(17 29 45)

In general, **map** takes a function and collection of sequences. There should be one sequence for each parameter required by the function. The function is applied in turn to each collection of parameters, and the results are captured in a list. This works with any **IEnumeable** sequence, for example, to operate on arrays:

(map (fn (x y) (+ (\* x x) (\* y y)))

#(1 2 3) #(4 5 6))

🡪

(17 29 45)

Note that the result is always a list.

It’s also possible to use a method from a static .NET class as the function to map. To do this, the **function** special form is used, which wraps the .NET static method in a Yarr closure so that **map** can invoke it:

(map (function math.sin) #(0 0.5 1.5 2.0))

🡪

(0 0.479425538604203 0.997494986604054 0.909297426825682)

The reference to **(function math.sin)** can be abbreviated as **#'math.sin**:

(map #'math.sin #(0 0.5 1.5 2.0))

🡪

(0 0.479425538604203 0.997494986604054 0.909297426825682)

Yarr provides several mapping functions:

### Map, Mapcar

The **mapcar** function is a synonym for **map**. It is called **map*car*** because it maps a function to successive *car’s* of sequences.

### Mapc

Operates like the **map/mapcar** function, but does not collect the results in a list. Instead, it simply returns the first argument sequence:

(map (fn (x) (\* x x)) #(1 2 3))

🡪

(1 4 9)

(mapc (fn (x) (\* x x)) #(1 2 3))

🡪

#(1 2 3)

### MapList

This maps the function to successive *cdr’s* and collects the results. For example:

(maplist (fn (x) (cons 'foo x)) '(a b c d))

🡪

((foo a b c d) (foo b c d) (foo c d) (foo d))

### Mapl

Operates like **maplist**, but does not collect the results. Instead it simply returns the first argument sequence.

### Reduce

The mapping functions are often used in conjunction with the **reduce** function. The **reduce** function applies parameters pair-wise to the function that is supplied to it. For example:

(reduce + '(1 2 3 4))

🡪

10

(reduce \* '(1 2 3 4))

🡪

24

(reduce #'string.concat '("a" "b" "c" "d"))

🡪

"abcd"

In addition to a function and a sequence, **reduce** accepts the following keyword parameters:

|  |  |
| --- | --- |
| :start | the index to start operating on the sequence |
| :end | the index to stop operating on the sequence |
| :from-end | if true, processes the sequence “right-to-left” instead of “left-to-right” |
| :initial-value | An initial value to use for the reduction |
| :key | a function that will be applied to each element in the sequence prior to reduction |

Some examples:

(reduce + '(1 2 3 4) :start 1 :end 2)

🡪 5

(reduce – '(1 2 3 4)) ;; == (- (- (- 1 2) 3) 4)

🡪 -8

(reduce – '(1 2 3 4) :from-end true) ;; == (- 1 (- 2 (- 3 4)))

🡪 -2

(reduce + '((1 a) (2 b) (3 c)) :key car)

🡪 6

(reduce append '((1) (2)) :initial-value '(i n i t))

🡪 (i n i t 1 2)

(reduce append '((1) (2)) :initial-value '(i n i t)

:from-end true)

🡪 (1 2 i n i t)

## Types

Yarr is a strongly-typed language. The types supported by Yarr are those supported by the .NET runtime. Although symbols and variables are not explicitly declared to be of a certain type, the runtime does track their data types. The function **typeof** can be used to determine the type of a Yarr object:

(= foo 3)

(typeof foo)

🡪 #<System.Int32>

(= foo 12345678910111213)

(typeof foo)

🡪 #<System.Int64>

(= foo 123.45m)

(typeof foo)

🡪 #<System.Decimal>

(= foo 123.10)

(typeof foo)

🡪 #<System.Double>

Note that we can bind values of any type to any symbol. The function **the** is used to convert values from one type to another:

(the *type value*)

Yarr supports a fairly extensive set of type conversions. If it makes sense to convert one type to another, the function **the** can usually figure it out:

(= foo (new Collections.Hashtable))

(= (elt foo "key1") "value1")

(= (elt foo "key2") "value2")

(the Yarr.Cons foo)

🡪

(("key1" . "value1") ("key2" . "value2"))

### Structures

Although Yarr cannot create .NET classes, it does provide a way to encapsulate data. This is done by defining *structures*. A structure is defined using the **defstruct** special form:

(defstruct *struct-name* (*parent-list*) *slot-name1 …*)

This defines a structure named *struct-name*, with *slots* specified by the *slot-names*. A *slot* is analogous to a property on a C# class, but is untyped. After a structure is defined, we can create instances of that structure using the **new** function:

(defstruct person () first-name last-name)

(= a-person (new 'person :first-name "Bob" :last-name "Smith")

(typeof a-person)

🡪 #<Struct person>

Equivalently, we can use the **#S(…)** syntax to define structures:

(defstruct person () first-name last-name)

(= footballer #S(person :first-name "Juninho"))

🡪

#S(person :first-name "Juninho")

(defstruct person () first-name last-name)

(= manager #S(person :last-name "Mourinho" :first-name "Jose"))

🡪

#S(person :first-name "Jose" :last-name "Mourinho")

Note that the order in which the slot values are provided does not matter. Also, not all slots need to be set. It’s also possible to specify default values for the slots in the **defstruct**. Continuing our example from above:

(defstruct person () first-name last-name (is-alive false))

(= abe #S(person :first-name "Abe" :last-name "Vigoda")

🡪

#S(:first-name "Abe" :last-name "Vigoda" :is-alive false)

Once we have an instance, we can use the **elt** function to get or set its slot values:

(defstruct person () first-name last-name)

(= a-person #S(person :first-name "Bob" :last-name "Smith")

(elt a-person :last-name)

🡪

"Smith"

(defstruct person () first-name last-name)

(= a-person #S(person :first-name "Bob" :last-name "Smith")

(= (elt a-person :is-alive) false) ;; RIP Bob ...

(prl a-person)

🡪

#S(person :first-name "Bob" :last-name "Smith" :is-alive False)

Unlike .NET classes, Yarr structures allow *multiple inheritance*, which means that a Yarr structure can have one or more parent structures. The *parent-list* in the structure definition identifies a structure’s parents. Let’s define some simple structures to illustrate this:

(defstruct animal () name leg-count (can-talk false))

(defstruct mammal (animal) (has-hair true) (leg-count 4))

(defstruct bird (animal) (has-feathers true) (leg-count 2))

(defstruct insect (animal) (leg-count 6))

(defstruct flyer () (has-wings true) wing-count)

(defstruct parrot (bird flyer) (can-talk true) (wing-count 2))

(defstruct fruit-bat (mammal flyer) (wing-count 2))

(defstruct dragon-fly (insect flyer) (wing-count 4))

Note how child structures can redefine the default values of their parent. The slots of a child will be the union of all the parent slots and the child’s defined slots. If a slot has more than one default value, then the child’s default value takes precedence. If the child does not redefine a slot’s default value, but it exists in more than one parent, then the definition from the left-most parent in *parent-list* is used:

;; make sure the above definitions remain at the top of

;; the script code window!

(= flipper #S(mammal :name "Flipper"

:leg-count 0

:has-hair false))

🡪

#S(mammal :name "Flipper" :leg-count 0

:has-hair false :can-talk False)

(= tweetie #S(bird :name "Tweetie Bird"))

🡪

#S(bird :has-feathers True :leg-count 2

:name "Tweetie Bird" :can-talk False)

(= puddy-tat #S(mammal :name "Sylvester Cat"))

🡪

#S(mammal :has-hair True :leg-count 4

:name "Sylvester Cat" :can-talk False)

(= polly #S(parrot :name "Polly Parrot"))

🡪

#S(parrot :can-talk True :wing-count 2 :has-feathers True

:name "Polly Parrot" :has-wings True)

(= veronica #s(dragon-fly :name "Veronica Dultry"))

🡪

#S(dragon-fly :wing-count 4 :leg-count 6

:name "Veronica Dultry" :can-talk False

:has-wings True)

(= bruce #S(fruit-bat :name "Bruce Wayne" :leg-count 2))

🡪

#S(fruit-bat :wing-count 2 :has-hair true :leg-count 2

:name "Bruce Wayne" :can-talk False

:has-wings True)

Yarr structures can be used anywhere that .NET classes can be used. In particular they can be used to dispatch multi-methods:

;; continuing from previous script example, those definitions

;; should remain in the script code window

(defmethod fly (x) (prl (elt x :name) " cannot fly."))

