AddyScript Manual

Developer Documentation (version 1.0.1.0, February 2026)

Michel Mbem

Table of Contents

1. Welcome to AddyScript

1.1 Overview

AddyScript is a lightweight scripting engine for the .NET platform. It can be used to add scripting functionality to applications targeting this platform or simply as a learning tool for young people.

AddyScript has a simple syntax, similar to that of C (or JavaScript or PHP, depending on your preference), with dynamic typing and consistent object-oriented programming features. You can familiarize yourself with this syntax by reviewing the example scripts provided with the source code.

This manual aims to guide you through its syntax and features.

2. Introduction

2.1 Features

Here's an overview of AddyScript's main features:

- C-like syntax with notable borrowings from popular scripting languages like JavaScript, PHP, Python, and Ruby.

  Note: AddyScript is, however, more influenced by C# than by any other language.
- Dynamic typing (variables are declared when assigned a value and can change type during script execution).
- Literals for most primitive types (including complex numbers, dates, strings, and blobs).
- Initializers for most composite types (tuples, lists, sets, maps, and objects).
- Functions accept parameters passed to them by reference or as a variadic list of arguments.
- Support for object-oriented programming with encapsulated properties, events, operator overloading, and introspection.
- Closures and lambda expressions.
- Tight integration with the .NET platform.
- Support for interoperability with COM (on Windows).
- Structured handling of runtime errors.
- Ability to import one script from another.
- Ability to pass initial parameters to a script and retrieve them after execution.
- Ability to extend the scripting engine by providing new functions or "built-in" classes.
- The entire scripting engine is exposed as a class library to the .NET platform.
- Availability of a graphical script editor and an interactive console for testing scripts.

2.2 Tutorials

2.2.1 Interpreting a script

The simpler approach

To interpret a script with AddyScript, just proceed as follows:

1. Type the script in an editor (here I suppose that the editor is a text widget named *txtScript*).
2. Save it in a file (or anywhere else) or keep it in memory if you prefer to do so.
3. Then add a reference to *AddyScript.dll* in your project.
4. Create a GUI in your project to allow the user to invoke the scripting engine.
5. In your code file import the *AddyScript* namespace.
6. You could import any additional namespace from the AddyScript assembly depending on what you intend to do.
7. Finally, type a code snippet like the following one in an event handler:

C#

```
var context = new ScriptContext();
context.Bindings["myString"] = "Hello!";
context.Bindings["myFloat"] = 0.9;
ScriptEngine.ExecuteString(txtScript.Text, context);
// Alternatively: ScriptEngine.ExecuteFile(@"path/to/my/script", context);
Console.WriteLine("myString = " + context.Bindings["myString"]);
Console.WriteLine("myFloat = " + context.Bindings["myFloat"]);
```

NOTES:

Don't forget to embed this code in a try-catch structure.

Parsing once, running later

Sometimes you need to parse the script once and run it multiple times later without needing to restart the parsing process. The *ScriptEngine* class provides means to accomplish this in a straightforward manner.

Look at this example:

C#

```
var context = new ScriptContext();
var program = ScriptEngine.ParseString(txtScript.Text);
// Alternatively: var program = ScriptEngine.ParseFile(@"path/to/my/script");

foreach (var item in myArray)
{
    context.Bindings["myValue"] = item;
    ScriptEngine.Execute(program, context);
    Console.WriteLine("myValue = " + context.Bindings["myValue"]);
 }
```

This will typically run the same script multiple times with different values of the "myValue" context variable without needing to parse the source code each time.

Sequentially parsing and running commands

Here is where the non-static version of the *ScriptEngine.Execute* method comes to action. It is used to sequentially interpret commands on an instance of the *ScriptEngine* class. Each command is interpreted in the same context as the previous one. So, the result of a command may affect a following one (for example, previously declared functions can be called by following commands). If a command includes multiple statements, all of these will be interpreted. If the tail of the command is insufficient to be parsed as a full statement, the *ScriptEngine.Execute* method keeps a copy of it and expects the following command to complete the previous one. The *Satisfied* property of the *ScriptEngine* class indicates whether the continuation of a statement is expected or not. The AddyScript Interactive Shell (**asis**) heavily relies on that possibility to work.

Here is an example of how to use the *ScriptEngine.Execute* instance method:

C#

```
const string MAIN_PROMPT = ">>> ";
const string CONTINUATION_PROMPT = "... ";

var context = new ScriptContext();
var engine = new ScriptEngine(context);
string prompt = MAIN_PROMPT;

while (true)
{
    Console.Write(prompt);
    string command = Console.ReadLine();

    if (string.IsNullOrEmpty(command)) continue;
    if (engine.Satisfied && command == "exit") break;

    try
    {
        var result = engine.Execute(command);
        if (result != null) Console.WriteLine(result);
        prompt = engine.Satisfied ? MAIN_PROMPT : CONTINUATION_PROMPT;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        prompt = MAIN_PROMPT;
    }
}
```

Interacting with the scripting engine

The instance version of the *ScriptEngine.Execute* method is itself a way of interacting with the scripting engine. But sometimes, you may need a closer interaction. Suppose for example that you want to call a function you've previously defined in the script from your C# (or VB) code. How to accomplish this? Well, the *ScriptEngine* class provides useful methods that allow such an interaction. The *ScriptEngine.Invoke* method can be used to invoke from user code a function defined in the script. It takes the function's name and an arbitrary sized array of objects as parameters. The arbitrary sized array of objects represents the arguments that will be passed to the function. The *ScriptEngine.Invoke* method returns an instance of the AddyScript.Runtime.DataItems.DataItem class. You can use one of the *AsXXX* properties of the returned object [or its `ConvertTo(Type targetType)` method] to cast it to the desired type. The *ScriptEngine.GetDelegate* method goes further by allowing the user code to retrieve a delegate that could be used to invoke a function defined in the script. It takes the function's name as a parameter. The target delegate type is indicated as a generic type parameter.

Here is an example of how to use both methods:

C#

```
delegate int SumType(int a, int b);

var context = new ScriptContext();
var engine = new ScriptEngine(context);

// We define a 'sum' function prior to any call to Invoke or GetDelegate
engine.Execute("function sum(a, b) => a + b;");

var x = engine.Invoke("sum", 10, 5).AsInt32;
Console.WriteLine($"sum(10, 5) gives: {x}");

var sum = engine.GetDelegate<SumType>("sum");
Console.WriteLine($"sum(7, -3) gives: {sum(7, -3)}");
```

The ScriptContext class

So far you have been reading this tutorial, you may have noticed that on each example, we create and use an instance of a class called *ScriptContext*. Well, this class is simply a mean for providing initial settings to the scripting engine. It exposes three interesting properties that are listed and explained in the following table:

Property	Description
`Dictionary<string, object> Bindings { get; }`	A set of variables bindings: they will be automatically declared in the script and can be retrieved upon script's completion.
`string[] ImportPaths { get; set; }`	The list of directories in which the import directive searches the files to include.
`Assembly[] References { get; set; }`	The list of assemblies in which references to .NET types will be searched for.

Notice that by default, the *ImportPaths* property is empty while the *References* property contains references to some essential .NET assemblies. You don't need to put a dot (symbolizing the current working directory) to the *ImportPaths* property since imported scripts are always first searched in the same directory as the importing script.

The RuntimeServices class

We have not demonstrated the utility of the *AddyScript.Runtime.RuntimeServices* class here. However, it provides services that can be very helpful at runtime. For example, its **In** and **Out** static properties are used by the scripting engine as standard input and (respectively) standard output devices for the currently running script. So if you want to redirect the script's input or output, just assign anything you want to those properties before invoking any *Execute* method on the *ScriptEngine* class.

2.2.2 Evaluating expressions

The simpler approach

To evaluate an expression with AddyScript, do the following:

1. Type the expression into an editor (I'll assume the editor is a text widget named *txtExpr*).
2. Add a reference to *AddyScript.dll* in your project.
3. Create a GUI in your project to allow the user to invoke the scripting engine.
4. In your code-behind file, import the *AddyScript* namespace.
5. You can import any additional namespaces from the AddyScript assembly depending on what you intend to do.
6. Finally, type a code snippet like this into an event handler:

C#

```
var context = new ScriptContext();
context.Bindings["myString"] = "Hello!";
context.Bindings["myFloat"] = 0.9;
var result = ScriptEngine.EvaluateString(txExpr.Text, context);
Console.WriteLine($"Given {context.Bindings["myString"]} and {context.Bindings["myFloat"]}, we obtain {result}");
```

NOTES:

Don't forget to embed this code in a try-catch block.

Parsing once, running later

Below is another example where the expression is parsed once and evaluated multiple times in a loop. Before testing it, you need to create the GUI and provide the logic for the *MoveTo* and *LineTo* methods yourself:

C#

```
// txtFunction is a text widget where the user types the expression in.
// A single parameter named 'x' is expected.
var expression = ScriptEngine.ParseExpression(txtFunction.Text);
var context = new ScriptContext();

// txtFrom, txtTo and txtBy are text widgets where the user types the plotting range in.
double start = double.Parse(txtFrom.Text);
double end = double.Parse(txtTo.Text);
double step = double.Parse(txtBy.Text);

// Determine the initial point and move the graphical cursor there.
double x = start;
context.Bindings["x"] = x;
double y = ScriptEngine.Evaluate(expression, context).AsDouble;
MoveTo(x, y);

// Plot the curve segment by segment.
do
{
    x += step;
    context.Bindings["x"] = x;
    y = ScriptEngine.Evaluate(expression, context).AsDouble;
    LineTo(x, y);
 } while (x < end);
```

2.2.3 More with the ScriptEngine class

Exporting the AST to XML

You can easily export an Abstract Syntax Tree to XML format by invoking the *ExportXml* static method of the *ScriptEngine* class.

Simply do as follows:

C#

```
var program = ScriptEngine.ParseString(txtScript.Text);
// Alternatively: var program = ScriptEngine.ParseFile(@"path/to/my/script");
ScriptEngine.ExportXml(program, @"C:\myScript.xml");
// Alternatively: ScriptEngine.ExportXml(program, someStream);
```

That functionality is specially helpful for debugging purpose. You can use it to ensure that the parsed script has the expected logical structure.

Regenerating the source code

The *ScriptEngine* class has a *GenerateCode* static method than can be used to regenerate the source of a script given its AST (as an instance of the *AddyScript.Ast.Program* class). The generated source code is so nicely formatted than the "Reformat code" functionality of the AddyScript Graphical Editor (**asgui**) fully relies on the *ScriptEngine.GenerateCode* method.

Here is an example of how to use it:

C#

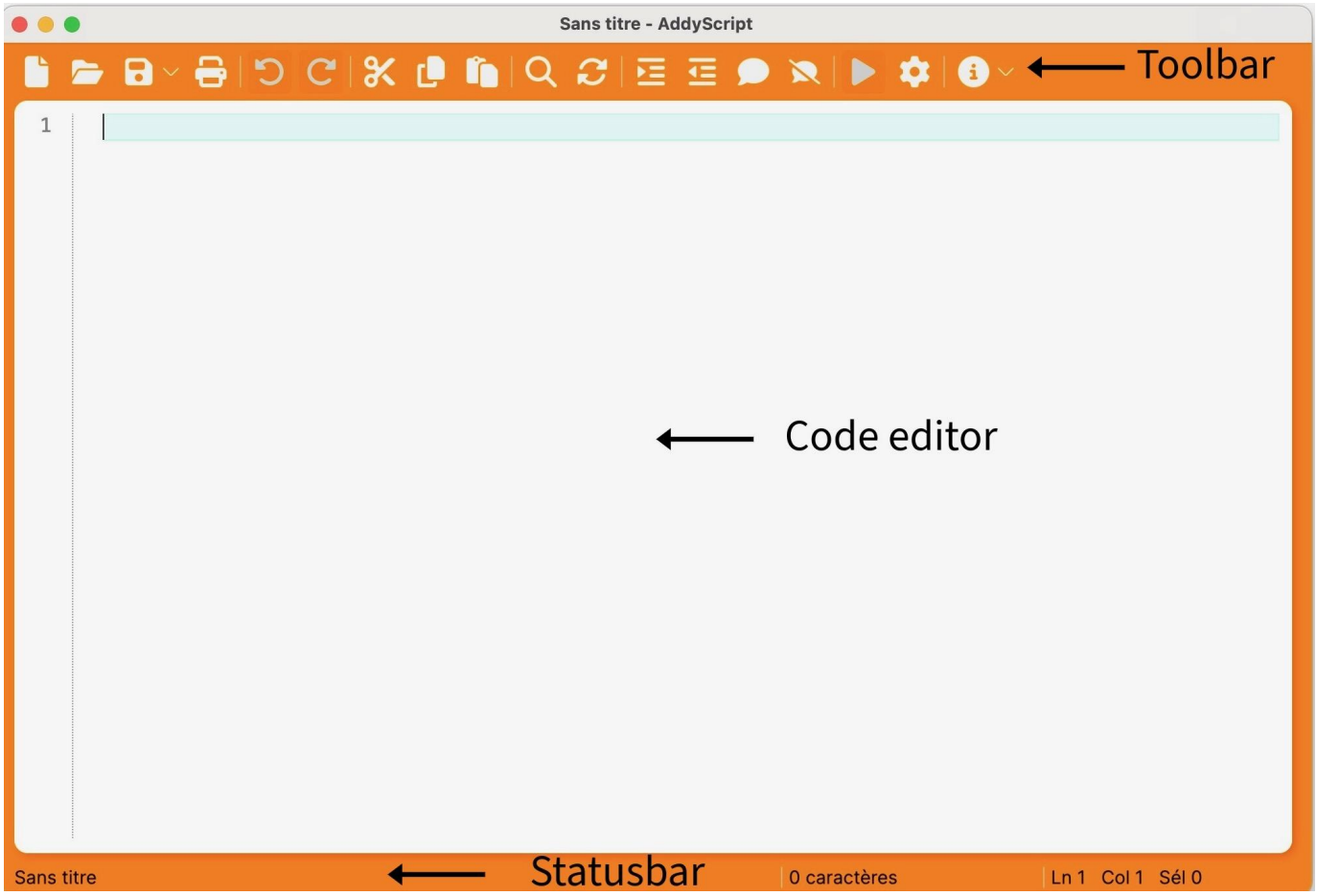
```
var program = ScriptEngine.ParseString(txtScript.Text);
txtScript.Text = ScriptEngine.GenerateCode(program);
```

2.2.4 Using the graphical editor and the command line interface

AddyScript comes with a graphical script editor and an interactive console. These two tools allow you to quickly write and test your own scripts. I will try to explain in the following lines how to use them.

Using the graphical script editor

The AddyScript Graphical User Interface (**asgui**) is an Avalonia UI application built around the AvaloniaEdit control. It offers useful features such as syntax highlighting, automatic indentation, call tips, code completion and code snippets insertion. Its use of the Avalonia UI toolkit ensures compatibility with most operating systems and desktop environments. Upon opening, it appears as a single window with a toolbar at the top, a text area (or code editor) in the middle, and a status bar at the bottom. The following image illustrates the GUI window:



EDITOR TOOLBAR'S BUTTONS:

From left to right, the toolbar buttons play the following role:

1. **New**: Creates a new window with an empty text box to allow you to edit a brand-new script.
2. **Open**: Displays a file selection dialog and allows you to open an existing script in the active window if it was still empty or in a new window if it was not.
3. **Save**: Saves the currently edited script to a file. The associated drop-down menu allows you to save the script under a new name or keep the existing name (if the currently edited script was already saved).
4. **Print**: Prints the currently edited script. This actually exports the script as a PDF document that is then sent to the printer using a system command. There is an option in the associated dropdown menu to save the generated PDF document in a local file for later printing.
5. **Undo**: Undoes the last action.
6. **Redo**: Reapplies the last undone action.
7. **Cut**: Cuts the selected text to the clipboard.
8. **Copy**: Copies the selected text to the clipboard.
9. **Paste**: Pastes the clipboard contents into the editor.
10. **Find**: Displays the Find/Replace dialog with the "Find" tab enabled.
11. **Replace**: Displays the Find/Replace dialog with the "Replace" tab enabled.
12. **Indent**: Increases the indentation level of the selected lines of code.
13. **Outdent**: Decreases the indentation level of the selected lines of code.
14. **Comment Lines**: Comments out the selected lines of code.
15. **Uncomment Lines**: Uncomment the selected lines of code.
16. **Run**: Launches a console window and interprets the script being edited.
17. **Configure**: displays a dialog box in which the user can set options such as the list of directories in which imported scripts should be searched or the list of assemblies referenced by the script.
18. **Help**: displays this manual. Another entry in the drop-down menu displays the about box.

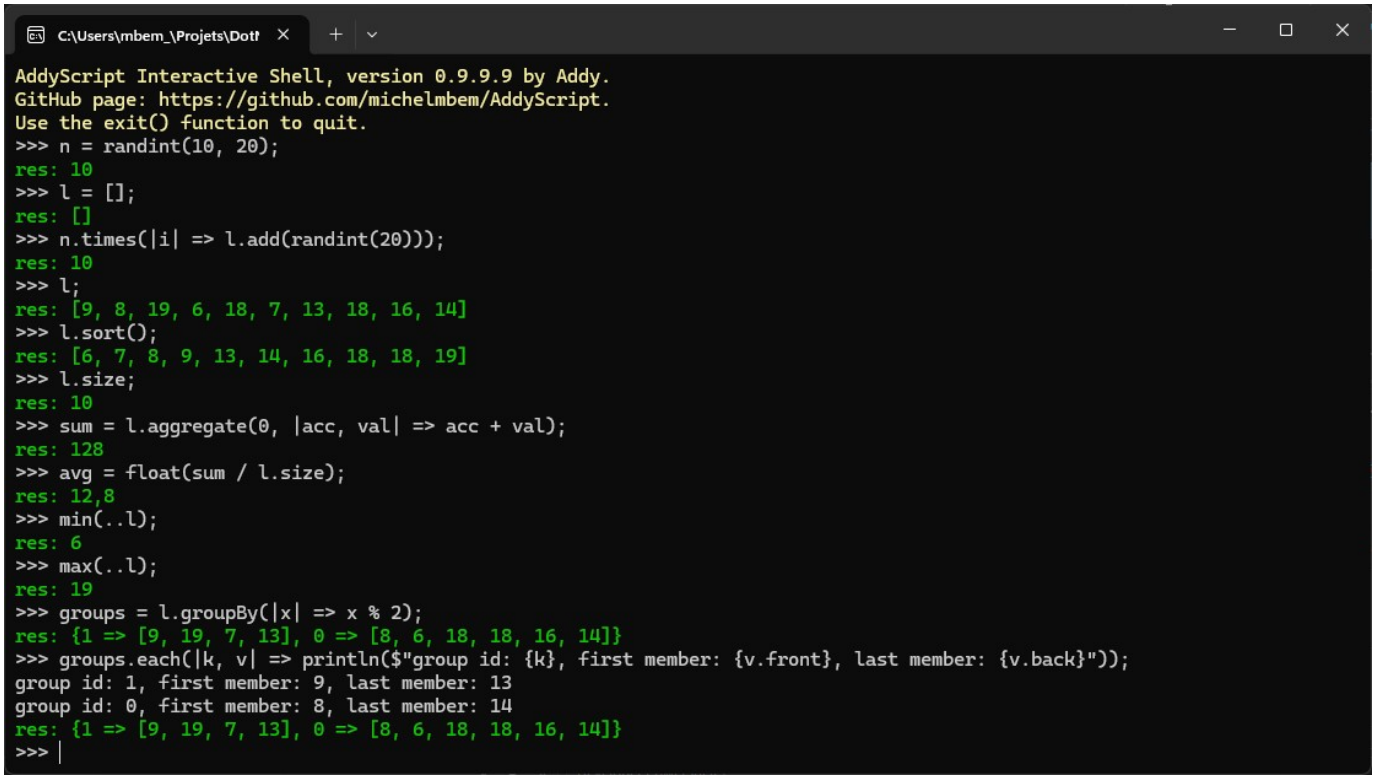
GENERAL EDITOR TIPS:

1. **asgui** does not execute scripts by itself. It delegates this task to **asis**. On Windows and Linux, the executables of both programs must remain in the same directory. On macOS, the application bundle must keep the bundled **asis** helper in its expected location.
2. To have time to read the output printed by your script, always add a call to the *readln()* function at the end of it. Of course this is not necessary for scripts that have a graphical interface.
3. The shortcut to execute a script is **[F5]**. In fact, many commands use the same shortcut as in VS:
    a. **[Ctrl+N]** to create a new script
    b. **[Ctrl+O]** to open an existing script
    c. **[Ctrl+S]** to save the current script (**[Ctrl+Shift+S]** for saving to a different name)
    d. **[Ctrl+P]** to print it (**[Ctrl+Shift+P]** to save as PDF)
    e. **[Ctrl+F]** to open the search window in find mode
    f. **[Ctrl+H]** to open the search window in replace mode
    g. **[F1]** to display help
    h. The **[Tab]** key increases the indentation of selected lines
    i. Combined with the **[Shift]** key it decreases the indentation level instead
    j. **[Ctrl+K]** comments out selected lines
    k. **[Ctrl+Shift+K]** uncomments selected lines
    l. **[Ctrl+I]** inserts a code snippet
    m. **[Ctrl+Shift+I]** surrounds the selection with a code snippet
4. On macOS all the above commands use the **[Command]** key in place of **[Ctrl]**.

5. Additional commands are available in the editor's context menu.

Using the Interactive Console

It may seem natural for a scripting language to have an interactive console. AddyScript adheres to this principle by providing a *REPL* interface called **asis** (for AddyScript Interactive Shell). Now, it's not as sophisticated as those of more mature languages (indeed, AddyScript is under construction, right?) but it does the essentials: read your commands, parse them, and interpret them. Here's an illustration of what it looks like:



Using the console is simple, just type a command and press [Enter]. Remember to add a semicolon at the end of each statement, except for blocks (this may seem strange if you are familiar with scripting languages, but currently AddyScript really expects a semicolon at the end of each statement. This behavior will probably change in the future).

TROUBLESHOOTING WITH THE IF-ELSE STATEMENT:

Suppose you want to enter a sequence of statements like this:

```
a = randint(10);
if (a > 5)
    println('Greater than five');
else
    println('Less than or equal to five');
```

You'll start by typing the first three lines. But as soon as AddyScript encounters the semicolon at the end of the third line, it assumes your if-else statement is complete, throwing an error when it encounters the next **else** (of course, no statement in AddyScript begins with the keyword **else**). To solve this problem, I recommend that you always use curly braces when typing an if-else statement. In particular, be sure to always place your **else** keyword between the closing brace of the **if** block and the opening brace of its own block. If you do this, the previous example would look like this:

```
a = randint(10);
if (a > 5) {
    println('Greater than five');
} else {
    println('Less than or equal to five');
}
```

Another option is to type the whole if-else statement in a single line of code, if it's short enough.

COMMAND LINE OPTIONS FOR ASIS:

Actually, you don't have to run **asis** in interactive mode. Depending on the parameters you pass to it, it can behave differently. Here is a table summarizing the **asis** command line options and their meaning:

Option	Description
-i	Tells that we want to run asis in interactive mode. Not required since that's the default behaviour of asis. This option can figure at most one time in the command line and cannot be mixed with -e and -f.
-e expression	Tells that we just want asis to evaluate the given expression and return after displaying the result to the standard output. This option can figure at most one time in the command line and cannot be mixed with -i and -f. Argument expression may contain several expressions separated by semi-columns. In that case, all expressions are evaluated but only the result of the last one is printed out.
-f path/to/script	Tells that we want asis to parse and run the script stored in the file pointed to by the given path. This option can figure at most one time in the command line and cannot be mixed with -i and -e.
-d directory-name	Tells asis to add the given directory to the SearchPath property of its internal instance of ScriptContext so that imported scripts would be searched in that directory. This option can figure zero, one or several times (with different directory names) in the command line.
-r assembly-name	Tells asis to add the so called assembly to the References property of its internal instance of ScriptContext so that .NET types would be searched in that assembly. This option can figure zero, one or several times (with different assembly names) in the command line.
-l path/to/log/file	Tells asis where to report the errors that may occur during the execution of a script. This option can figure at most one time in the command line and is more useful when combined with the -f option.
-c culture-name	Tells asis to use a specific culture while running a script. 4 cultures are supported at this stage: English (the default), French, Spanish and Portuguese. If not specified, the default UI culture of the current OS user will be used. This option can figure at most one time in the command line and can be combined with any other option.

Here are some useful examples of how to invoke asis:

- Simple interactive mode: `asis`
- Just run the *test5.add* file: `asis -f test5.add`
- Run *test5.add* using the *es* (Spanish) culture and writing logs in *test5.log*: `asis -f test5.add -l test5.log -c es`
- Evaluate an expression: `asis -e "x = 10; y = 5; 2*x - 5*y + 3"`.

Output

```
-2
```

- Interactive mode with reference to some common .NET assemblies; imported scripts should be looked for in the provided *samples/library* directory: `asis -r System.Data.OleDb -r System.Windows.Forms -d ../../../samples/library`.
- Run the *test5.add* file with references to some common .NET assemblies; imported scripts should be looked for in the provided *samples/library* directory: `asis -f test5.add -r System.Data.OleDb -r System.Windows.Forms -d ../../../samples/library`

Conclusion

Well, that's it. More features will be added to both tools in the future. But for now, I hope they do the basics.

3. AddyScript's syntax

3.1 Anatomy of a script

A script in AddyScript is a sequence of statements in any order. Even an empty file is a valid script. Statements come in many different forms: **import** directives, class or function definitions, tests, loops, blocks, assignments, **try-catch-finally** structures, etc.... There is no particular order in which you should organize your statements. For example, you might assign a value to a variable, then declare a class, and then call a function. Some statements are elementary, while others may contain child statements. We will learn about the different types of statements and the proper syntax for each as we progress through this manual. Here are some sample scripts to get you started:

Simply printing *"Hello World!"* to the standard output:

Hello World

```
println('Hello World!');
```

Reading *n* from the standard input and computing the sum and average of n numbers:

Sum and average of n numbers

```
n = (int)readln('How many numbers? ');
sum = 0;

for (i = 0; i < n; ++i)
{
   print($'Item number {i + 1}: ');
   sum += (float)readln();
   // Could also be: sum += (float)readln($'Item number {i + 1}: ');
}

println($'The sum is {sum}');
println($'The average is {sum / n}');
```

Declaring a function to say *"Hello"* to each name in a list:

Hello to each one

```
function hello(name)
{
   println($'Hello {name}');
   if (name === 'roger')
      println('Have you been a football player?');
}

names = ['john', 'mike', 'bill', 'david', 'mark', 'roger'];

// Hello to everyone:
foreach (name in names)
   hello(name);

// Another way to do the same stuff:
names.each(hello);

// Or without declaring the 'hello' function at all:
names.each(function (x) {
    println($'Hello {x}');
    if (x !== 'bill') return;
    println('Have you been a CEO somewhere?');
});
```

Well, now you can try your own.

3.2 Variables, operators and expressions

3.2.1 Variables

AddyScript is a dynamic language. This means that you don't have to declare variables. A variable starts existing as soon as you assign a value to it. This also means that variables are dynamically typed: they simply accept anything you put into them. As far as that aspect is concerned, a single variable can hold an integer at a certain time and hold a list of dates later.

Examples

```
n = 10; // n is an integer
n = 7.5; // now n is of type float
n = now(); // n becomes a date
s = 'I like AddyScript'; // s is a string
t = "Hello y'all!"; // t is also a string
v = ["joe", 45, 18.2, 'house']; // v is a list, items are not necessarily of the same type
o = new {firstName = "James", lastName = "Bond", number = 007}; // o is an object, field names don't have to be known in advance
```

Variable's scope

As mentioned earlier, a variable exists from its initialization until the end of the block in which it is defined. If a variable is created at the root of the script, it will be accessible to the rest of the script. AddyScript does not check the existence of variables accessed from a function, nor does it check the existence of the functions themselves. This means that a function can reference a variable even before the variable is created. As long as the function is not called before the variable is created, this will not generate any errors. However, it is not recommended to do this in functions that you intend to reuse outside their original script, as there is no guarantee that the referenced variable will be defined in the context of the call.

The var keyword

Even though AddyScript allows you to declare variables dynamically, there are situations where it's necessary to declare them explicitly. In particular, to avoid confusion between a local variable and a pre-declared global variable with the same name. This is where the **var** keyword comes in. As in other C-based scripting languages, it's used to explicitly declare variables. Such a variable will hide any constant or variable with the same name declared in global scope.

The syntax for explicitly declaring variables is the following:

```
var <variable_name1> = <initial_value1>, <variable_name2> = <initial_value2>, ... ,<variable_nameN> = <initial_valueN>
```

Setting the initial value of a variable is optional. However, an uninitialized variable remains unusable until it is assigned a value. Therefore, the above syntax is equivalent to the following:

```
var <variable_name1>, <variable_name2>, ... ,<variable_nameN>
<variable_name1> = <initial_value1>;
<variable_name2> = <initial_value2>;
```

```
...
<variable_nameN> = <initial_valueN>;
```

Example

```
// a global variable named toto
toto = 10;

// a function in which toto is declared locally
function foo()
{
    var toto = 15;
    println('in foo, toto = ' + toto);
}

// a function in which toto is not declared locally
function bar()
{
    toto = 20;
    println('in bar, toto = ' + toto);
}

// var in action:
println('in main, toto = ' + toto);
foo();
println('back to main, toto = ' + toto);
bar();
println('back to main, toto = ' + toto);
```

Output

```
in main, toto = 10
in foo, toto = 15
back to main, toto = 10
in bar, toto = 20
back to main, toto = 20
```

3.2.2 Constants

A constant is a read-only variable. That is: it's assigned a value once and cannot be altered afterward. You will typically declare constants using the **const** keyword like in the following example.

The syntax for declaring constants is the following:

```
const <constant_name1> = <constant_value1>, <constant_name2> = <constant_value2>, ... ,<constant_nameN> = <constant_valueN>;
```

Unlike variables, constants cannot be declared without an initial value. Only values of the types **bool**, **int**, **long**, **rational**, **float**, **decimal**, **complex**, **date**, **duration**, and **string** can be used as initial values for constants.

Example

```
// declare a constant named MAX_ITEMS
const MAX_ITEMS = 10;

// try to alter the constant: this will lead to an error
MAX_ITEMS = 100;
```

The following constants are predefined in AddyScript:

Constant	Value	Description
MININT	-2.147,483,648	The minimum value for the **int** type.
MAXINT	+2.147,483,647	The maximum value for the **int** type.
MINFLOAT	-1.79,769,313,486,232E+308	The minimum value for the **float** type.
MAXFLOAT	+1.79,769,313,486,232E+308	The maximum value for the **float** type.
PI	3.14,159,265,358,979	A numeric value used in geometry.
E	2.71,828,182,845,905	Exponential one.
EPSILON	4.94,065,645,841,247E-324 on my developer's machine	A value that varies from machine to machine and indicates which precision to use in arithmetic operations on floating-point numbers.
NINFINITY	(None)	A symbolic representation of the negative infinity.
PINFINITY	(None)	A symbolic representation of the positive infinity.
NAN	(None)	A value indicating that a floating-point number is in an invalid state.
MINDATE	0001-01-01 00:00:00	The minimum value for the **date** type.
MAXDATE	9999-12-31 23:59:59	The maximum value for the **date** type.
NEWLINE	"\r\n" on Windows, "\n" on Unix based systems	The sequence of characters used to mark the end of a line by the underlying platform.

3.2.3 Data types

Even if you don't have to explicitly define the type of your variables in AddyScript, they still have a type. In fact, AddyScript recognizes a set of 30 predefined data types. In addition to that, you can create your own classes and add them to the set of existing data types. Below are listed the AddyScript's built-in types and their meaning:

Type	Description	.NET equivalent
void	The type of **null** or any undefined symbol	System.Void
bool	A boolean: **true** or **false**	System.Boolean
int	A 32-bits signed integer: ranging from -2147483648 to 2147483647. Operations on this type may produce a result of the **long** type as soon as an overflow occurs. Dividing two **int**s produces a result of the **rational** type when the dividend is not divisible by the divisor.	System.Int32
long	A limitless-precision signed integer.	System.Numerics.BigInteger
float	A double precision floating-point number.	System.Double
decimal	A limitless-precision decimal number. The scale is however limited to 50 decimal digits.	System.Decimal (but more like Java's BigDecimal)
rational	A rational number (i.e. a fraction).	(None)
complex	A double precision complex number.	System.Numerics.Complex
date	A point in time (an instant). Basically a simple date or a date-and-time combination.	System.DateTime
duration	The time elapsed between two instants (i.e. an interval of dates). Can also be used to represent a time in a day.	System.TimeSpan
string	An immutable sequence of unicode characters.	System.String
blob	An abstraction of a byte array.	System.Byte[]
tuple	An immutable sequence of data items accessible by index in read-only mode	System.Tuple<T>, System.ValueTuple<T>
list	A dynamically sized sequence of data items accessible by index in read and write mode	System.Collections.ArrayList, System.Collections.Generic.List<T>
set	An emulation of the mathematical concept of a set.	System.Collections.Generic.HashSet<T>
queue	A *first-in-first-out* type of collection.	System.Collections.Queue, System.Collections.Generic.Queue<T>
stack	A *last-in-first-out* type of collection.	System.Collections.Stack, System.Collections.Generic.Stack<T>
map	A set of key-value pairs. Each value is accessible in read and write mode by its key.	System.Collections.HashTable, System.Collections.Generic.Dictionary<TKey, TValue>
object	An object with dynamic fields. Fields are dynamic in number and type.	System.Object (but more like System.Collections.Generic.Dictionary<System.String, System.Object>)
resource	A reference to an imported .NET or COM object	System.Object
closure	A reference to a function or method, a callback.	System.Delegate
Exception	The representation of an error that occurs at runtime.	System.Exception
Attribute	Additional information attached to a function, class, class member, or parameter that can be used by the scripting engine to apply special processing to the target symbol	System.Attribute

Type	Description	.NET equivalent
TypeInfo	A set of information describing a type. Used for introspection.	System.Type
MemberInfo	A set of information describing a class member, the base class of FieldInfo, PropertyInfo, MethodInfo and EventInfo. Used for introspection.	System.Reflection.MemberInfo
FieldInfo	A set of information describing a field. Used for introspection.	System.Reflection.FieldInfo
PropertyInfo	A set of information describing a property. Used for introspection.	System.Reflection.PropertyInfo
MethodInfo	A set of information describing a method. Used for introspection.	System.Reflection.MethodInfo
EventInfo	A set of information describing an event. Used for introspection.	System.Reflection.EventInfo
ParameterInfo	A set of information describing a parameter. Used for introspection.	System.Reflection.ParameterInfo

3.2.4 Literal values

Depending on their type, literal values have the following forms:

- **Null pointer**: the **null** keyword.
- **Boolean**: the **true** or **false** keyword.
- **Integer**: a sequence of decimal digits (i.e.: `0` to `9`) or a sequence of hexadecimal digits (i.e.: `0` to `9`, `A` to, or `a` to `f`) prefixed with `0X` or `0x` (**e.g.**: `154`, `0XF5D4`).
- **Long integer**: a literal integer with the `l` or `L` suffix or simply a huge literal integer (one that doesn't fit in a 32-bits signed integer).
- **Floating-point number**: a sequence of decimal digits optionally followed by a dot and another sequence of decimal digits, optionally followed again by `e` or `E` plus a `+` or `-` sign plus another sequence of decimal digits and optionally terminated by a `f` or `F` suffix. (**e.g.**: `0.5`, `1F`, `14.5e33`, `735e-3`, `88.33f`, `1.5e+6F`).
- **Decimal number**: just like floating-point numbers with a mandatory `d` or `D` suffix.
- **Complex number**: Any literal numeric value that has the suffix `i` or `I` is considered the imaginary part of a complex number. This makes AddyScript have a very natural syntax for representing complex numbers, as in `2 - 5i` or `1 + 2i`. When the imaginary part is 1, it should be represented as `1i` or `1I`. Simply typing `i` or `I` will cause the AddyScript interpreter to look for a variable with that name.
- **Date**: any valid date between backticks (**e.g.**: `` `2008-04-11` ``, `` `2:30 PM` ``, `` `05/18/2009 13:04` ``).
- **String**: a sequence of Unicode characters between single (`'`) or double (`"`) quotes. When a backslash (`\`) appears in a string, it alters the meaning of the following characters. Combinations of backslash and its followers are called **escape sequences** and have the following meaning:
  - ****: a literal backslash
  - **\'**: a single quote (not needed in strings wrapped in double quotes)
  - **\"**: a double quote (not needed in strings wrapped in single quotes)
  - **\t**: a horizontal tabulation
  - **\v**: a vertical tabulation
  - **\r**: a carriage return
  - **\n**: a line break
  - **\f**: a page break
  - **\b**: the backspace
  - **\a**: a beep
  - **\xnn** (where each **n** is an hexadecimal digit): any ascii character
  - **\unnnn** (where each **n** is an hexadecimal digit): any unicode character

  e.g.: `'Hello World!'`, `"Joe's dog's bell"`, `"C:\\Documents and Settings\\Addy"`, `'Living\r\nLa vida\tloca'`.
- **Blob**: A literal string value prefixed with the letter `b` or `B`. Each character in the string represents a single byte (**e.g.**: `b"Initial content of my buffer"`, `B'Another large binary object\xff\x7c'`).