(defmethod fly ((x flyer)) (prl (elt x :name) " is flying."))

(fly flipper)

🡪

Flipper cannot fly

(fly polly)

🡪

Polly Parrot is flying

Since *flipper* is an instance of a **mammal** and does not have **flyer** as a parent structure, it is dispatched to the default primary method. But *polly* is an instance of a **bird**, which does inherit from **flyer**, and so it is dispatched to the primary method that takes a **flyer** as its first parameter.

### Local declarations

It is possible to explicitly declare symbols to be of a specific type using the **declare** special form:

(declare (foo int32) (bar int32 10))

This declares **foo** and **bar** to be of type **int32**. Any attempt to bind anything other than an **int32** (or a class with an **op\_implicit** cast to **int32**) will result in an error. When a variable is declared, its value is set automatically to **null** for reference types, or to the type’s default value for value types. It is possible to provide a value as part of the declaration, as in the next example.

The **declare** special form operates on the local environment, so if it is executed in the context of a **let**, **let\***, **to**, or inside a function definition, it only applies to symbols in those local environments:

(let ((x 1) (y 2))

(declare (z int32) 10) ;; a “local variable” for the let

(+ x y z))

🡪 13

(let ((x 1) (y 2))

(declare (z int32) 10) ;; a “local variable” for the let

(+ x y z))

(= z "a string") ;; z declared outside of let,

;; so no error results

🡪

"a string"

In this example, **z** could have been declared along with **x** and **y**:

(let ( (x 1) (y 2) ((z int32) 10) ) ;; “declare” assumed for z

(+ x y z))

🡪 3

## Symbol Packages

Up to this point, all the symbols that we have used are globally unique, for example, all references to a symbol named **foo** refer to the same instance of a **Yarr.Symbol** with “**foo**” as the value of its **Name** property. This means that we can only bind one value to a symbol.

Sometimes it is convenient for a symbol to have different meanings based on the context in which we’re using it. For example, if we define a function and bind it the symbol **string**, then we can no longer use the symbol **string** to refer the type **System.String**. It also means that if there are multiple developers developing Yarr scripts, they have to be careful to use unique symbols in their scripts.

Yarr *packages,* loosely based on Common LISP packages, provide a way around these limitations. A Yarr package is a collection of symbols that are unique *in that package*. A symbol **foo** in one package can be a different **Yarr.Symbol** instance from the symbol **foo** in another package.

Packages are created with the **make-package** function. Once a package is created, symbols can be defined in that package by prefixing the symbol name with the package name. For example:

(make-package "alice")

(defun alice:list (a b) (list alice a b))

(in-package "alice")

(list 3 4)

🡪

(user:alice 3 4)

Let’s examine this line-by-line to see what’s going on. The first statement:

(make-package "alice")

Creates a new symbol package named **alice**. This allows us to define symbols in that package.

Next we define a function, and bind it to the symbol **list** from the **alice** package:

(defun alice:list (a b) (list alice a b))

Only the symbol **list** is defined in the **alice** package. As we will see, the other symbols in this statement will come from other packages.

The next statement sets the *current package* to be the **alice** package:

(in-package "alice")

The current package is the package where symbols will be defined by default, if they are not prefixed with a package name. Now we invoke the function bound to **alice:list**:

(list 3 4)

Note that since we set the current package to **alice**, we can just refer to **list**. Note what the function returned:

(user:alice 3 4)

When we initially defined **alice:list**, the **alice** package was not the current package. At startup, Yarr sets the current package to the built-in **user** package. Therefore, all symbols that are not prefixed by a package name will be defined in the **user** package. So the symbol **alice** in the function definition was created there.

Continuing our example, suppose we want to redefine **alice:list** so that all of its symbols will be created in the **alice** package:

(defun list (a b) (list alice a b))

🡪 ERROR!!!