  Notes:

  a. In literal numeric values represented with decimal digits, underscores (`_`) can be inserted between the digits to group them (in thousands, for example) and make the number more human-readable. There is no particular rule on how to group them, but it will typically be 3-by-3 (**e.g.**: `21_345_986`, `9_876_544_785`, `6_438.59e+33`).

  b. Basically, in AddyScript literal string values are not allowed to span over multiple lines of code. However, it can be necessary in certain circumstances to define a literal string constant that wraps all its content in the source code, including line-breaks and tabulations. That kind of literal string value is called a **verbatim string** and is declared in AddyScript using the `@` prefix. In a **verbatim string**, **escape sequences** are not needed at all, neither are they recognized. The only character that needs to be escaped is the string wrapper itself. This is done by doubling it (**e.g.**: `@'Say ''Hello'' to my friend Jonathan'`, `@"C:\MyMovies\""My Bad Movie.mp4"""`).

  c. Some literal string values embed expressions between curly braces into them. Those expressions are to be evaluated and replaced within the string by their value at runtime. The rendered string will be made of the static parts of its initial form concatenated with the results of the evaluation of embedded expressions. That kind of literal string value is called a **mutable string** and has to be prefixed with a dollar-sign (`$`). Each embedded expression in a **mutable string** can be followed by a format or length specification within the same pair of curly braces. The overall protocol to follow is exactly the same than for a call to the builtin **format** function (in fact, a mutable string is translated at runtime to a call to **format**). The process of rendering **mutable strings** is called **string interpolation**. **mutable strings** can also be verbatim; in that case they start with both dollar (`$`) and at (`@`) signs (**e.g.**: `$'item number {i}'`, `$"sine of PI is: {sin(PI)}"`, `$@'{emp.name} is a ''{emp.jobTitle}'' since {emp.hireDate:d}'`, `$@"movie ""D:\{movieDir}\{movieFile.Name}""" is {movieLen,3} minutes long`).

3.2.5 Initializers

Initializers are like literal values for composite types: they provide an initial value to them in a single step. AddyScript provides initializers for 4 data types: tuples, lists, maps, and sets. Depending on their type, initializers have the following forms:

- **Tuple**: a sequence of expressions (literal or not) in parentheses separated by commas (**e.g.**: `(5, -7, 2)`, `('Joe', 'Martin')`). The expressions that figure between the parentheses are called tuple items. If a tuple item is a sequential collection (i.e.: another **tuple**, a **list** or a **set**), it can be preceded by the **spread operator** (`..`) to indicate that it is not the item itself that is to be added to the tuple being initialized, but its contents (**e.g.**: `t1 = (5, 10, 15); t2 = (..t1, 20, 25); println(t2);` Output: `(5, 10, 15, 20, 25)`). For a single-item tuple, a final comma should be appended to the list to avoid confusion with parenthesized expressions (**e.g.**: `(18,)`, `(now(),)`). AddyScript doesn't allow a tuple to be empty. There should always be at least one item in a tuple.

  Note: Tuples are a new data type in AddyScript. The syntax used to represent tuple initializers was formerly used for complex initializers (with two items between the parentheses only respectively representing the real and the imaginary parts). AddyScript doesn't need complex initializers anymore as it has a built-in support for complex literals.

- **List**: a sequence of expressions (literal or not) in square brackets separated by commas (**e.g.**: `[4, 5, 'joe', 'adam', true, 0.5]`). The expressions that appear between the square brackets are called list items. Just like with tuples, if a list item is a sequential collection (i.e.: a **tuple**, another **list** or a **set**), it can be preceded by the **spread operator** (`..`) to indicate that it is not the item itself that is to be added to the list being initialized, but its contents (**e.g.**: `[17, 23, ..prime_numbers, 19]`, where *prime_numbers* is another list or a set).

- **Set**: a sequence of expressions enclosed in curly braces separated by commas. **e.g.**: `{'one', 'two', 'three'}`. As with tuple and list initializers, the spread operator can be used to include the contents of another sequential collection.

- **Map**: a sequence of key-value pairs between curly braces separated by commas. Each pair has the form: `key => value` where key and value are both expressions. **e.g.**: `{'name' => 'joe', 'age' => 18, 'job' => 'student'}`.

  Note: An empty map initializer must have this form: `{=>}`. This helps to make a difference between an empty map initializer and an empty set initializer.

3.2.6 Type checking

You can check the type of an expression by using the **is** operator. Here is an illustration:

```
lst = [5, 'andy', now(), PI, new Exception("")];

foreach (item in lst)
    if (item is int)
        println('int');
    else if (item is float)
       println('float');
    else if (item is date)
       println('date');
    else if (item is Exception)
       println('Exception');
    else
       println('something else');
```

Output

```
int
something else
date
float
Exception
```

Notes:

- This also works with user-defined classes and takes inheritance into account: if B is a subclass of A, then for any instance b of B, `b is A` returns **true**.
- For any data item x, the `x is void` test is simply a way to check whether x is declared in the current scope or not (exactly like JavaScript's `x === 'undefined'`).
- The **is** operator can optionally be followed by the **not** keyword to complement the result. This means that `x is not some_type` is the same as `!(x is some_type)`.
- The **is** operator can also be used to check if an expression matches a particular pattern. For example `x is { color: 'Blue' }` checks if *x* is an object with a property named *color* that has *Blue* as its value.

3.2.7 Conversion

For some operations, data are automatically converted to the right type. But this is not always the case. In the case where you have to manage conversion by yourself, use the C language conversion syntax like in the following example:

```
d = (date)'5/12/1980'; // d is a date
n = (decimal)'9876543210'; // n is a decimal
```

AddyScript also supports an alternative conversion syntax that uses type names as functions:

```
d = date('2026-01-10T19:35'); // d is a date
n = decimal('1234567890.987654321'); // n is a decimal
```

Remember that this doesn't work for user-defined classes. But most of the time, it will not be required for them since AddyScript uses duck-typing. In the case where you need to convert an object of a user-defined class to another type, you should provide a conversion method in that class.

3.2.8 Operators

Below are listed AddyScript's operators with their meaning:

Operator	Description
+ (unary)	Identity, does nothing
- (unary)	Opposite; transforms negative values to positive and vice-versa
!	In the prefix form, it's the logical negation (it returns **true** if the operand evaluates to **false** and vice-versa). In the suffix form, it checks for non-emptiness: it throws an exception if its operand is **null** or an empty collection or string
~	Bitwise complement
++	Increment; can be prefix or postfix
--	Decrement; can be prefix or postfix
+ (binary)	Addition
- (binary)	Subtraction
*	Multiplication
/	Division
%	Remainder of a division
**	Exponentiation
<<	Bitwise shift left
>>	Bitwise shift right
==	Equality test: tries to convert both operands to the same type before comparing them. Returns **true** anytime both operands can be considered as representing the same value. Returns **false** when both operands cannot be converted to the same type.
!=	Difference test: tries to convert both operands to the same type before comparing them. Returns **false** anytime both operands can be considered as representing the same value. Returns **true** when both operands cannot be converted to the same type.
===	Equality test: returns **true** if both operands are of the same type and have the same value. Returns **false** otherwise.
!==	Difference test: returns **true** if both operands are of the different types or have different values. Returns **false** otherwise.
<	... is less than ...
<=	... is less than or equal to ...
>	... is greater than ...
>=	... is greater than or equal to ...
&	Logical 'AND'
|	Inclusive logical 'OR' (returns **true** whenever one of its operands evaluates to **true**)
^	Exclusive logical 'OR' (only returns **true** when both operands have different logical values)
&&	Logical short-circuiting 'AND'; the second operand is not evaluated if the first is **false**
||	Logical short-circuiting 'OR'; the second operand is not evaluated if the first is **true**
startswith	String comparison operator, checks that the first operand starts with the second
endswith	String comparison operator, checks that the first operand ends with the second
contains	For strings, it checks that the second operand is part of the first. For collections, it checks that the collection in the left contains the item in the right
matches	String comparison operator, checks that the first operand is a match of the regular expression represented by the second

Operator	Description
in	A reversed form of the **contains** operator, `a in b` is equivalent to `b contains a`. The **in** operator can be preceded by a **not** keyword to complement the result. Thus, `a not in b` is equivalent to `!(a in b)`
=	Simple assignment; can be combined with a binary operator like in `x += y` (a shortcut for `x = x + y`) or in `x |= y` (a shortcut for `x = x | y`)
?:	A C-like conditional ternary operator: `x ? y : z` is a shortcut for `if (x) y else z`
??	returns the first operand if it's not empty (**i.e. null**, an empty collection or an empty string); returns the second otherwise
()	Parentheses are used to break precedence rules and force an expression to be evaluated in a certain way
[]	Square braces are used to access lists and maps items. They can also be used to extract a single character from a string. If a native type exposes an indexer, objects of this type will support the [] operator too
switch	Pattern matching operator. We'll look closer at this in the next sections
with	Mutable copy operator. We'll look closer at this in the next sections

3.2.9 Operator precedence

Operator precedence in AddyScript can be summarized in the following terms:

From the lowest to the highest priority, we have:

1. Assignment: =, +=, -=, *=, /=, %=, **=, &=, |=, ^=, <<=, >>=, ??=
2. The conditional ternary operator: ?:
3. Conditional binary operators: &, |, ^, &&, ||, ??
4. Relational operators: ==, !=, ===, !===, <, <=, >, >=, **startswith**, **endswith**, **contains**, **matches**, **in**, **is**
5. Addition and subtraction: +, -
6. Multiplication, division and bitwise shift operators: *, /, %, <<, >>
7. Exponentiation: **
8. Postfix unary operators: ++, --, !
9. Prefix unary operators: +, -, !, ~, ++, --
10. Wrapping and special binary operators: (), [], **switch**, **with**

3.2.10 Expressions

An expression is any combination of operands and operators that can produce a value. An operand can be a literal value, a named constant, a reference to a variable, a reference to a list item, a reference to a field or property, a function or method call, the keyword **this**, an assignment, a unary expression, a binary expression, a ternary expression, or any of these wrapped into parentheses.

3.2.11 Assignments

An assignment is a particular kind of expression where a value is set to a location in memory. So it's made of two child expressions: the **lvalue** which represents the memory location about to be set, and the **rvalue** which represents the value that's being assigned to the lvalue. Both children are separated by an operator. The typical assignment operator in AddyScript is the equal sign (`=`). So an assignment in AddyScript typically looks like this:

```
lvalue = rvalue;
```

But AddyScript provides other assignment operators which are combinations of binary arithmetic or logical operators with the equal sign. When such an operator is used, the initial value of lvalue is combined with rvalue using the given binary operator and then the result is reassigned to lvalue. The complete list of these combined operators is given in the table below:

Operator	Equivalence
`+=`	`a += b` is equivalent to `a = a + b`
`-=`	`a -= b` is equivalent to `a = a - b`
`*=`	`a *= b` is equivalent to `a = a * b`
`/=`	`a /= b` is equivalent to `a = a / b`
`%=`	`a %= b` is equivalent to `a = a % b`
`**=`	`a **= b` is equivalent to `a = a ** b`
`<<=`	`a <<= b` is equivalent to `a = a << b`
`>>=`	`a >>= b` is equivalent to `a = a >> b`
`&=`	`a &= b` is equivalent to `a = a & b`
`|=`	`a |= b` is equivalent to `a = a | b`
`^=`	`a ^= b` is equivalent to `a = a ^ b`
`??=`	`a ??= b` is equivalent to `a = a ?? b`

Group Assignment

Sometimes you need to assign values to multiple variables. You can do this in multiple steps, like in `a = 5; b = 2; c = -7`. But AddyScript has a type of statement in its syntax that allows you to do this in a more elegant way: **group assignment**. In a group assignment, a tuple of values is assigned to a tuple of variables (in the broad sense of the term). This allows you to set the value of multiple variables at once. So a typical group assignment looks like this:

```
(var1, var2, ..., varN) = (val1, val2, ..., valN);
```

Notes:

- Both tuples must have exactly the same number of elements.
- Neither tuple must be empty.
- Each element of the left tuple must be a valid reference (such as a variable, a list item, an object property, or another tuple).
- The right tuple can contain elements of type **tuple**, **list** or **set** preceded by the *spread* operator (`..`). In this case, the contents of the collection replace the collection itself in the parent tuple. This is very convenient for assigning values to several variables at once from elements of a collection.
- Any underscore (`_`) in the left tuple is simply ignored. This is useful when you want to extract only some values from the right tuple without having to declare dummy variables for the values you want to skip.

Example

```
1  l = [7, 6, 4, 2];
2  (a, b, _, c) = (..l);
3  println($'a = {a}');
4  println($'b = {b}');
5  println($'c = {c}');
```

Output

```
a = 7
b = 6
c = 2
```

3.2.12 The **let** keyword

AddyScript has a dedicated keyword that can be used to introduce an assignment. That's the **let** keyword. It's especially useful when there is a risk of ambiguity in the way the code is parsed (like when destructuring an object).

An assignment with the **let** keyword looks like this:

```
let lvalue = rvalue;
```

With this syntax chaining assignments is not allowed. No other operator than the equal-sign (`=`) can be used.

Example

```
let a = 5;
let b = 8;
let c = a + b;
```

Note: **let** does not locally redeclare the variable. It's just a way to introduce an assignment.

3.2.13 Reading and printing values from/to the console

As you may have already guessed, you typically read a value from the console in AddyScript by using the **readln** function. It lets the user type a string and returns that string once the user presses on the [Return] key. The returned string can be converted to any type following your needs and expectations. **readln** accepts an optional parameter, a string that would be displayed as a prompt if given.

Similarly, you print values to the console in AddyScript by using the **print** and **println** functions. Both functions are similar except that **print** remains on the same line after printing a value while **println** automatically skips to the next line. Both functions accept a variable number of arguments. The first argument, if present, represents a format string; any `{n}` sequence (n being an integral number) in that string will be replaced with one of the corresponding following argument (`0` for the second, `1` for the third, and so on...). **print** requires at least on argument while **println** can be used without any argument. When used without an argument, **println** simply skips to the next line.

If you are integrating AddyScript in your own system, you could change the meaning of those statements and make **print** and **println** display a popup window for example.

3.3 Controlling the program flow

Like any other language, AddyScript provides means to control the program flow. The following sections describe these means.

3.3.1 The If-Else statement

The if-else statement is used to perform an action only if a certain condition is met. An alternative action can be defined to be executed in case the condition is not met. An if-else therefore has the following 2 possible forms:

Form 1:

`if (condition) action`

Example

```
n = randint(20); // n is a randomly generated number between 0 and 20

if (n > 10)
    println('n is greater than 10');
```

Form 2:

`if (condition) action else alternative_action`

Example

```
i = randint(100); // i is a randomly generated number between 0 and 100

if (i % 2 == 0)
    println('even');
else
    println('odd');
```

Note: In all cases, actions can be statement blocks or other if-else statements. A block is a series of statements enclosed in curly braces. It is treated by the interpreter as a single composite statement. Use a block if you want your **if** or **else** section to perform multiple actions.

3.3.2 The Switch statement

Like the if-else statement, the switch statement is used to choose the action to perform based on the value of an expression. The main difference is that unlike the if-else statement, the switch statement is not limited to 2 alternatives. Its syntax is as follows:

```
switch (expression)
{
    case value1:
       action1;
       break;
    case value2:
    case value3:
      action2;
      break;
    //...
    case valueN:
       actionN;
       break;
    //...
    default:
       defaultAction;
       break;
}
```

Breaks are not required but if a **break** is missing, execution will continue at the next **case** section. The **default** section is also optional but at least one **case** section (or *default* section) must be present in the switch block.

Example

```
result = int(readln("What's your result? "));

switch (result)
{
    case 0:
       println('Null!');
       break;
    case 1:
    case 2:
    case 3:
       print('Very '); // This will finally print 'Very Bad!' as there is no break!
    case 4:
       println('Bad!');
       break;
    case 5:
       println('Average!');
       break;
    case 6:
       println('Good!');
       break;
    case 7:
    case 8:
    case 9:
       println('Very Good!');
       break;
    case 10:
       println('Perfect!');
       break;
    default:
       println('Out of range');
       break;
}
```

3.3.3 Loops

Loops are used to repeat an action until a condition is met. In AddyScript, we have four different kinds of loops. These are:

The While loop:

It performs an action as long as a condition is satisfied. If the condition is initially unsatisfied, the action will never be performed.

Syntax:

`while (condition) action`

Example

```
i = 1;
while (i <= 12)
{
    println('2 x {0} = {1}', i, 2*i);
    ++i;
}
```

Output

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
2 x 11 = 22
2 x 12 = 24
```

The Do-While loop:

Just like the While loop, it performs an action as long as a condition is satisfied. The action is performed before the condition is checked thus leading to the action always being performed at least once.

Syntax:

```
do action while (condition);
```

Example

```
i = 1;

do
{
   println('2 x {0} = {1}', i, 2*i);
   ++i;
} while (i <= 12);
```

The For loop:

It's mainly used to perform an action for each value of a counter.

Syntax:

```
for (counter_initialization; counter_limit_test; counter_increment) action
```

Example

```
for (i = 1; i <= 12; ++i)
   println('2 x {0} = {1}', i, 2*i);
```

Notes:

1. In the above example, *counter_initialization* and *counter_increment* are single statements, but they could also be comma-separated lists of statements; *counter_limit_test* on the other hand is always a single logical expression.
2. The example given is equivalent to the while loop example.

The For-Each loop:

It's used to iterate over a collection of items, performing an action on each of them.

It has the following forms:

Form 1:

```
foreach (item in collection) action
```

Example

```
words = ['john', 'paul', 'second', 'the', 'pope'];

foreach (word in words)
   print(word + ' ');

println();
```

Output

```
john paul second the pope
```

Form 2:

```
foreach (key => value in collection) action
```

Example

```
jobs = {'paul' => 'general manager', 'roland' => 'accountant', 'david' => 'driver'};

foreach (name => job in jobs)
   println(name + ' is a ' + job);
```

Output

```
paul is a general manager
roland is a accountant
david is a driver
```

Notes:

- Even in the first form, there is an implicit key named **__key**.
- In both forms, the *collection* must be a **string**, a **blob**, a **tuple**, a **list**, a **map**, a **set**, or an instance of a .NET class that implements the **IEnumerable** interface.
- It is possible to use the foreach loop to iterate over instances of user-defined classes that implement the *interator protocol*. We'll come back on this in the section dedicated to inheritance and polymorphism.

3.3.4 Jumps

Jumps allow execution to resume from a location other than the next instruction. They are generally very useful for prematurely exiting from functions or loops. The table below describes them:

Jump statement	Syntax	Effect
continue	`continue;`	Skips to the next iteration of a loop before the end of the current iteration
break	`break;`	Exits a loop before the last iteration is reached. Exits a switch block before the last statement is reached.

goto	`goto label;`
	or *(in a switch block)*
	`goto case X;`
	`goto default;`
		Unconditionally jumps to the specified location.
		In a switch block can also jump to one of the case labels.
		Note: A label is any identifier that's appears at the beginning of an instruction and is followed by a colon. **e.g.**: in `printout: println('hello');` *printout* is the label.

return	`return;`
	or
	`return expression;`
		Returns from a function with or without a value.

throw	`throw exception;`
	or
	`throw "some error message";`
		Raises an error and stops the execution of the script until the raised error is caught by a **try-catch-finally** statement.

3.3.5 Pattern matching

So far, we've seen the **switch** statement and the possibilities it offers. But its syntax is somewhat cumbersome: we have to repeat the **case** keyword a lot of times. We also have to use **break** to prevent the flow of execution from continuing to the next **case** section. This also leaves us with a very poor choice about what kind of operation to use to compare the value of our expression with the value of each **case** label. This is the kind of problem that pattern matching solves. It combines a better matching syntax with the **switch** keyword, which this time is used as an operator, helping us create expressions that not only choose what action to perform based on the value of an expression, but also return a value. A pattern matching expression typically looks like this:

```
expression switch {
    pattern1 => result1,
    pattern2 => result2,
    ...
```

```
    patternN => resultN
};
```

Notes:

1. Each *result* in this syntax is an expression.
2. Patterns come in different types (see the table below).
3. There is no default pattern at all, there is just a special type of pattern that matches everything, thus acting as a default (or fallback) pattern.
4. A result can be produced in a block, in which case the block must end with a **yield** statement which acts as a **return** statement in a function body. Using a **return** statement here will either fail or interrupt the flow of execution without setting the value returned by the block.
5. Throwing an exception is also a valid result.
6. A pattern can be supplemented with a **guard** which is a logical expression linked to the pattern by a **when** keyword.
7. The **guard** restricts the cases that should match by adding a condition to check.
8. During pattern evaluation the value of the tested expression is represented by the **__value** automatic variable.

Example 1

```
n = (int)readln('Please type a number: ');

res = n switch {
    not >= 0 => {
        println('this is a block');
        yield 'negative';
    },
    >= 0 and <= 9 => 'from 0 to 9',
    10 or 11 or 12 => '10, 11, or 12',
    (>= 14 and <= 17) or 13 or 18 => '13 to 18',
    int when 20 <= __value && __value < 30 => '20 to 29',
    >= 30 => '30 and above',
    _ => throw 'an exception for 19'
};

println(res);
```

Example 2

```
l = [
    new { name = 'cube', size = 18, color = 'blue' },
    'hello funny people, how funny are you?',
    new Exception('something went wrong'),
    null,
    PI,
    (8, 4, 6),
    now()
];

foreach (o in l) {
    res = o switch {
        null => 'absolutely null',
        float => "i'm floating",
        Exception(message) => message,
        object { name : 'cube', color : 'blue' } => 'a blue cube',
        $'hello {quality} {target}, how {quality} are you?' => $'very {quality} {target} indeed!',
        (8, _, 6) => 'a triplet that starts with 8 and ends with 6',
        _ => 'did not match any pattern'
    };

    println($'o is: {o}');
    println($'result with o: "{res}"');
    println();
}
```

Output

```
o is: <object {name = cube, size = 18, color = blue}>
result with o: "a blue cube"

o is: hello funny people, how funny are you?
result with o: "very funny people indeed!"

o is: Exception
result with o: "something went wrong"

o is:
result with o: "absolutely null"

o is: 3,141592653589793
```

```
result with o: "i'm floating"

o is: (8, 4, 6)
result with o: "a triplet that starts with 8 and ends with 6"

o is: 2026-01-26 15:11:01
result with o: "did not match any pattern"
```

The above examples showcase the different kinds of patterns. They are explained in the table below:

Pattern	Syntax	Description
always true pattern	an underscore (_)	Matches everything; usually acts as the default. Should normally appear at the end of the list to avoid short-circuiting the flow of comparisons
null checking pattern	the keyword **null**	Matches only expressions that evaluate to the null reference
literal value pattern	a literal **bool**, **int**, **long**, **float**, **decimal**, **date**, or **string**	Matches only expressions whose value is equal to its own
relational pattern	a relational binary operator (one of: <, >, <=, or >=) followed by a literal **bool**, **int**, **long**, **float**, **decimal**, **date**, or **string**	Matches expressions whose value have the relation indicated by its operator with its own value
regex pattern	the **matches** operator followed by a literal **string**	A particular kind of relational pattern that checks that the expression evaluates to a string that matches the given regular expression
type pattern	any type name, including user-defined classes	Matches objects of the given type; also matches instances of subclasses if the given type is a class
object pattern	a curly-braced list of *property : pattern* pairs optionally preceded by a type name; **e.g.**: `Exception { message : 'something went wrong'}` *(An Exception with "something went wrong" as its message)*	Matches an object of the given type (if a type name is specified) with properties of the given names that match the corresponding patterns
destructuring pattern	a type name followed by a list of property names between parentheses; **e.g.**: `Exception(name, message)}`	Matches the expression to be evaluated to an object of the specified type with properties defined by the given names. Any matching object is automatically destructured, and a variable with the same name as each of the specified properties is initialized with the value of the corresponding property.
negative pattern	the keyword **not** followed by a child pattern; **e.g.**: `not int` *(anything but an integer)*	Negates its child pattern.
logical pattern	two elementary patterns joined with the **and**/**or** keywords; **e.g.**: `10 or 11`, `int and >= 0`	Whith the **or** operator, tries to match the expression against at least one of its components. Whith the **and** operator, tries to match the expression against both components.
grouping pattern	another pattern between parentheses; **e.g.**: `(>= 12 and <= 18) or ((>= 22 and <= 30) and not 25)` *(in the range 12 to 18 or in the range 22 to 30 except 25)*	Changes the priority in which chained logical patterns are evaluated. By default they are evaluated left to right. Any pattern between parentheses in a chain is evaluated first.
positional pattern	a coma seperated list of elementary patterns between parentheses (i.e.: a tuple of patterns).	Matches expressions whose value is an iterable data item (one that can be used in a **foreach** loop) of the same length than the given list of patterns with items that match the elementary patterns at corresponding positions.
string destructuring pattern	a mutable string in which all substitutions are variable references.	Matches expressions whose value is a string that matches the regular expression generated from the given mutable string. In this regular expression, a capture group is generated from each substitution. If the same variable appears multiple times among the substitutions, all its occurrences will reference the same capture group, thus requiring the same substring to appear in different places within the input string.

3.4 Special types

AddyScript supports a number of special data types that can be used for various purposes. These are structured data types that behave like primitive types. They are described below:

3.4.1 Rational Numbers

A rational number is a pair of integers resulting from their division. The first element of the pair, called the **numerator**, can have any sign, while the second, called the **denominator**, is always strictly positive. The main purpose of defining a rational number type in AddyScript is to improve the handling of integer division: dividing two integers always produces a rational number as the result. If you divide two integers and want the result to also be an integer, simply convert the result to the desired type. In AddyScript, rational numbers are represented by the **rational** data type. Variables of this type can be used in arithmetic operations like any other numeric data and are convertible to and from all other numeric data types. AddyScript even offers a smart conversion mechanism for **double** and **decimal** to **rational** numbers: it attempts to guess which fraction might have generated the floating-point or decimal number being converted. However, there is no literal value or initializer for the **rational** type; the only way to get a rational number is by dividing two integers (**int** or **long**). Here is an example script that uses rational numbers:

Example

```
a = 3/4;
b = 1/4;
c = 11/7;
l = (14, 271, 0.3333333333333333, .5d);

println('the numerator of {0} is {1}', a, a.num);
println('the denominator of {0} is {1}', a, a.den);
println('the inverse of {0} is {1}', a, a.inverse());
println('the inverse of {0} is {1}', b, b.inverse());
println('{0} + {1} = {2}', a, b, a + b);
println('{0} - {1} = {2}', a, b, a - b);
println('{0} * {1} = {2}', a, b, a * b);
println('{0} / {1} = {2}', a, b, a / b);
println('{0} ** 2 = {1}', a, a ** 2);
println('sign({0}) = {1}', a, sign(a));
println('abs({0}) = {1}', a, abs(a));

println();
println('{0} as an int is {1}', c, int(c));
println('{0} as a long is {1}', c, long(c));
println('{0} as a float is {1}', c, float(c));
println('{0} as a decimal is {1}', c, decimal(c));
println('{0} as a complex is {1}', c, complex(c));

println();
foreach (n in l) {
    println($'{n} ({n.type}) converted to rational is {rational(n)}');
}
```

Output

```
the numerator of (3/4) is 3
the denominator of (3/4) is 4
the inverse of (3/4) is (4/3)
the inverse of (1/4) is 4
(3/4) + (1/4) = 1
(3/4) - (1/4) = (1/2)
(3/4) * (1/4) = (3/16)
(3/4) / (1/4) = 3
(3/4) ** 2 = (9/16)
sign((3/4)) = 1
abs((3/4)) = (3/4)

(11/7) as an int is 1
(11/7) as a long is 1
(11/7) as a float is 1,5714285714285714
(11/7) as a decimal is 1,571428571428571428571428571428571428571428571428571428571428571
(11/7) as a complex is 1,5714285714285714

14 (int) converted to rational is 14
27 (long) converted to rational is 27
0,3333333333333333 (float) converted to rational is (1/3)
0,5 (decimal) converted to rational is (1/2)
```

Rational Number API

The following table summarizes the members of the **rational** type and their usage:

Member	Nature	Description
`long num { read; }`	property	Gets the numerator of the target rational number.
`long den { read; }`	property	Gets the denominator of the target rational number.
`rational|long inverse()`	method	Gets the inverse of the target rational number. This can be another rational number or a long integer.

3.4.2 Complex Numbers

AddyScript also supports complex numbers as a primitive data type. This type is represented by the **complex** class. Internally, a complex number is represented as a pair of real numbers; the first member of the pair is called the **real part** while the second is called the **imaginary part**. The language supports literal values for purely imaginary complex numbers (i.e.: those that have a zero real part). Those are numeric literal values with the **i** or **I** suffix (**e.g.:** `2i` or `-5I`). Complex numbers can be involved in arithmetic operations like other numeric data types. Any arithmetic operation involving a real number and a purely imaginary complex number will produce a complex number as a result. So in an expression like `3 + 2i`, `3` is a real number, `2i` is a purely imaginary complex number and the result is a complex number. The purely *imaginary one* **i** is always represented as `1i`, so instead of `2 + i` use `2 + 1i`. The *sqrt* (square root) function always returns a complex number when its argument is negative (**e.g.:** `sqrt(-1)` returns `1i`). Here is an example script using complex numbers:

Example

```
a = 2 - 1i;
b = 1 + 2i;

println('the real part of {0} is {1}', a, a.real);
println('the imaginary part of {0} is {1}', a, a.imag);
println('the conjugate of {0} is {1}', a, a.conjugate());
println('{0} + {1} = {2}', a, b, a + b);
println('{0} - {1} = {2}', a, b, a - b);
println('{0} * {1} = {2}', a, b, a * b);
println('{0} / {1} = {2}', a, b, a / b);
println('{0} ** 2 = {1}', a, a ** 2);
println('abs({0}) = {1}', a, abs(a));
```

Output

```
the real part of (2-1i) is 2
the imaginary part of (2-1i) is -1
the conjugate of (2-1i) is (2+1i)
(2-1i) + (1+2i) = (3+1i)
(2-1i) - (1+2i) = (1-3i)
(2-1i) * (1+2i) = (4+3i)
(2-1i) / (1+2i) = -1i
(2-1i) ** 2 = (3,0000000000000004-4i)
abs((2-1i)) = 2,23606797749979
```

Complex Number API

The following table summarizes the members of the **complex** class and their usage:

Member	Nature	Description
`float real { read; }`	property	Gets the real part of a complex number.
`float imag { read; }`	property	Gets the imaginary part of a complex number.
`complex conjugate()`	property	Gets the conjugate of a complex number.

3.4.3 Dates

Date/time values are instances of the **date** class. You will typically create **date** instances either by calling the global **now** function (which returns the current date and time) or by using a date literal. A date literal is anything enclosed in backticks (`) that can be translated by the .NET runtime to a

value of type *System.DateTime*. They typically conform to the date format of the local culture (**e.g.**: `` `03/03/1980` ``, `` `nov, 16 2002` ``). Here is an example of a script that manipulates dates:

Example

```
println("hello! it's {0:t} o'clock", now());
d = (date)readln("what's your birth date? ");
println("it was a " + d.weekday);
today = now().$date;
println("you are {0} years old now", today.subtract(d, "year"));
d = d.add(1, "year");
println("your first birthday was on {0:d}", d);
```

Date API

The **date** class supports the following operators:

Operator	Operands	Description
+	A date on one side and a duration on the other side	Adds the given duration to the given date.
-	Two dates	Computes the difference of two dates and returns a duration.

In addition to those operators, the **date** class exposes the following members:

Member	Nature	Description
`date of(int year, int month, int day, int hour = 0, int minute = 0, int sesond = 0, int millisecond = 0)`	static method	A static factory method for creating dates. The *hour*, *minute*, *second* and *millisecond* arguments are optional. Only *year*, *month* and *day* are required.
`int year { read; }`	property	Extracts the *year* component of the target date object.
`int month { read; }`	property	Extracts the *month* component of the target date object.
`int yearday { read; }`	property	Extracts the *day-of-year* component of the target date object.
`string weekday { read; }`	property	Extracts the *day-of-week* component of the target date object.
`int day { read; }`	property	Extracts the *day-of-month* component of the target date object.
`int hour { read; }`	property	Extracts the *hour* component of the target date object.
`int minute { read; }`	property	Extracts the *minute* component of the target date object.
`int second { read; }`	property	Extracts the *second* component of the target date object.
`int millisecond { read; }`	property	Extracts the *millisecond* component of the target date object.
`date $date { read; }`	property	Extracts the *date-only* part (in the strict sense of the term) of a date object.
`duration time { read; }`	property	Extracts the *time-only* part of a date object as a duration.
`long ticks { read; }`	property	Gets the number of ticks stored in the target date instance.
`date add(int amount, string unit)`	method	Adds some amount of the given unit to the target date object and returns an altered copy of it. The target itself remains unchanged. Accepted units are: "year", "month", "day", "hour", "minute", "second" and "millisecond".
`date addTicks(long ticks)`	method	Adds some ticks to the target date object and returns an altered copy of it. The target itself remains unchanged.
`int subtract(date d, string unit)`	method	Computes the difference in the given unit between the target date object and its first argument. Accepted units are: "year", "month", "day","hour", "minute", "second" and "millisecond".

3.4.4 Durations

A duration is the amount of time elapsed between two dates. Durations are very important in AddyScript as some operations on dates involve and/or return durations. The language syntax doesn't provide neither literal values nor initializers for duration. A duration can only be obtained by computing the difference of two dates or by invoking a factory function. AddyScript has five (5) functions that all take a numeric value as argument and return a properly initialized duration. Those are described in one of the tables bellow.

Example