Since the current package is the **alice** package, Yarr will try to find a function bound to **alice:defun**. But there isn’t anything bound to **alice:defun** yet! It turns out that all of Yarr’s built-in functions are bound to symbols in the built-in **system** package. Therefore, if the current package is **alice**, we must prefix the names of symbols in other packages:

(system:defun list (a b) (system:list alice a b))

(list 3 4)

🡪 (alice 3 4)

It would be nice if we could use all of the various built-in functions bound to symbols in the **system** package without having to constantly use the **system:** prefix in our Yarr scripts. This is accomplished with the **use-package** function. Continuing with our example:

(make-package "alice")

(in-package "alice")

(system:use-package "system") ;; current package is now “alice”

;; so prefix “use-package” with

;; “system”

(cons alice '(bob))

🡪

(alice bob)

Since the symbol **alice:cons** isn’t defined yet—we have not referenced it in our example up to now—Yarr will look in the **system** package for a symbol named **cons**, and finds the one with the **cons** function bound to it. The **alice** package has *inherited* the symbols from the **system** package. A package can inherit the symbols from any number of other packages.

Note that this inheritance happens even if we explicitly ask for **alice:cons**:

(make-package "alice")

(in-package "alice")

(system:use-package "system")

(alice:cons alice '(bob))

🡪

(alice bob)

We can verify this using the symbol’s **FullName** property:

(make-package "alice")

(in-package "alice")

(system:use-package "system")

(alice:cons alice '(bob))

(FullName 'alice:cons)

🡪

"system:cons"

This is not the case if we define a symbol before we invoke **use-package:** the **alice:list** symbol, because it was created before we invoked **use-package**:

(make-package "alice")

;; define alice:list before we use-package system

(defun alice:list (a b) (list alice a b))

(in-package "alice")

(system:use-package "system")

(alice:cons alice '(bob))

(FullName 'alice:list)

🡪

"alice:list"

What if we want to bind a value to **alice:cons**? Since **alice:cons** is really **system:cons**, binding a value to **alice:cons** will actually replace the current binding of **system:cons**. The **shadow** function is used to define an explicit **alice:cons** that is distinct from **system:cons**:

(make-package "alice")

(in-package "alice")

(system:use-package "system")

(shadow "cons")

(FullName 'alice:cons)

🡪

"alice:cons"

We can now bind a value to **alice:cons** and also reference **system:cons**:

(make-package "alice")

(in-package "alice")

(system:use-package "system")

(shadow "cons")

(= cons "some value")

(prl cons)

(prl (system:cons alice '(bob)))

🡪

Some value

(alice bob)

If we know what packages we want to inherit, and which symbols we want to shadow, we can specify those in the initial **make-package** call, so for example we could have specified the following at the start:

(make-package "alice" :use '("system") :shadow '("list" "cons"))

This creates a package named **alice** that inherits the symbols from the **system** package, and creates explicit **alice:list** and **alice:cons** symbols.

### The keyword package

It is possible to specify a symbol with an “empty” package name. Examples of these are the symbols used to tag *keyword parameters* in function calls. Such symbols are called *keywords*, and the package they belong to—which has an empty string as its name—is called the *keyword package*. In Yarr, keywords are special in that Yarr will not permit values to be bound to them:

(= :foo "some value")

🡪

Assign: Cannot bind values to keywords

Keywords are useful whenever we want to use a symbol only for their name, and want to ensure that no value is bound to the symbol (so that it will evaluate to itself), and that the symbol will not be created in the current package. In addition to tagging keyword parameters, we can also use keywords to denote types, for example:

(defun concat ((x :cons) (y :cons)) (append x y))

defines a function that takes two **Yarr.Cons** instances as parameters, and appends them. In this example, the symbol **:cons** is guaranteed to resolve to the type **Yarr.Cons** because **:cons** is a keyword, and Yarr will not allow a value to be bound to keywords. This is also useful, for example, when referencing .NET types, as we saw in the section on multi-dispatch methods.

## Property lists

In Yarr, symbols can be assigned *properties*, and each property can have a value. Properties and values are not lexically scoped and affect the symbol globally. The **get** function is used to get and set a symbol’s properties:

;; set foo’s my-prop property to 10

(= (get 'foo 'my-prop) 10)

(let (foo) (= (get 'foo 'my-prop) 20))

;; this set symbol property on foo. It doesn’t

;; matter that foo is in a local environment,

;; the property is on the symbol itself, not

;; the environment binding

(get 'foo 'my-prop)

🡪

20

A symbol can have any number of properties defined. Distinct symbols in different packages will each have their own property list, as they are distinct symbols.

# Meta-Programming

Yarr provides three facilities for extending the language with Yarr code: *macros*, *setf methods*, and *reader macros*.