```
d1 = `2025-09-17 14:30`;
d2 = date::of(2019, 8, 30, 19, 15);
elapsed = d1 - d2;
println("time elapsed between {1} and {0} : {2}", d1, d2, elapsed);
println("number of days {0}", elapsed.days);
println("number of hours {0}", elapsed.hours);
println("number of minutes {0}", elapsed.minutes);
println("total number of hours {0}", elapsed.totalHours);
println("total number of minutes {0}", elapsed.totalMinutes);
d3 = now() + days(100) + hours(15) + minutes(45);
println("in 100 days 15 hours and 45 minutes we will be on {0}", d3);
```

Duration API

The functions bellow can be used to create **duration** instances:

Function	Description
duration days(any value)	Creates a duration with the given number of days.
duration hours(any value)	Creates a duration with the given number of hours.
duration minutes(any value)	Creates a duration with the given number of minutes.
duration seconds(any value)	Creates a duration with the given number of seconds.
duration milliseconds(any value)	Creates a duration with the given number of milliseconds.

The **duration** class supports the following operators:

Operator	Operands	Description
+	A duration on both sides	Computes the sum of two durations.
-	A duration on both sides	Computes the difference of two durations.

In addition to those operators, the **date** class exposes the following members:

Member	Nature	Description
`duration of(int days, int hours, int minutes, int sesonds, int milliseconds)`	static method	A static factory method for creating durations. All parameters are mandatory.
`int days`	property	Extracts the *days* component of the target date object.
`int hours`	property	Extracts the *hours* component of the target date object.
`int minutes`	property	Extracts the *minutes* component of the target date object.
`int seconds`	property	Extracts the *seconds* component of the target date object.
`int milliseconds`	property	Extracts the *milliseconds* component of the target date object.
`float totalDays`	property	Extracts the total number of days (eventually factional) stored in the target date object.
`float totalHours`	property	Extracts the otal number of hours (eventually factional) stored in the target date object.
`float totalMinutes`	property	Extracts the otal number of minutes (eventually factional) stored in the target date object.
`float totalSeconds`	property	Extracts the _otal number of seconds (eventually factional) stored in the target date object.
`float totalMilliseconds`	property	Extracts the otal number of milliseconds (eventually factional) stored in the target date object.
`long ticks { read; }`	property	Gets the number of ticks stored in the target duration instance.

3.4.5 Strings

In AddyScript, sequences of characters more commonly called *strings* are instances of the **string** class. You can obtain a string in various ways such as using a literal string value, invoking the global **format** or **readln** functions, invoking the "toString" method of any object and so on. In fact, the **string** class is one of the more commonly used data types in AddyScript (and I think, in any scripting language). This is why it exposes a wide range of methods. Here is an example of a script that uses strings:

Example

```
s = readln("Type some text: ");

println("Lower case: " + s.toLower());
println("Upper case: " + s.toUpper());
println("Words: " + ", ".join(..s.split(@"\s+")));

w = readln("Type some word: ");

if (s startswith w)
    println(w + " is at the beginning of the text");
else if (s endswith w)
    println(w + " is at the end of the text");
else if (s contains w)
    println(w + " is contained in the text at position " + s.indexOf(w));

println("Replacing " + w + " by hello gives: " + s.replace(w, "hello"));

readln("Press [Return]");

println("Those are substrings of " + s);

for (i = 0; i < s.length - 1; ++i)
    for (j = 1; j <= s.length - i; ++j)
       println(s.substring(i, j));
```

String API

The **string** class supports the following operators:

Operator	Operands	Description
+	Two string or A string on one side and anything else on the other side	Concatenates two strings. Automatically casts to string any argument that's not a string by invoking its "toString" method.
*	A string on one side and an integer on the other side	Concatenates a string to itself the given number of times.
startswith	Two strings	Returns **true** if the left string starts with the right string. Returns **false** otherwise.
endswith	Two strings	Returns **true** if the left string ends with the right string. Returns **false** otherwise.
contains	Two strings	Returns **true** if the right string is a substring of the left string. Returns **false** otherwise.
matches	Two strings	Returns **true** if the left string is a match for the regular expression contained in the right string. Returns false otherwise.
[int]	A string to the left and an integer between brackets (called the **index**)	Gets a single character from the target string (the character is returned as a one-character string). A negative index indicates that the character should be searched from the end of the string back to the beginning.
[int..int]	A string to the left and a pair of integers separated by a double-dot (..) between brackets (representing a range, the first integer is the range **lower bound**, the second is the range **upper bound**)	Gets a slice (or substring) of the target string. Negative bounds are evaluated relative to the length of the string. Both bounds are optional. If the lower bound is omitted, it's automatically replaced with 0. A missing upper bound will be replaced by the length of the string. If both bounds are omitted, the entire target string is returned.

In addition to those operators, the **string** class exposes the following members:

Member	Nature	Description
`int length { read; }`	property	Gets the length of the string.
`int indexOf(string value, int start = 0, int length = 0)`	method	Searches for a substring and returns its position in the target string if found or -1 otherwise. The optional "start" and "length" parameters tell which part of the string to search. if "start" is negative, it will be evaluated modulo the total length of the target string. if "length" is negative or zero, it will be ignored.
`int lastIndexOf(string value, int start = -1, int length = 0)`	method	Searches for a substring backward and returns its position in the target string if found or -1 otherwise. The optional "start" and "length" parameters tell which part of the string to search. if "start" is negative, it will be evaluated modulo the total length of the target string. if "length" is negative or zero, it will be ignored.
`string toLower()`	method	Converts the target instance to lowercase.
`string toUpper()`	method	Converts the target instance to uppercase.
`string capitalize()`	method	Converts the first character of the target instance to uppercase.
`string uncapitalize()`	method	Converts the first character of the target instance to lowercase.
`string substring(int start, int length = 0)`	method	Extracts a substring (or slice) of the given length from the target string at the given position. If "position" is negative, it is evaluated modulo the length of the target string. If "length" is omitted (or 0 or negative), the part of the target string that's to the right of the given position is returned.
`string insert(int index, string value)`	method	Inserts a string into the target instance at the given position. If "position" is negative, it is evaluated modulo the length of the target string.
`string remove(int index, int count = 0)`	method	Removes "count" characters from the target string starting at the position indicated by "index" or simply truncates the target string at that position if "count" is omitted (or negative or 0). If "index" is negative, it is evaluated modulo the length of the target string.
`string replace(string pattern, string value)`	method	Replaces each occurrence of the given pattern (a regular expression) with "value" in the target string.
`string ltrim(string chars = " ")`	method	Removes each of the given characters from the left of the target string.
`string rtrim(string chars = " ")`	method	Removes each of the given characters from the right of the target string.
`string trim(string chars = " ")`	method	Removes any of the given characters from both ends of the target string.
`string lpad(int width, string padding = " ")`	method	Repeatedly adds the given character to the left of the target string until it reaches the length specified by "width".
`string rpad(int width, string padding = " ")`	method	Repeatedly adds the given character to the right of the target string until it reaches the length specified by "width".
`tuple split(string pattern = @"\s+")`	method	Creates a tuple of substrings of the target string separated by the given separator. The separator must be a regular expression.
`string join(..values)`	method	Creates a string by concatenating the given values. The target instance is used as a separator. When invoked with a collection as an argument, the *spread* operator (..) must be used to expand the collection.

Note: none of the above methods alters the target string. They simply create a modified copy of it return that copy. The original string remains unchanged.

3.4.6 Blobs

A **blob** is AddyScript's abstraction of a byte array. Blobs are especially useful when it comes to using methods in .NET classes that take a byte array as an argument (such as the *Read* and *Write* methods of the *System.IO.Stream* class). Blobs have a lot in common with strings, but unlike strings, they are not immutable: their contents are meant to be changed.

There are several ways to get blobs, such as using a blob literal value (a string literal preceded by a "b" or "B"), or invoking the static method *blob::of* (which expects the desired length in bytes as an argument), or invoking one of the other static methods "fromHexString" and "fromBase64String" of the blob class which as their name suggests, convert strings to blobs using base-16 or base-64 encoding. You can also convert a string to a blob, which will convert each of its characters to a byte.

Once a blob is created, you can access each of its bytes individually for reading or writing, you can get its length by reading the "length" property, you can fill it partially or completely with a byte of your choice, you can resize it or copy it to another blob at a particular position. You can also create slices of blobs like you do with strings.

Here is an example script that manipulates blobs:

Manipulating blobs

```
b1 = b'Hello friends!';
b2 = blob::of(24);
println($'b1 = {b1}, b1.length = {b1.length}');
println($'b2 = {b2}, b2.length = {b2.length}');
println($'b2[0] = {b2[0]}, b2[-1] = {b2[-1]}');
println();

b2.fill(ord('a'), 0, 8);
b2.fill(ord('b'), 8, 8);
b2.fill(ord('c'), 16, 8);
println($'b2 = {b2}, b2[0] = {b2[0]}, b2[-1] = {b2[-1]}');
println();

b1.copyTo(b2);
println($'b2 = {b2}, b1 == b2 ? {b1 == b2}');
println();

b2 = b2[..b1.length];
println($'b2 = {b2}, b2.length = {b2.length}, b1 == b2 ? {b1 == b2}');
println();

b2 = blob::fromHexString('48656C6C6F20667269656E647321');
println($'b2 in base-16 = {b2.toHexString()}, b2.length = {b2.length}, b1 == b2 ? {b1 == b2}');
println();

b1 = blob::fromBase64String('SGVsbG8gZnJpZW5kcyE=');
println($'b1 in base-64 = {b1.toBase64String()}, b1.length = {b1.length}, b1 == b2? {b1 == b2}');
println();
```

Blob API

The **blob** class supports the following operators:

Operator	Operands	Description
+	Two blobs	Concatenates two blobs.
*	A blob on one side and an integer on the other side	Concatenates a blob to itself the given number of times.
contains	A blob to the left and an integer to the right	Returns **true** if the blob contains the given byte at any index. Returns **false** otherwise.
[int]	A blob to the left and an integer between brackets (called the **index**)	Gets a single byte from the target blob (the byte is returned as an integer). A negative index indicates that the byte should be searched from the end of the blob back to the beginning.
[int..int]	A blob to the left and a pair of integers separated by a double-dot (..) between brackets (representing a range, the first integer is the range **lower bound**, the second is the range **upper bound**)	Gets a slice of the target blob. Negative bounds are evaluated relative to the length of the blob. Both bounds are optional. If the lower bound is omitted, it's automatically replaced with 0. A missing upper bound will be replaced by the length of the blob. If both bounds are omitted, the entire target blob is returned.

In addition to those operators, the **blob** class exposes the following members:

Member	Nature	Description
`int length { read; }`	property	Gets the length of the blob.
`blob of(int length)`	static method	Creates a blob with the desired length. The returned blob is initially filled with zeros.
`blob fromHexString(string hexString)`	static method	Creates a blob by converting the given string to a byte array. The string should be made of hexadecimal digits in odd number
`string toHexString()`	method	Gets a string that represents the target blob as a large hexadecimal integer
`blob fromBase64String(string base64String)`	static method	Creates a blob by converting the given string to a byte array. The string should be made of base 64 digits
`string toBase64String()`	method	Gets a string that represents the target blob as a large base 64 integer
`int indexOf(int byteValue, int start = 0, int length = 0)`	method	Searches for a byte and returns its position in the target blob if found or -1 otherwise. The optional "start" and "length" parameters tell which part of the blob to search. if "start" is negative, it will be evaluated modulo the total length of the target blob. if "length" is negative or zero, it will be ignored.
`int lastIndexOf(int byteValue, int start = -1, int length = 0)`	method	Searches for a byte backward and returns its position in the target blob if found or -1 otherwise. The optional "start" and "length" parameters tell which part of the blob to search. if "start" is negative, it will be evaluated modulo the total length of the target blob. if "length" is negative or zero, it will be ignored.
`void fill(int byteVale, int start = 0, int length = 0)`	method	Fills a blob with the given byte starting at position "start" and stopping at position "start" + "length". Both "start" and "length" are evaluated modulo the length of the blob. If "length" is negative, it is replaced with target.length - "start", where target is the blob on which the method is invoked
`void copyTo(blob other, int srcIndex = 0, int destIndex = 0, int length = 0)`	method	Copies one blob to another. The portion of the source blob to be copied is between the indices "srcIndex" and "srcIndex" + "length". The portion of the destination blob that will be affected is between the indices "destIndex" and "destIndex" + "length". Both blobs must be sufficiently long, otherwise an exception will be thrown.
`void resize(int newLength)`	method	Resizes a blob preserving its current content (as much as possible).

3.5 Collections and objects

3.5.1 Collections

Collections are ways to store multiple values in a single variable. AddyScript offers six different types of collections: tuples, lists, sets, queues, stacks, and maps. Each of these types has a unique set of features and is suited for a particular usage scenario.

Tuples

A tuple is a collection in which items are accessed by index. You typically create a tuple using a tuple initializer. Once created, the contents of the tuple do not change, meaning that tuples are immutable. Items cannot be added, updated, or deleted, and the size of the tuple does not change during its lifetime. You can search for an item in a tuple using the **contains** operator or the "indexOf" and "lastIndexOf" methods. You can read the "size" property to get the number of items stored in the tuple. Here is an example of using a tuple in AddyScript.

Example

```
t = ('John Doe', 19, 'New York');

println($'t.size = {t.size}');
println('contents:');

for (var i = 0; i < t.size; ++i)
    println($'t[{i}] = {t[i]}');

println($'19 is at index {t.indexOf(19)}');
println($'Does t contain "New York"? {t contains "New York"}');
println($'Does t contain "Tokyo"? {t contains "Tokyo"}');
```

TUPLE API

The following tables summarize all the operators, properties, and methods provided by the AddyScript tuple API.

The **tuple** class supports the following operators:

Operator	Operands	Description
[index]	an integer	Gets the value of an item in the tuple. Negative indices are processed modulo the length of the tuple.
[lbound..ubound]	2 integers	Gets a slice (i.e. a sub-tuple) of the target tuple. Either "lbound" or "ubound" can be omitted. When "lbound" is omitted it's replaced with 0, a missing "ubound" will be replaced with the length of the tuple.
+	2 tuples	Concatenates two tuples.
*	A tuple on one side and an integer on the other side	Concatenates a tuple with itself the given number of times.
contains	a tuple to the left and anything to the right	Checks if the tuple contains the given value.
==	2 tuples	Checks that both tuples have the same length and contain equal items at each position.
!=	2 tuples	Checks that both tuples have different lengths or contain different items at least at one position.

In addition to the above operators, the **tuple** class exposes the following members:

Member	Nature	Description
`int size { read; }`	property	Gets the number of items currently stored in the tuple.
`any front { read; }`	property	Gets the first item of a tuple.
`any back { read; }`	property	Gets the last item of a tuple.
`int indexOf(any value, int start = 0, int count = 0)`	method	Gets the position of the first occurrence of an item in the given range of a tuple. Negative values of the optional "start" parameter are processed modulo the length of the tuple. Parameter "count" is ignored if it's negative or zero (that's the default). Returns -1 if the item is not found.
`int lastIndexOf(any value, int start = -1, int count = 0)`	method	Gets the position of the last occurrence of an item in the given range of a tuple. Negative values of the optional "start" parameter are processed modulo the length of the tuple. Parameter "count" is ignored if it's negative or zero (that's the default). Returns -1 if the item is not found.

Lists

Like a tuple, a list is a collection in which items are accessed by index. But on contrary of tuples, lists are not immutable. A list is a kind of dynamically sized array. You typically create a list using a list initializer. After creating a list, you can add new items to it using the "add" or "insert" methods. You can remove existing items from it using the "remove" and "removeAt" methods. You can search for an item in a list using the **contains** operator or the "indexOf" and "lastIndexOf" methods. The contents of the list can be sorted using the "sort" method. The "inverse" method returns a list with the same contents but with the items ordered in reverse order. To get the number of items currently stored in the list, simply read the "size" property and finally, to empty the list, call the "clear" method. Here is an example of using a list in AddyScript.

Example

```
n = (int) readln('How many names? ');
names = [];

for (i = 0; i < n; i++)
{
    print('Name number {0}: ', i + 1);
    names.add(readln());
}

println('You entered the following names: ' + names);

names = names.sort();
println('After sorting: ' + names);

someName = readln('Type a name: ');

if (names contains someName)
{
    i = names.indexOf(someName);
    println('{0} was found in the list at position {1}', someName, i);
    names.removeAt(i);
    println('After removal of ' + someName + ': ' + names);
}
else
{
    i = (int) readln('Enter a position in the list: ');

    if (i < names.size)
    {
        names.insert(i, someName);
        println('After insertion of {0} at position {1}: {2}', someName, i, names);
    }
    else
        println('Out of list boundaries!');
}

someName = readln('Type another name: ');

if (names.remove(someName))
    println('After removal of ' + someName + ': ' + names);
else
    println('Not found!');

names.clear();
println('After clearing the list: ' + names);
```

LIST API

The following tables summarize all the operators, properties, and methods provided by the AddyScript list API.

The list class supports the following operators:

Operator	Operands	Description
[index]	an integer	Gets or sets the value of an item in the list. Negative indices are processed modulo the length of the list.
[lbound..ubound]	2 integers	Gets a slice (i.e. a sub-list) of the target list. Either "lbound" or "ubound" can be omitted. When "lbound" is omitted it's replaced with 0, a missing "ubound" will be replaced with the length of the list.
+	2 lists	Concatenates two lists.
*	A list on one side and an integer on the other side	Concatenates a list with itself the given number of times.
contains	a list to the left and anything to the right	Checks if the list contains the given value.
==	2 lists	Checks that both lists have the same length and contain equal items at each position.
!=	2 lists	Checks that both lists have different lengths or contain different items at least at one position.

In addition to the above operators, the **list** class exposes the following members:

Member	Nature	Description
`int size { read; }`	property	Gets the number of items currently stored in the list.
`bool empty { read; }`	property	Returns **true** if the list has no item, returns **false** otherwise.
`any front { read; }`	property	Gets the first item of a list. Returns **null** if the list is empty.
`any back { read; }`	property	Gets the last item of a list. Returns **null** if the list is empty.
`void add(any value)`	method	Appends an item to the list.
`void insert(int position, any value)`	method	Inserts an item at the given position into the list. Negative values of the optional "start" parameter are processed modulo the length of the list.
`void insertAll(int position, list values)`	method	Inserts the items of the given collection at the given position into the list. Negative values of the optional "start" parameter are processed modulo the length of the list.
`int indexOf(any value, int start = 0, int count = 0)`	method	Gets the position of the first occurrence of an item in the given range of a list. Negative values of the optional "start" parameter are processed modulo the length of the list. Parameter "count" is ignored if it's negative or zero (that's the default). Returns -1 if the item is not found.
`int lastIndexOf(any value, int start = -1, int count = 0)`	method	Gets the position of the last occurrence of an item in the given range of a list. Negative values of the optional "start" parameter are processed modulo the length of the list. Parameter "count" is ignored if it's negative or zero (that's the default). Returns -1 if the item is not found.
`int bsearch(any value)`	method	Operates a binary search of the given value on a sorted list.
`bool remove(any value)`	method	Tries to remove an item from a list. Returns **true** on success, **false** otherwise.
`void removeAt(int position, int count = 1)`	method	Removes a range of items from the calling list. If "count" is negative or zero, it's automatically replaced by 1. An exception is thrown if either "position" or "position" + "count" is beyond the boundaries of the list.
`void clear()`	method	Empties a list.
`int frequencyOf(any value, int start = 0, int count = 0)`	method	Gets the number of occurrences of the a value in the given range of a list. Negative values of the optional "start" parameter are processed modulo the length of the list. Parameter "count" is ignored if it's negative or zero (that's the default)
`list sort(closure comparator = null)`	method	Returns a sorted clone of the target list using the given closure to compare items. If no parameter (or null) is passed, sorts the list following the logic of the "compareTo" method of each item.
`list inverse()`	method	Returns a list with the same content than the target list by with items sorted in reverse order.
`list sublist(int start, int count)`	method	Extracts a subset of the list delimited by the given boundaries.
`list unique()`	method	Gets a clone of the calling list in which each item is unique.
`map mapTo(list other)`	method	Creates and returns a map using the items of the calling list as keys and those of the other list as values. Both lists must have the same length.

Sets

A set is an emulation of the mathematical concept of a set. It is a kind of map in which we are only interested in the keys. Like lists and maps, you typically create a set using a set initializer. After that, you can add elements to it using the "add" method or get the number of elements stored by

reading the "size" property. The **contains** operator can be used to check the existence of a particular element in a set. To remove an element from a set, simply call the "remove" method. Finally, to empty a set, simply call its "clear" method. Here is an example of using a set in AddyScript.

Example

```
t = {'john', 'mike', 'bob'};
u = {'steve', 'mike', 'john'};

println('t = ' + t);
println('u = ' + u);
println('t + u = ' + (t + u));
println('t - u = ' + (t - u));
println('t & u = ' + (t & u));
println('t | u = ' + (t | u));
println('t ^ u = ' + (t ^ u));
println('(t + u) === (t | u) : ' + ((t + u) === (t | u)));
println();

v = {};
v.add('steve');

println('v = ' + v);
println('v < u : ' + (v < u));
println('v <= u : ' + (v <= u));
println('u < u : ' + (u < u));
println('t > v : ' + (t > v));
```

SET API

The following tables summarize all the operators, properties, and methods provided by the AddyScript set API.

The set class supports the following operators:

Operator	Operands	Description
+	2 sets	Computes and returns the union of both sets. Identical to operator |.
-	2 sets	Computes and returns the difference of both sets.
|	2 sets	Computes and returns the union of both sets. Identical to operator +.
&	2 sets	Computes and returns the intersection of both sets.
^	2 sets	Computes and returns the symmetric difference of both sets (i.e.: a and b being sets, a ^ b is equal to (a - b) + (b - a)).
==	2 sets	Verifies that both sets contains exactly the same elements, that is their symmetric difference is empty.
!=	2 sets	Verifies that there is at least one element in one set that does not belong to the other.
<	2 sets	Verifies that left operand is a proper subset of the right operand.
<=	2 sets	Verifies that left operand is a subset of the right operand.
>	2 sets	Verifies that left operand is a proper superset of the right operand.
>=	2 sets	Verifies that left operand is a superset of the right operand.
contains	a set to the left and anything to the right	Checks if the set contains the given value.

In addition to the above operators, the **set** class exposes the following members:

Member	Nature	Description
`int size { read; }`	property	Gets the number of elements currently stored in the set.
`bool empty { read; }`	property	Returns **true** if the set has no item, returns **false** otherwise.
`bool add(any value)`	method	Adds a value to the set. Returns **true** if the value was effectively added to set; returns **false** if the value was initially present in the set.
`bool remove(any value)`	method	Tries to remove a value from a set. Returns **true** on success, **false** otherwise.
`void clear()`	method	Empties a set.

Queues

A queue is a collection that implements the *first-in, first-out* design pattern. You will typically get a queue from a call to the **queue**::*of* static method. After that, you can add elements to it using the "enqueue" method or get the number of elements stored in it by reading the "size" property. To extract an element from a queue, use the "dequeue" method. The "front" property does the same thing, except that it does not remove the element from the queue. Finally, to empty a queue, simply call its "clear" method.

QUEUE API

The following table summarizes all the properties and methods provided by the AddyScript queue API.

Member	Nature	Description
`int size { read; }`	property	Gets the number of elements currently stored in the queue.
`bool empty { read; }`	property	Returns **true** if the queue has no item, returns **false** otherwise.
`any front { read; }`	property	Gets the oldest value of a queue without removing it. Throws an exception if the queue is empty.
`void enqueue(any value)`	method	Adds a value to the queue.
`any dequeue()`	method	Extracts and returns the oldest value in the queue.
`void clear()`	method	Empties a queue.

Stacks

A stack is a collection that implements the *last-in, first-out* design pattern. You will typically get a stack as a result of a call to the **stack**::*of* static method. After that, you can add elements to it using the "push" method or get the number of elements stored in it by reading the "size" property. To pop an element off the stack, use the "pop" method. The "top" property does the same thing, but it does not remove the element from the stack. Finally, to empty a stack, simply call its "clear" method.

STACK API

The following table summarizes all the properties and methods provided by the AddyScript stack API.

Member	Nature	Description
`int size { read; }`	property	Gets the number of elements currently stored in the stack.
`bool empty { read; }`	property	Returns **true** if the stack has no item, returns **false** otherwise.
`any top { read; }`	property	Gets the value on top of a stack without removing it. Throws an exception if the stack is empty.
`void push(any value)`	method	Adds a value to the stack.
`any pop()`	method	Pops a value from the stack.
`void clear()`	method	Empties a stack.

Maps

A map is a collection of key-value pairs. The key is used as an index to add, retrieve, and update values in the map. So, a map can be thought of as a list where the indices are neither necessarily integers nor necessarily contiguous. Similar to lists, you typically create a map using a map initializer. After that, you can get the number of key-value pairs stored in the map by reading the "size" property. The **contains** operator can be used to check the presence of a particular key in the map. The "containsValue" method on the other hand is used to check the existence of a particular value in the map. To remove a pair, simply call the "remove" method. The "keys" and "values" properties are used to retrieve all the keys and all the values in a map, respectively. Note that both methods return sets. So, if a value appears twice in a map, it will be represented only once in the set returned by the "values" property. The "entries" property returns a set of all the key-value pairs of a map, each pair being represented as a tuple. To get all the keys associated with a particular value in a map, call its "keysOf" method with that value as a parameter. The "frequencyOf" method on the other hand simply tells how many distinct keys a value is associated with. So `someMap.frequencyOf(someValue)` is equivalent to

`someMap.keysOf(someValue).size`. The "inverse" method is used to create a map in which the key-value pairs are inverse to those in the calling map. Finally, to make a map empty, simply call its "clear" method. Here is an example of using the map in AddyScript.

Example

```
tom = {'name' => 'Tom Berenger', 'job' => 'Lawyer', 'age' => 38};
tom['company'] = 'Holy Lawyers & co.';
tom['hire date'] = `2004-05-18`;
tom['salary'] = 3600D;

foreach (prop => value in tom)
    println('{0} : {1}', prop, value);

someProperty = readln('Type the name of a property: ');

if (tom contains someProperty)
    println('Property ' + someProperty + ' is already defined for tom!');
else
{
    tom[someProperty] = readln('Enter a value for ' + someProperty + ': ');
    println('After adding that property:');
    foreach (prop in tom.keys)
        println('{0} : {1}', prop, tom[prop]);
}

someProperty = readln('Type the name of another property: ');

if (tom.remove(someProperty))
{
    println('After removal of ' + someProperty + ':');
    foreach (value in tom)
        println('{0} : {1}', __key, value);
}
else
    println('Property ' + someProperty + ' is not defined for tom!');

someValue = readln('Type any value: ');

if (tom.containsValue(someValue))
{
    println(someValue + ' is the value of ' + tom.frequencyOf(someValue) + ' properties of tom');
    print('Those are :');
    foreach (prop in tom.keysOf(someValue))
        print(prop);
    println();
}
else
    println('No property of tom has value ' + someValue);
```

MAP API

The following tables summarize all the operators, properties, and methods provided by the AddyScript map API.

The map class supports the following operators:

Operator	Operands	Description
[key]	a value of any type	Gets or sets the value attached to the given key in the map.
+	2 maps	Merges both map in a single one. It fails if both maps have a key in common.
==	2 maps	Checks that both maps contain equal key-value pairs.
!=	2 lists	Checks that one of the maps has at least one key-value pair that the other map does not have.
contains	a map to the left and anything to the right	Checks if the map contains a pair with a key equal to the given value.

In addition to the above operators, the **map** class exposes the following members:

Member	Nature	Description
`int size { read; }`	property	Gets the number of key-value pairs currently stored in the map.
`bool empty { read; }`	property	Returns **true** if the map has no key-value pair, returns **false** otherwise.
`set keys { read; }`	property	Gets a set of all the keys of a map.
`set values { read; }`	property	Gets a set of all the distinct values of a map.
`set entries { read; }`	property	Gets a set of tuples representing all the key-value pairs of a map.
`bool containsValue(any value)`	method	Checks if the map contains the given value.
`any get(any key, any defaultValue)`	method	Tries to get the value that's associated with the given *key* from the map. Returns the supplied *defaultValue* if no value is associated with the given *key* in the map.
`bool update(any key, any value)`	method	Updates the value that's associated with the given *key* in the map. Returns **true** if a value was effectively updated, returns **false** otherwise.
`void add(any key, any value)`	method	Adds a new key-value pair to the map. Throws an exception if the supplied key was already present in the map.
`map apply(any key, closure action)`	method	Invokes the given *action* on the value that's associated with the given *key* in a map if any. Returns the map itself upon completion thus allowing chained calls.
`set keysOf(any value)`	method	Gets a set of all the keys of a map that are bound to a particular value.
`int frequencyOf(any value)`	method	Gets the number of all the keys of a map that are bound to a particular value. This is also the number of occurrences of a value in the map.
`bool remove(any key)`	method	Tries to remove a key-value pair from a map (the one that has the given key if any). Returns **true** on success, **false** otherwise.
`void clear()`	method	Empties a map.
`map inverse()`	method	Creates and returns a map in which key-value pairs are inverse of those of the calling one.

Iteration Methods

In addition to the members listed in the tables above, collection classes also have methods that can be used to iterate over their instances while performing an action, evaluating a predicate, or collecting a summary value. Such a method is generally equivalent to a loop, except that it has a more compact syntax and can appear anywhere an expression is expected, which loops cannot do. Most of these methods return the instance on which

they are invoked, which allows for call chaining. The following table summarizes AddyScript iteration methods and the classes they belong to. One of them, the "times" method of the **int** and **long** types, does not belong to a collection class but has similar behavior.

Method	Description	Example
`int int::times(closure action)` `long long::times(closure action)`	Performs the given action n times where n is the integer value on which the method is invoked.	`4.times(|i| => println('hello!'));`

`string string::each(closure action)`
`blob blob::each(closure action)`
`tuple tuple::each(closure action)`
`list list::each(closure action)`
`set set::each(closure action)`
`queue queue::each(closure action)`
`stack stack::each(closure action)`
`map map::each(closure action)`
	Performs the given action on each item (each character for strings, each byte for blobs, or each key-value pair for maps) of the target object.
	When the target object is a map, the closure expects two arguments.
	In any other case, it expects a single argument.
		`l = [4, 2, 5, 3, 8, 6];`
		`sum = 0;`
		`l.each(|x| => sum += x);`
		`println(sum);`

`tuple tuple::eachIndex(closure action)`
`list list::eachIndex(closure action)`
	Performs the given action on each index of the target tuple or list.	`l = [4, 2, 5, 3, 8, 6];`
		`l.eachIndex(|i| => l[i] *= 2);`
		`println(', '.join(..l));`

`map map::eachKey(closure action)`	Performs the given action on each key of the target map.	`m = {'age' => 30, 'weight' => 80, 'height' => 170};` `m.eachKey(|k| => println(k + ': ' + m[k]));`
`map map::eachValue(closure action)`	Performs the given action on each distinct value of the calling map.	`m = {'age' => 70, 'weight' => 70, 'height' => 180};` `m.eachValue(|v| => println(v + ': ' + m.keysOf(v)));`

`bool tuple::all(closure predicate)`
`bool list::all(closure predicate)`
`bool set::all(closure predicate)`
	Evaluates a predicate on each item of a tuple, list, or set.
	Returns **true** if the predicate is true for all items;
	returns **false** otherwise.
		`s = {4, 2, 0, 8, 6};`
		`b = s.all(|e| => e % 2 == 0);`
		`if (b) println('all them are even');`

`bool tuple::any(closure predicate)`
`bool list::any(closure predicate)`
`bool set::any(closure predicate)`
	Evaluates a predicate on each item of a tuple, list, or set.
	Returns **true** if the predicate is true for at least one of them;
	returns **false** otherwise.
		`s = {4, 1, 2, 0, 3, 8, 6};`
		`b = s.any(|e| => e % 2 == 1);`
		`if (b) println('there is an odd one');`

`any tuple::first(closure predicate)`
`any list::first(closure predicate)`
`any set::first(closure predicate)`
	Successively evaluates a predicate on each item of a tuple, list, or set,
	returning the first item for which the predicate evaluates to true.
		`l = [4, 0, 2, 5, 3, 7, 1, 8, 6];`
		`a = l.first(|x| => x % 2 == 1);`
		will return 5

`any tuple::last(closure predicate)`
`any list::last(closure predicate)`
	Traverse a tuple or a list backwards by successively evaluating a predicate on each of its items,
	returning the first item for which the predicate evaluates to true.
		`l = [4, 0, 2, 5, 3, 7, 1, 8, 6];`
		`a = l.last(|x| => x % 2 == 1);`
		will return 1

`int tuple::findIndex(closure predicate)`
`int list::findIndex(closure predicate)`
	Finds the index of the first item of a tuple or list that satisfies the given predicate.	`l = [4, 0, 2, 5, 3, 7, 1, 8, 6];`
		`a = l.findIndex(|x| => x % 2 == 1);`
		will return 3

`any tuple::findLastIndex(closure predicate)`	Finds the index of the last item of a list that satisfies the given predicate.	`l = [4, 0, 2, 5, 3, 7, 1, 8, 6];` `a = l.findLastIndex(|x| => x`

Method	Description	Example
`any list::findLastIndex(closure predicate)`		`% 2 == 0);` will return 8

`list list::where(closure predicate)`
`set set::where(closure predicate)`
	Filters the contents of a tuple, list, or set based on the given predicate.	`l = [4, 0, 2, 5, 3, 7, 1, 8, 6];`
		`a = l.where(|x| => x % 2 == 1);`
		will return [5, 3, 7, 1]

`list list::select(closure transform)`
`set set::select(closure transform)`
	Maps each item of a tuple, list, or set to the value returned by the given closure.	`s = {'nadia', 'dave', 'roland', 'rick', 'john'};`
		`t = s.select(|x| => x.toUpper());`
		returns

`any tuple::aggregate(any seed, closure aggregator)`
`any list::aggregate(any seed, closure aggregator)`
`any list::aggregate(any seed, closure aggregator)`
	Generate a single value by aggregating the items of a tuple, list or set;
	the *aggregator* is a function that takes 2 arguments: an *accumulator* and the current item;
	it generates the next value of the *accumulator*;
	parameter *seed* is the initial value of the *accumulator*;
	aggregate returns the last value of the *accumulator*.
		`l = [4, 0, 2, 5, 3, 7, 1, 8, 6];`
		`sum = l.aggregate(0, |acc, val| => acc + val);`
		will return 36

`map list::groupBy(closure criterion)`	Groups the items of a list according to the given criterion and
	returns a map of sub-lists identified by the distinct group identifiers that were produced by the criterion.
		`l = [4, 0, 2, 5, 3, 7, 1, 8, 6];`
		`g = l.groupBy(|x| => x % 2);`
		will return

Note:

For iteration methods that are invoked on strings, blobs, tuples, or lists, the closure argument can optionally take a second parameter that will receive the value of the index of the current element on each iteration.

Example

```
l = ['one', 'two', 'three', 'four', 'five'];
l.each(|x, i| => println('Item #' + i + ' : ' + x));
ranks = l.groupBy(|x, i| => i % 2 ? 'odd rank' : 'even rank');
println('Items at even or odd ranks: ' + ranks);
evenItems = l.where(|x, i| => i % 2 == 0);
println('Items at even ranks: ' + evenItems);
concatOddItems = l.aggregate('', |acc, val, ind| => ind % 2 == 1 ? acc + val : acc);
println('Items at odd ranks concatenated: ' + concatOddItems);
```

3.5.2 Objects

Just like a collection, an object is another way to store multiple values in a single variable. These values are then called the fields of the object. The fields of the object are accessed by their name, using the dot syntax: *objectName.fieldName*.

Creating an object

There are 2 ways to create objects in AddyScript: by using an initializer or by invoking a constructor. We will talk about constructors in the chapter dedicated to object-oriented programming. For now, let's just talk about object initializers.

To create an object using an initializer, simply use this syntax:

```
varname = object_initializer;
```

Where:

- An object initializer is a comma-separated list of field assignments, enclosed in curly braces and preceded by the keyword **new**.
- A field assignment consists of an identifier (the field name), followed by an equals sign (`=`), and then an expression (the initial value of the field).
- If the equals sign and the expression are omitted, the field will be initialized to a variable of the same name, which must be defined in the object's initialization context.

Example

```
actor = new { firstName = 'John', lastName = 'Snow', age = 24 };
movie = new { title = 'The Matrix', year = 1999, rating = 8.5, actor };
println('{0} {1} is aged {2}', actor.firstName, actor.lastName, actor.age);
println('{0}, released in {1} is rated {2}; it\'s main actor is {3} {4}',
        movie.title, movie.year, movie.rating, movie.actor.firstName, movie.actor.lastName);
```

Output

```
John Snow is aged 24
The Matrix, released in 1999 is rated 8,5; it's main actor is John Snow
```

Sometimes it is better to initialize the object with an empty initializer and then add fields to it as needed.

Example

```
student = new {};
student.firstName = 'André';
student.lastName = 'Dikos';
student.age = 19;

println('{0} {1} is aged {2}', student.firstName, student.lastName, student.age);
```

Output

```
André Dikos is aged 19
```

Finally, it's worth mentioning that AddyScript can convert a map to an object and vice versa. In this case, if one of the map's keys isn't a valid identifier, the corresponding field in the output object will be accessible via a special identifier—that is, an identifier that starts with the dollar sign (`$`) or contains hexadecimal escape sequences. The dollar sign allows you to create identifiers that correspond to keywords or numeric values. Escape sequences, on the other hand, allow you to represent special characters contained within identifiers. **e.g.:** `$1` , `$if` , or `for\x20each` (respectively derived from `1` , `if` , and `for each`).

Example

```
dict = { 'long' => 120, '2way' => 80, 'depth in cm' => 20 };
shape = (object) dict;
println('Shape size: {0} x {1} x {2}', shape.$long, shape.$2way, shape.depth\x20in\x20cm);
```

Output

```
Shape size: 120 x 80 x 20
```

Object destructuring

In AddyScript, it's possible to initialize multiple variables simultaneously by assigning them values from an object's properties. This mechanism is called **object destructuring**. Object destructuring is performed by writing an assignment where the *lvalue* is a list of variable names enclosed in curly braces, and the *rvalue* is an expression that evaluates to an object. Such an assignment requires the use of the **let** keyword (otherwise, AddyScript will interpret the opening curly brace as the beginning of a block of instructions).

Here is an example of object destructuring:

```
person = new { firstName = 'Mael', lastName = 'Jordano', age = 25 };
let { firstName, lastName, age } = person;
println($'{firstName} {lastName} is a {age}-year-old person');
```

Output

```
Mael Jordano is a 25-year-old person
```

In the syntax above, each variable in the list within the curly braces must correspond to a property of the same name in the source object. To prevent a runtime error if no property of the same name is found in the source object, a default value can be assigned to the target variable. The default value is ignored if a property of the same name is found in the source object.

Illustration:

```
person = new { firstName = 'Mael', lastName = 'Jordano', age = 25 };
let { firstName, lastName, job = 'Journalist', age = 17 } = person;
println($'{firstName} {lastName} is a {age}-year-old {job}');
```

Output

```
Mael Jordano is a 25-year-old Journalist
```

Objects can be destructured recursively: if the source object has a property that is also an objet it can be recursively destructured as illustrated below.

```
person = new { firstName = 'Mael', lastName = 'Jordano', age = 25, job = new { title = 'Accountant', company = 'Paradise Co.', since = `2018-08-12` } };
let { firstName, lastName, age, { title, company } = job } = person;
println($'{firstName} {lastName} is a {age}-year-old {title} at {company}');
```

Output

```
Mael Jordano is a 25-year-old Accountant at Paradise Co.
```

If the name of a variable is preceded by the *spread* operator (`..`), it will be used to collect the remaining properties of the source object (those that were not explicitly extracted).

```
person = new { firstName = 'Mael', lastName = 'Jordano', age = 25, job = new { title = 'Accountant', company = 'Paradise Co.', since = `2018-08-12` } };
let { firstName, lastName, age, { title, ..rest } = job } = person;
println($'{firstName} {lastName} is a {age}-year-old {title} at {rest.company} since {rest.since:d}');
```

Output

```
Mael Jordano is a 25-year-old Accountant at Paradise Co. since 2018-08-12
```

Manipulating objects

Well, there is no special manipulation on objects, other than storing values in their fields and retrieving them later. Advanced object manipulation is a concern of object-oriented programming.

3.6 Functions

3.6.1 Inner functions

At the time of writing this manual page, AddyScript has a set of 45 predefined functions. We have already discovered some of them in the previous sections. Here is a more general overview of AddyScript's built-in functions:

Utility functions

- `any eval(string expression)` : evaluates the expression contained in the given string and returns its value.
- `int hash(any first, ..more)` : combines the provided values to calculate their hash code. It accepts a variable number of arguments. Only the first argument is required; the others are optional.
- `void sleep(int milliseconds)` : pauses the script's execution for the specified number of milliseconds.
- `void exit(int code = 0)` : Exits the program with the given status code. **Note**: This will also terminate the host application.

Conversion functions

- `string chr(int ascii)` : gets the Unicode character corresponding to the given code. The returned value is of type string.
- `int ord(string chr)` : gets the Unicode order of the first character of the given string. In fact, the string must be one character long.
- `blob pack(string format, ..values)` : packs several values in a binary string (a **blob**): a way to create structured data items in preparation for a call to a .NET method or a native function that expects a structured argument.
- `tuple unpack(string format, blob bytes)` : unpacks the combined values into a binary string following the given format.

Math functions

- `float rand()` : generates a random **float** between `0.0` and `1.0`.
- `int randint(int min, int max = 0)` : generates a random **int** between *min* (inclusive) and *max* (exclusive). If *max* is omitted, the random value will be generated between `0` and *min*. In all cases, the limits are always sorted so that *max* is greater than or equal to *min*.
- `float|complex sin(float|complex x)` : computes the sine of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float|complex cos(float|complex x)` : computes the cosine of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float|complex tan(float|complex x)` : computes the tangent of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float|complex asin(float|complex x)` : computes the arc sine of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float|complex acos(float|complex x)` : computes the arc cosine of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float|complex atan(float|complex x)` : computes the arc tangent of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float atan2(float y, float x)` : computes the arc tangent of y/x.
- `float|complex sinh(float|complex x)` : computes the hyperbolic sine of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float|complex cosh(float|complex x)` : computes the hyperbolic cosine of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float|complex tanh(float|complex x)` : computes the hyperbolic tangent of `x`. Returns a **complex** if `x` is a **complex**, a **float** otherwise.
- `float deg2rad(float x)` : converts x from degrees to radians.
- `float rad2deg(float x)` : converts x from radians to degrees.
- `float|complex log(float|complex x)` : computes the natural logarithm of `x`.
- `float|complex log10(float|complex x)` : computes the base-10 logarithm of `x`.
- `float|complex log2(float|complex x, float base = 2)` : computes the logarithm of x to the given base. Returns a **complex** if `x` is a **complex**, a **float** otherwise. Base-2 is assumed by default.
- `float|complex exp(float|complex x)` : computes the exponential of `x`.

- `float|complex sqrt(any x)` : calculates the square root of `x`. Returns a **complex** if `x` is a **complex** or a negative value. Returns a **float** otherwise.
- `int sign(any x)` : determines the sign of `x`. Returns -1 for negative, 0 for 0, and 1 for positive.
- `any abs(any x)` : determines the absolute value of `x`. Returns a value of the same type as the argument.
- `any min(first, second, ..more)` : determines the minimum of two or more values.
- `any max(first, second, ..more)` : determines the maximum of two or more values.
- `float|decimal trunc(float|decimal x)` : truncates `x` to its integer part. Returns a value of the same type as the argument.
- `float|decimal floor(float|decimal x)` : determines the floor of `x` (i.e.: the largest integer that's less than or equal to `x`). Returns a value of the same type as the argument.
- `float|decimal ceil(float|decimal x)` : determines the ceiling of `x` (i.e.: the smallest integer that is greater than or equal to `x`). Returns a value of the same type as the argument.
- `float|decimal round(float|decimal value, int precision = 0)` : gets *value* rounded to *precision* decimal digits. The result is of the same type as the first argument.

Factory functions

- `date now()` : gets the current date and time.
- `duration days(float value)` : creates a duration with the given number of days.
- `duration hours(float value)` : creates a duration with the given number of hours.
- `duration minutes(float value)` : creates a duration with the given number of minutes.
- `duration seconds(float value)` : creates a duration with the given number of seconds.
- `duration milliseconds(float value)` : creates a duration with the given number of milliseconds.
- `string format(string pattern, ..substitutions)` : interpolates a string with the given pattern and values. using a mutable string is equivalent to invoking this function.

I/O functions

- `void print(string pattern, ..substitutions)` : prints a formatted message to standard output and stays on the same line.
- `void println(string pattern = '', ..substitutions)` : prints a formatted message to standard output and moves to the next line.
- `string readln(string prompt = '')` : reads a string from the standard input device and returns it to the script. An optional prompt can be displayed.

Other functions

While using the interactive console *(asis)*, you can also use the `void source(string path)` function to load a script from a file and execute it.

3.6.2 User-defined functions

In addition to the predefined functions, users can define their own functions in AddyScript and use them. The following sections describe how to do this.

Creating a function

To create a function in AddyScript, use the following syntax:

```
function functionName (comma_separated_list_of_parameters)
{
    // function's logic goes here
    // returning a value is optional and can be done like this:
    return 10;
}
```

Example

```
function sayHello()
{
    println("Hello to anyone!");
}

function sayHelloTo(name)
{
    println("Hello " + name);
    return readln("How do you do? ");
}

// Now we can call those 2 functions like this:

sayHello();

name = readln("What's your name? ");
ans = sayHelloTo(name);

if (ans != "fine")
    println("What's the matter?");
```

When the body of a function is reduced to a **return** statement or a simple expression, the entire function can be formulated with this shorter syntax:

```
function functionName (comma_separated_list_of_parameters) => expression;
```

Example

```
function addTwo(a, b) => a + b;

n = (float)readln('first number: ');
m = (float)readln('second number: ');
res = addTwo(n, m);
println($'the result is: {res}');
```

Invoking functions

As in most languages, a function invocation in AddyScript consists of its name followed by a comma-separated list of arguments in parentheses. In AddyScript, however, arguments can be either positional or named.

A positional argument is any expression that appears at a particular position in an argument list. It is automatically mapped at runtime to the parameter that appears at the same position in the function header.

A named argument is one that consists of a name (i.e., an identifier) followed by a colon (`:`) and then an expression. It is mapped to the parameter that has the same name in the function header. Named arguments are particularly useful when you are calling a function that has optional parameters and you do not want to provide values for some of the ones that come first in the function header.

Positional arguments must always come first in an argument list. Once the parser encounters a named argument, it expects all following arguments to be named as well.

Example

```
// A function that concatenates values and wrap between a prefix and a suffix
function concat(values, separator = ', ', prefix = '{', suffix = '}')
    => prefix + separator.join(..values) + suffix;

// A list to concatenate:
l = [8, 3, 4, 9, 2, 4, 6, 0, 7];

// Calling concat with default values for separator, prefix and suffix:
s1 = concat(l);
println('s1 = ' + s1);

// Calling concat with explicit values for separator, prefix and suffix:
// All arguments are positional
s2 = concat(l, '-', '[', ']');
println('s2 = ' + s2);

// Calling concat with an explicit value for separator and default values for prefix and suffix:
// All arguments are positional
s3 = concat(l, '; ');
println('s3 = ' + s3);

// Calling concat with explicit values for prefix and suffix:
// The prefix and suffix arguments are named
s4 = concat(l, prefix: '(', suffix: ')');
println('s4 = ' + s4);

// Calling concat with explicit values for separator, prefix and suffix:
// All arguments are named and given in random order
s5 = concat(suffix: '[', separator: ':', values: l, prefix: ']');
println('s5 = ' + s5);
```

SPREADING ARGUMENTS

An important feature of the AddyScript syntax is that you can invoke a function with arguments of type **list** or **set** preceded by the **spread operator** (`..`). This tells AddyScript that the **list** or **set** should be substituted for its contents. This works similarly to the right operand of group assignment. The only requirement here is that none of the parameters provided by the spread collection should be passed to the function by reference.

Example

```
function add(a, b, c) => a + b + c;

res = add(1, 2, 3);
println('add(1, 2, 3)' + res);
// Output: 6

l = [5, 6, 7];
res = add(..l);
println('add(..l)' + res);
// Output: 18

s = {9, 10};
res = add(..s, 11);
println('add(..s, 11)' + res);
// Output: 30

res = add(..l[2..], ..s);
println('add(..l[2..], ..s)' + res);
// Output: 26
```

Manage how parameters are passed to a function

A parameter can be passed to a function by value, by reference, or as a variable-length list of values. By default, parameters are passed by value. To indicate that a parameter is passed by reference, simply prefix it with the ampersand sign (`&`). Similarly, prefixing a parameter with the double-dot sign (`..`) indicates that it represents a variable-length list of values. Note that a variable-length list of values must always be the last in a parameter list. Thus, a function cannot have multiple variable-length lists of values in its header.

When a parameter is passed by value to a function, it can be assigned a default value. This makes the parameter optional (i.e., it does not need to be provided with a value when calling the function). Once you add an optional parameter to a parameter list, the only types of parameters that can follow are other optional parameters and a variable-length list of values.

Another feature that AddyScript offers in handling function parameters is emptiness checking. When a parameter name is followed by an exclamation mark (`!`) in a function header, this tells AddyScript that the parameter in question should not receive empty values. Empty here means a null reference, a zero-length string, or an empty collection. Whenever the parameter receives such a value, an exception is thrown.

Closures

A closure is a function used as a variable. Closures are typically used to pass functions as parameters to other functions (customizing their behavior) or to return a function as the result of another function. They appear in two forms in AddyScript: function references and inline function declarations. A function reference is like a reference to a variable (a simple identifier in the code) while an inline function declaration is an anonymous function definition that appears where an expression was expected. Both techniques are illustrated in the example below:

Example

```
// Repeats an action on each item of a list; parameter 'action' is a (mandatory) closure
function repeat(l, action!)
{
    foreach (item in l)
        action(item);
}

// A list for testing purpose
myList = [2, 5, 7, 8, 3, 0, 1, 6, 9, 4];

// Invoke repeat with a reference to the builtin 'println' function
repeat(myList, println);

// Declare a function inline and store it in a variable called myFunc
myFunc = function (n) {
    println('{0} x 2 = {1}', n, 2 * n);
};

// Invoke repeat with myFunc
repeat(myList, myFunc);

// Something more compact
repeat(myList, function (x) {
    println(x % 2 == 0 ? 'even' : 'odd');
});
```

REMARKS:

1. When the body of an anonymous function is reduced to a return statement or a simple expression, the entire function can be formulated like this: `|parameters| => expression`. In this form, we call it a **lambda expression**. For example, we could invoke the "repeat" function from the previous example like this: `repeat(myList, |n| => println('{0} x 2 = {1}', n, 2*n));`. A lambda expression can also have a real function body delimited by curly braces and optionally ending with a **return** statement.
2. If the parameter list of a lambda expression is empty, put a space between the vertical bars. This prevents parsers from confusing them with an or-else operator (`||`).

THE CLOSURE'S "BIND" METHOD

The **closure** type has a single member: the "bind" method. Its prototype is: `closure closure::bind(string parameterName, any defaultValue)`. Its main purpose is to create a clone of a closure with a modified prototype. The "bind" method operates as follows:

1. If *parameterName* matches the name of an existing parameter in the original function's header, "bind" proceeds to **currying**: The parameter *parameterName* is removed from the resulting function's header and is replaced by a fixed value, which in this case is *defaultValue*. The resulting function will then have one parameter less than the original one, with the exact same body that will always evaluate *parameterName* to *defaultValue*. Here is an example of how to curry a function:

```
// add is the original function: it simply adds two numbers
function add(a, b) => a + b;

// add10 is a variant of add that always use 10 for parameter a
add10 = add.bind('a', 10);

println(add10(5));
println(add10(-7));
```

Output

```
15
3
```

2. If *parameterName* doesn't match the name of an existing parameter in the original function's header, "bind" simply adds an optional parameter with the given name and default value to the resulting function. This is very helpful when you are creating a function that expects a closure as an argument and that you want the closure to match variable prototypes. As an example, the "each" method of the **list** type is defined as follows:

```
public function each(action)
{
    // action is replaced by a clone that has an optional parameter named __index
    action = action.bind('__index', null);

    foreach (__index => __value in this)
        action(__value, __index);

    return this;
}
```

External functions

AddyScript allows a script to invoke a function declared in a native library (such as a DLL or a shared object). To do this, the target function must first be declared as an external function in the script using this syntax:

```
[LibImport("nativeLibraryName", procName = "importedFunctionName", returnType = "someDotNetType")]
extern function functionName(list_of_parameters_with_type_attribute);
```

Where :

- *nativeLibraryName* is the name of the native library that contains the definition of the function we want to import,
- *importedFunctionName* is the name of the function we want to import, and
- *someDotNetType* is the name of the return type of the function.
- *functionName* is the name we want to give to the function in our code.

Notes:

1. If *functionName* is equal to *importedFunctionName*, then the **procName** field of the **LibImport** attribute can be omitted.
2. If the function does not return anything, then **returnType** can also be omitted.
3. Each parameter in the formal parameter list must be decorated with a **Type** attribute indicating what type the parameter is.
4. If the **Type** attribute is omitted, the *System.Object* type will be used by default.
5. In fact, when it comes to P/Invoke, AddyScript is less dynamically typed than usual.
6. Unqualified type names will be prefixed with "System.", and will therefore be searched in the *System.Private.CorLib* assembly.

The following example shows how to invoke the Win32 *MessageBox* function from a script:

External function demo