## Macros

Yarr functions take values as their arguments and return values. By contrast, Yarr macros take Yarr code as their arguments, and return Yarr code, which is then evaluated.

A Yarr macro consists of a *pattern* and a set of arguments. The arguments are then used to *macro-expand* the pattern into Yarr code, which is then evaluated. Macros are defined using the **macro** special form. For example:

(= my-square (macro (x) `(\* ,x ,x)))

This macro takes one argument, bound to x, which is any unevaluated Yarr form. The template to expand is `(\* ,x ,x). The back-quote (`) indicates that macro-expansion will be occurring in the list. Macro-expansion happens wherever the unquote (,) marker appears. The **macroexpand** function can be used to see the results of macro-expansion without actually executing the result:

(= my-square (macro (x) `(\* ,x ,x)))

(macroexpand '(my-square 3))

🡪

(\* 3 3)

(= my-square (macro (x) `(\* ,x ,x)))

(macroexpand '(my-square (foo 2)))

🡪

(\* (foo 2) (foo 2))

Macro-expansion can also “splice” lists into the template. This is done using the splice marker (,@):

(= my-mac-1 (macro (&rest x) `(list ,@x)))

(macroexpand '(my-mac-1 a b c))

🡪

(list a b c)

(= my-mac-2 (macro (&rest x) `(list ,x)))

(macroexpand '(my-mac-2 a b c))

🡪

(list (a b c))

This example should make clear the difference between unquote (,) and splice (,@).

The built-in macro **defmacro** simplifies the syntax for macro definition, and could have been used to define the macros above:

(defmacro my-mac-1 (&rest x) `(list ,@x))

(defmacro my-mac-2 (&rest x) `(list ,x))

Macros, like functions, are a kind of closure, and enclose the environment in which they are defined.

Since macro parameters are usually lists, macro parameter lists provide support for *destructuring*. For example, if we want to create a macro that converts input of the form **(a op b)** to **(op a b)**, we can define it as follows:

(defmacro infix-to-prefix ( (a op b) ) `(,op ,a ,b))

(infix-to-prefix (1 + 2))

🡪

3

Destructuring parameters can also be used to split lists, for example:

(defmacro split ( (list-car . list-cdr) )

`(list ',list-car ',list-cdr))

(split (a b c d))

🡪

(a (b c d))

As slightly more complicated example, here’s a macro that swaps the first two elements of a list:

(defmacro swap-head ( (first-elt second-elt . list-rest) )

`(cons ',second-elt (cons ',first-elt ',list-rest)))

(swap-head (a b c d))

🡪

(b a c d)

We have already seen various built-in macros. In addition to **defmacro, defun, make-array,** and **when** (among others) are all implemented as macros. Try them with **macroexpand** to see what they do behind the curtain.

## Setf methods

Previously, we saw that the assignment function **=/setq/setf** could be used to not only bind symbols, but to also alter structures such as lists or arrays. For example:

(= (aref foo 1 1) 10)

sets the value of an array element.

What happens behind the scenes is that **=/setq/setf** examines its arguments, and notices that 10 needs to be assigned to (aref foo 1 1) so it checks to see if there is a *setf method* bound to **aref**. If there is, the setf method is invoked, and is passed the value 10, the object **foo** and the arguments (1 1).

It’s possible to define custom setf methods using the **defsetf** macro. When a setf method is invoked, it is passed at least two arguments: the first argument is the value that is to be set (10 in our example), the second is the object that is to be affected (**foo** in our example). If there are additional arguments expected (in our example, (1 1)) then the setf method must account for them in its argument list.

As an example, let’s define our own version of **aref** that uses **Array.SetValue** to set an array value:

(defsetf my-aref (value arr &rest index)

(arr.setvalue value (the System.Array index)))

With **my-aref** defined, we can now invoke it:

(defsetf my-aref (value arr &rest index)

(arr.setvalue value (the System.Array index)))

(= foo #2a((1 2)(3 4)))

(= (my-aref foo 1 1) 10)

foo

🡪

#2a((1 2)(3 10))

Note that it is possible (in fact, it is desirable) for setf methods to be bound to the same symbol as regular functions. So, for example, there is a Yarr function **first** that returns the first element in an array, and there is also a setf method **first** that sets the value of the first element in an array.

## Reader macros

Reader macros are functions that can intercept the Yarr input stream. They can then reformat the input before it reaches the parser. There are various built-in Yarr reader macros. For example there is a reader macro that reads '(a b c) and converts it (quote (a b c)), and another that converts input of the form #'foo to (function foo). Also, the #(…) and #*nn*a(…) array syntax is handled by reader macros that convert these to (array …) and (md-array …)respectively.

The Yarr reader expects the reader macro to take three parameters:

(fn (*reader readtable character*) …)

The parameters are:

* *Reader* – the **System.IO.TextReader** that is the current input stream. The reader macro should read from this stream
* *Readtable* – the *readtable* from which the reader macro was dispatched
* *Character* – The character that caused this reader macro to be invoked. This character has already been read and removed from the input stream

The Yarr reader expects a **Yarr.ReadTable** instance to be bound to the symbol **\*readtable\***, and uses that **ReadTable** instance to process the input stream. It looks up each character that it reads from the input stream in the **ReadTable**, and if there is a macro defined for that character, the reader invokes the macro and passes it the input stream, the **ReadTable** currently in use, and the character that caused the macro to be called. When the macro returns, the reader continues processing input stream.

To implement custom reader macros, the **bind-macro-character** macro can be used to bind a Yarr function in **\*readtable\***, for example:

(bind-macro-character #\$ (fn (*reader readtable character*) …))

Binds a reader macro for the **$** character. Note that in Yarr, characters are entered using **#\*char***, so **#\$** specifies the dollar-sign character. As suspected, **#\** invokes a reader macro for reading characters.

Suppose we want to create a reader macro that allowed us to construct symbols that had any embedded character, including white-space. For example, input of the form !a symbol! should result in a symbol named “a symbol” including the space. We want to be able to bind our macro to any character, so if we bind it to **$**, then it should be able to parse $a symbol$. We can implement the reader macro as follows:

(defun symbol-reader ((rdr System.IO.TextReader)

(tbl Yarr.ReadTable)

(chr Int32))

(let ((sb (new System.Text.Stringbuilder))

c)

(while (and (!= (= c (rdr.Read)) chr)

(!= c -1))

(sb.append (the System.Char c)))

(the Yarr.Symbol (sb.ToString))))

(bind-macro-character #\! symbol-reader)

(bind-macro-character #\$ symbol-reader)

(prl (typeof !a symbol!))

(prl (typeof $another symbol$))

🡪

Yarr.Symbol

Yarr.Symbol

### Dispatch reader macros

The dispatch macro, which by default is bound to the pound sign (#), is special: it can invoke other reader macros, and pass them a numeric parameter. This reader macro is called the *dispatch macro*, because it can dispatch to other reader macros. For example, there is an array *dispatch reader macro* bound to the character **a**. So when reader sees the following in the input stream:

#2a((1 2) (3 4))

The reader discards the **#** character and invokes the dispatch macro. The dispatch macro then inspects the input stream for an integer, and collects it. In this example, the integer **2** is collected. Then the dispatch macro sees the **a**, and passes the **2** as a parameter to the *dispatch reader macro* bound to **a**, which in this case is the array reader.

We can write our own dispatch reader macros, and bind these dispatch reader macros by using the **bind-dispatch-character** macro. Dispatch reader macros have a signature similar to reader macros, but contain an extra parameter: the numeric argument collected by the dispatch macro. For example, suppose we want to write a reader macro that takes input of the form #*base*Nxxxx and parses the **xxxx** part as a number in base *base.* This would allow us to enter, for example, hexadecimal numbers such as #16n3ff3:

(defun radix-reader ((rdr System.IO.TextReader)

(tbl Yarr.ReadTable)

(chr Int32)

(radix Int32))

(rdr.read) ; the reader does not discard the character

; that we’re bound to, so get rid of it now

(let ((number 0)

c)

(or (> radix 1) (= radix 16)) ; default to base 16

(while (not (Reader.TerminatingCharP (rdr.peek)))

(= c (Char.ToLower

(the System.Char (rdr.read))))

(or

(and (>= c #\0) (< (- c #\0) radix))

(and (>= c #\a) (< (- c #\a -10) radix))

(throw "Incorrect number format"))

(= number (+ (\* number radix)

(if (and (>= c #\0) (<= c #\9))

(- c #\0)

(- c #\a -10)))))

number))

(bind-dispatch-character #\# #\n radix-reader)

(bind-dispatch-character #\# #\N radix-reader)

(+ #n1f #8N10)

🡪

39

Note that we can bind our dispatch reader macro to any two-character sequence, not just #n and #N. If we wanted to use @n and @N instead, we could have done that with:

(bind-dispatch-character #\@ #\n radix-reader)

(bind-dispatch-character #\@ #\N radix-reader)

In this case, we could *not* also bind a reader macro to **@**, since we have caused it to be bound to the dispatch macro.

# Scripting and Workflows

Scripts are added to CopilotBuilder workflows as workflow steps. When a script executes, the output of the previous workflow step, or the user’s input if the script is the first workflow step, will be bound to symbols in the script’s environment. The output of the script will be the input to the next workflow step, or the final output of the workflow if the script is the last workflow step.

## Script Input/Output

When running as part of a workflow, the following symbols will be bound in the environment before the script is executed:

|  |  |
| --- | --- |
| **Symbol** | **Binding** |
| \*message\* | Bound to the output string of the previous workflow step. If the script is the first workflow step, this will be bound to the user’s input |
| \*previous-result\* | Bound to an **IEnumerable** that contains the results of previous workflow steps (see below). If the script is the first workflow step, this will be null |
| \*context\* | Bound to the Microsoft Semantic Kernel **ChatHistory** instance that contains the context for the workflow. This will be null unless there are workflow steps sharing their context, and the current script is sharing its environment. See [Creating and managing a chat history object](https://learn.microsoft.com/en-us/semantic-kernel/concepts/ai-services/chat-completion/chat-history?pivots=programming-language-csharp) for more information about **ChatHistory** objects |

These symbols are dynamically scoped. Note that these symbols are not bound when executing/testing the script on the script editing page.

The **\*previous-result\*** symbol is bound to an **IEnumerable** collection of **ChatMessage** instances that contain the results of previous workflow steps. Each **ChatMessage** instance contains the following information:

|  |  |
| --- | --- |
| **Property** | **Contents** |
| AuthorRole | The Microsoft Semantic Kernel **AuthorRole** (enum value) of the author of this **ChatMessage** |
| Author | The name of the author of this **ChatMessage**. This will usually be the name of the Copilot that generated the message |
| Message | The output of the **Author**. This is what will be bound to the **\*message\*** symbol of the next workflow step if that step is a script |
| Citations | If the Message was produced by referencing one or more data collections, this property will contain a .NET **List** of citation instances (see below). Otherwise this **List** will be empty |

If applicable, the **Citations** property will contain a collection of **ChatMessageCitation** instances. These instances contain the following information:

|  |  |
| --- | --- |
| **Property** | **Contents** |
| DocumentId | The ID (GUID) of the document that was referenced. |
| DocumentName | The name of the document that was referenced. |
| CitationSegments | A **List** of segments, each containing a **Text** property with the citation text, and a **Relevance** property (**System.Double)**, indicating the relevance value of the **Text** |

If the script invokes the **pr** and/or **prl** functions during script execution, that output will be aggregated as the output of the script when the script terminates. If the script doesn’t invoke these functions, then the output of the script will be the result of the last Yarr statement executed in the script. This output will added to the **\*previous-results\*** list, and will be the input to the next workflow step (so it will be bound to **\*message\*** if the next step is a script). If the script is the final workflow step, the script output will be the final output of the workflow.

## Sharing Environments

By default, a script is run in its own environment, isolating it from other scripts in the workflow. To run multiple scripts in the same environment, uncheck the “Do not share this script's context with other scripts” checkbox. All scripts with this checkbox unchecked will run in the same environment.

If passing data between scripts by binding objects to symbols in the shared environment, those symbols should be dynamically scoped with **defparam** at the start of the first script in the workflow that uses them.

## Referencing Other Scripts

A script can invoke another script *via* the **load** function. For example, to execute the CopilotBuilder script named “My Common Script”:

(load "My Common Script")

The **load** function causes the specified script to immediately execute in the current script’s environment. This allows creating scripts that act as function libraries, or that execute common code.

Calls to the **load** function can be nested, *i.e.*, a script that is being **load**’ed can itself call the **load** function. However, if such **load** calls are nested more than ten calls deep, an exception will be thrown to prevent inadvertent circular references.