```
[LibImport("user32", procName = "MessageBox", returnType = "Int32")]
extern function msgbox(
        [Type("IntPtr")] hWnd,
        [Type("String")] message,
        [Type("String")] title,
        [Type("Int32")] flags);

// Now we can call the imported function like this:
res = msgbox(null, "Hello funny people!", "AddyScript", 0);
println("MessageBox returned: " + res);
```

Re-using code: the import directive

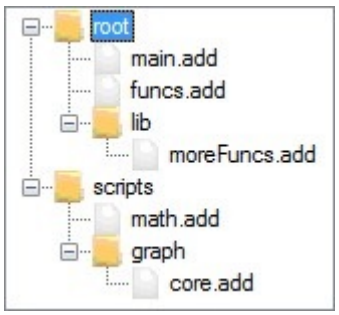
AddyScript obviously allows the user to define a function once and reuse it multiple times later. To do this, simply save the target functions in a script and import that script from another. You typically import a script using the **import** directive. Its syntax is as follows:

```
import script_name_without_extension;
```

AddyScript assumes that imported scripts have the extension *.add*. The imported script will first be searched in the same directory as the script from which it is imported. If it is not found in that directory, AddyScript will continue searching in each of the directories that are listed in the *ImportPaths* property of the *ScriptContext* instance with which the current *ScriptEngine* object was initialized. An error occurs if the search is unsuccessful. To indicate that the script to be imported would be in a subdirectory, use the double-colon operator (::) as a path separator in the script name.

Example:

Suppose that you have the following file structure:



To import *funcs.add* from *main.add*, simply add the following line of code to *main.add*:

```
import funcs;
```

To import *moreFuncs.add* from *main.add*, simply add the following line of code to *main.add*:

```
import lib::moreFuncs;
```

Supposing that the full path to *scripts* figures in the *ImportPaths* property of the current *ScriptContext*, we can import *math.add* from *main.add* by simply adding the following line of code to *main.add*:

```
import math;
```

And finally, *scripts* being in the *ImportPaths*, we can import *core.add* from *main.add* by simply adding the following line of code to *main.add*:

```
import graph::core;
```

REMARKS:

The import directive can be used to import any symbol defined in a script. These include constants, variables, functions and classes.

3.7 Object oriented programming

3.7.1 Classes

Declaring a class

Declaring a class in AddyScript is straightforward. Simply use this syntax:

```
[modifier] class className [: superClassName]
{
    classMembers
}
```

Where

- *modifier* is one of the keywords **final**, **static** and **abstract**.
- *className* is the name you want to give your class.
- *superClassName* is the name of a previously defined class.
- *classMembers* is a series of field, property, method, operator, and/or event definitions.

Note: The parts in brackets are optional, so you can omit them.

Defining class members

Generally speaking the definition of a class member follows this syntax:

`[scope] [modifier] specification`

Where

- *scope* is one of the keywords **private**, **protected** and **public**. If no scope is defined, **private** is applied by default.
- *modifier* has the same meaning as for a class. It must therefore be **final**, **static** or **abstract**. However, the **abstract** modifier is reserved for properties and methods, it cannot be applied to fields or events. A field can be both **static** and **final**, such a field is a class constant whose default value must be specified.
- The *specification* varies depending on the type of member defined.

SPECIFYING A FIELD:

A field specification is simply the name of the field, optionally followed by an equal sign, and its initial value.

Note: You do not have to explicitly declare the fields of a class. Objects are dynamic in AddyScript, so you can add fields to an object as you wish. However, declaring fields allows you to control how they are shared between instances of the class. It also allows you to control their visibility: whether they are accessible outside the class or not. It also ensures that these fields are discovered by any code that introspects the class.

SPECIFYING A METHOD:

The specification of a method is just a function declaration embedded into the class body.

SPECIFYING A PROPERTY:

A property is a pair of methods used to read and/or write the value of a field. AddyScript provides a dedicated syntax for defining the properties of a class. The specification of a property follows this syntax:

```
property property_name
{
    [scope] read
    {
       // statements
    }
    [scope] write
    {
       // statements
    }
 }
```

Where

- *property_name* is the name you want to give to the property.
- The block prefixed by **read** is called the **read accessor** (or simply **reader**). This is actually the body of the method that will be used to read the value of the backing field. It usually ends with a parameterized **return** statement.
- The block prefixed by **write** is called the **write accessor** (or simply **writer**). This is the body of the method that will be used to write (update) the value of the backing field. In this block, the script can refer to a special variable called **__value**. This is the value assigned to the property.
- Each of these accessors can be omitted, but not both at the same time.
- One of the accessors can be defined in a different scope than the property itself: for example, a public property can have a private writer. Such a property can only be updated by the class itself.
- Neither **read**, nor **write**, nor **__value** are AddyScript keywords. They are simply identifiers that have a special meaning in a particular context (let's say they are **contextual keywords**).
- Since both reader and writer are functions, they can also be reduced to an arrow followed by an expression when they consist of just an expression or a parameterized **return** statement.
- When the property is read-only (i.e.: it has no writer), and that its reader does nothing more than return a value, the entire property can be declared with the following syntax: `property property_name => returned_value;`. That syntax is equivalent to `property property_name { read => returned_value; }`.

Automatic properties:

Since defining a property consists of declaring a backing field, then defining the property's read and write accessors, AddyScript can help the programmer save time by doing most of the work automatically. To this end, the scripting language supports a shorter syntax for the definition of a property:

```
property property_name { [scope] read; [scope] write; }
```

As you can see, with this syntax, accessors are reduced to the contextual keywords **read** and **write** optionally preceded by a scope. For such a property, the scripting engine automatically generates a backing field as well as the accessor's logic. The content between braces can even be omitted if no accessors have a different scope than the property itself, resulting in something as short as: `property property_name;` which is equivalent to `property property_name { read; write; }`.

Remark: The specification of an automatic property is identical to that of an abstract property except that the scripting engine doesn't generate any logic for an abstract property. It expects concrete subclasses to do so.

Semi-automatic properties:

A property can have one of its accessors defined automatically, while the other one is defined manually. This can be useful when one of the accessors is trivial while the other one requires more complex logic. Such a property is called a **semi-automatic property**. The scripting engine will generate the backing field for it. An auto-generated backing field is always private and has the same name as the property itself prefixed by a double underscore (`__`). The syntax for defining a semi-automatic property is as follows.

```
property property_name
{
  [scope] read;
  [scope] write
  {
    // statements
  }
}
```

Or

```
property property_name
{
  [scope] read
  {
    // statements
  }
  [scope] write;
}
```

Indexers:

Indexers are a special type of property that allows instances of user-defined classes to behave like collections. A class can have only one indexer, and it cannot be static. An indexer is always declared as a property with an empty pair of square brackets (`[]`) as its name. It cannot be automatic: you must provide logic for its accessors. An indexer's accessors take an implicit parameter called **__key** that contains the value of the expression enclosed in the square brackets (i.e., the index or key) at the time the indexer is invoked. An indexer's writer also takes the implicit parameter **__value** like any other writer.

SPECIFYING AN EVENT:

An event specification consists of the **event** keyword followed by the prototype of the event. The prototype is a name (an identifier) followed by a comma-separated list of parameters in parentheses. Once a *foo* event is defined in a class, this automatically adds three methods to the class:

- A method to add handlers to the *foo* event: `void add_foo(closure handler)`
- A method to remove handlers from the *foo* event: `void remove_foo(closure handler)`
- A method to trigger the *foo* event: `void trigger_foo(...)` .

trigger_foo is always private and always has the same parameters as the event itself.

SPECIFYING AN OPERATOR OVERLOAD:

An operator overload specification consists of the **operator** keyword followed by the operator itself and its parameters. Depending on the operator being overloaded, the list may be empty or contain a single parameter. The complete list of operators that can be overloaded is as follows:

- Unary operators: +, -, ++, --, ~
- Binary operators: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=, **startswith**, **endswith**, **contains**, **matches**

Remarks:

1. No modifiers should be specified when overloading an operator. This means that an operator cannot be **abstract**, **static**, or **final**.
2. In the case of increment (`++`) or decrement (`--`) operators, the difference between overloading the prefix form and overloading the postfix form is that the postfix form expects an unused parameter while the prefix form expects no parameter.
3. Operators are not really a special type of class member in AddyScript. Each operator is actually a method with a special name. This can be verified by introspecting a class in which an operator is overloaded. AddyScript looks for such a method whenever it encounters a unary or binary operation involving an instance of a user-defined class and an overloadable operator.

Example of a class

A SIMPLE CLASS

In this example, we will define a Person class with three fields, four properties (three of them mapping the three fields plus an automatic one), a method, and an event.

```
class Person
{
    // Fields
    private _name;
    private _sex = 'Male';
    private _courtesy = 'Mr.';

    // A property with the 'compact' syntax
    public property name
    {
        read => this._name;
        write => this._name = __value;
    }

    // A property with the typical syntax
    public property sex
    {
        read { return this._sex; }
        write
        {
            // Updating sex will also update courtesy and raise the sex_changed event
            var oldSex = this._sex;
            this._sex = __value;
            this._courtesy = __value.toLower() switch {
                'male' => 'Mr.',
                'female' => 'Mrs.',
                _ => 'Dear'
            };
            this.trigger_sex_changed(oldSex, __value);
        }
    }

    // A read-only property encapsulating the _courtesy field
    public property courtesy => this._courtesy;

    // A fully automatic property
    public property age;

    // An event
    public event sex_changed(oldSex, newSex);

    // A method
    public function summary()
    {
        return $'{this.courtesy} {this.name}, {this.sex} person aged {this.age}';
    }
}

// Usage:
john = new Person();
john.name = 'John';
john.age = 42;
println(john.summary());

john.add_sex_changed(|old, _new| => println($'Sex changed from {old} to {_new}'));
john.add_sex_changed(|o, n| => println('Maybe the name should change too'));
john.sex = 'Female';
println(john.summary());
```

Output

```
Mr. John, Male person aged 42
Sex changed from Male to Female
Maybe the name should change too
Mrs. John, Female person aged 42
```

A CLASS WITH AN INDEXER

Example of a class that has an indexer:

```
class PhoneBook
{
    // Backing field for the indexer
    private _entries = {=>};

    // Indexer definition
    public property []
    {
        read => this._entries[__key];
        write => this._entries[__key] = __value;
    }

    // A method to display the content of the phone book
    public function toString(format = 'g')
    {
        var result = 'Phone Book:';
        foreach (name in this._entries.keys)
        {
            result += $'\n - {name}: {this._entries[name]}';
        }
        return result;
    }
}

// Usage:
pb = new PhoneBook();
pb['John Doe'] = '555-1234';
pb['Jane Smith'] = '555-5678';
println(pb);
```

Output

```
Phone Book:
 - John Doe: 555-1234
 - Jane Smith: 555-5678
```

The this keyword

In the body of a method, the keyword **this** can be used to refer to the current instance of the class (the one on which the method is invoked). Most of the time, you will use this feature to access other members of a class from the body of one of its methods.

Constructors

A **constructor** is a special method automatically invoked by the scripting engine when an instance of a class is created to initialize that instance. In AddyScript, a class has only one constructor since the language does not support method overloading. The definition of a constructor follows this special syntax:

```
[scope] constructor (list_of_parameters) [: super(list_of_arguments)]
{
   statements;
}
```

Where

- *scope* is one of the **private**, **protected** and **public** keywords. If no scope is provided, **private** is assumed by default.

scope has the following effects on instance creation:

- **private**: Only the class will be able to create instances of itself.
- **protected**: Only the class and its derived classes will be able to create instances of itself.
- **public**: Instances of the class can be created anywhere in the code.
- The optional 'colon-super' part is used to invoke the constructor of the parent class (if any) prior to any other statement.
- A constructor is not allowed to return a value.

Example: let's add a constructor in the Person class

```
class Person
{
    // Fields: we don't need default values anymore as the constructor is supplying them
    private _name;
    private _courtesy;

    // Here is the constructor, all parameters are made optional to allow the user to omit them
    public constructor(name = "", sex = "Male", age = 0)
    {
       this.name = name;
       this.sex = sex;
       this.age = age;
    }

    // A property with the 'compact' syntax
    public property name
    {
        read => this._name;
        write => this._name = __value;
    }

    // A semi-automatic property
    public property sex
    {
        read;
        write
        {
            // Updating sex will also update courtesy and raise the sex_changed event
            var oldSex = this.__sex;
            this.__sex = __value;
            this._courtesy = __value switch {
                matches '/male/i' => 'Mr.',
                matches '/female/i' => 'Mrs.',
                _ => 'Dear'
            };
            this.trigger_sex_changed(oldSex, __value);
        }
    }

    // A read-only property encapsulating the _courtesy field
    public property courtesy => this._courtesy;

    // A fully automatic property
    public property age;

    // An event
    public event sex_changed(oldSex, newSex);

    // A method
    public function summary()
    {
        return $'{this.courtesy} {this.name}, {this.sex} person aged {this.age}';
    }
}

// Usage:
jane = new Person("Jane", "Female", 30);
println(jane.summary());
jane.add_sex_changed(|old, _new| => println("Sex changed from {0} to {1}", old, _new));
jane.add_sex_changed(|o, n| => println("Why not call him John?"));
jane.sex = "Male";
println(jane.summary());
```

Output

```
Mrs. Jane, Female person aged 30
Sex changed from Female to Male
Why not call him John?
Mr. Jane, Male person aged 30
```

CONSTRUCTORS AND PROPERTY INITIALIZERS

A constructor call can be followed by a set of property initializers. This allows you to quickly initialize fields or properties that are not initialized by the constructor. It can also be used to add undeclared fields to a specific instance of a class. When using property initializers, if the constructor has no parameters, there is no need to add parentheses to it.

Example

```
class Point
{
    public property x;

    public property y;

    public function toString(format = 'g')
    {
        return '(' + this.x + ', ' + this.y + ')';
    }

}

pt = new Point {x = 10, y = -5};
println(pt.toString());
```

Output

```
(10, -5)
```

Modifiers

When working with classes in AddyScript, you will always come across keywords like **private**, **protected**, **public**, **static**, **abstract**, and **final**. These are modifiers (in the broad sense of the word). The first three (**private**, **protected**, and **public**) are used to control the scope of a class member. The other three are used to manage the behavior of a class or one of its members. The table below details the meaning of each modifier depending on where it is used.

Modifier	Effect on a class	Effect on a class member
private	Not applicable	Makes the member inaccessible out of its declaring class
protected	Not applicable	Makes the member accessible only to its declaring class and subclasses of its declaring class
public	Not applicable	Makes a member accessible to any part of the script
static	Forces all members of the class to be static. Disallows creation of instances of the class.	Indicates that the member only accesses other static members of the class. Such a member can be referenced without having to create an instance of the class. Can be combined with **final** on fields. Not applicable to constructors and indexers.
final	Forbids that another class inherits from this one	Indicates that the property or method cannot be overridden in a subclass. Can be combined with **static** on fields. Not applicable to constructors and events.
abstract	Allows the declaration of abstract properties and methods within the class. Disallow the creation of instances of the class	Indicates that the property or method is purely virtual, thus having no default implementation. Not applicable to constructors, fields and events.

Common members

There is a set of members that any class in AddyScript (including the void type) exposes. The table below presents those members and describes them.

Member	Nature	Description
`ClassInfo type { read; }`	final property	Gets the description of the class to which the target object belongs.
`bool equals(any other)`	method	Compare two objects and returns true if they're equal. Returns false otherwise.
`int hashCode()`	method	Gets the hash code of an object.
`int compareTo(any other)`	method	Compares the target object to the given one. Returns -1, 0, 1 depending on whether the target object is less than, equal or greater than the given value.
`string toString(string format = 'g')`	method	Gets the (eventually formatted) textual representation of an object.
`any clone()`	method	Gets a deep copy of an object. For most types, this simply returns the target object itself. *clone* is specially useful with collections and objects (native or not). For a native object, clone tests if the object's type implements the *ICloneable* interface and if so, invokes its *Clone* method.
`void dispose()`	method	Tries to release the unmanaged resources held by the target object. Does nothing for most types. For a collection or an object, recursively calls itself on each item or field. For a native object, tests if the object's type implements the *IDisposable* interface and if so, invokes its *Dispose* method.

Apart from the *type* property, any of these members can be overridden in your custom classes. AddyScript internally uses them to compare data items and to identify them in collections.

Records

A record is a special type of class that allows you to represent a simple data structure. Records are immutable and final by default. The scripting engine automatically generates a constructor for them, as well as overriding the `equals`, `hashCode`, `toString`, and `clone` methods. It also automatically generates a property for each of the record's fields. The equality (`==`) and inequality (`!=`) operators are also overloaded to make comparing records easier.

The syntax for defining a record is as follows:

Form 1:

```
record recordName(field1, field2, ..., fieldN);
```

Form 2:

```
record recordName(field1, field2, ..., fieldN)
{
    additionalMembers
}
```

Note:

- *recordName* is the name you want to give your record type.
- *additionalMembers* is a series of definitions for fields, properties, methods, operators, and/or events.
- None of the additional members should have the same name as any of the record's fields.
- No constructor can be explicitly defined in a record.
- The immutability of records with additional members is not guaranteed, as any of these members can modify the record.
- Manually defined fields and/or properties are not taken into account in automatically generated methods; if necessary, it is up to the user to redefine them.
- In all its forms, the above syntax is essentially equivalent to the following:

```
final class recordName
{
    // Generated fields:
    private final __field1;
    private final __field2;
    ...
    private final __fieldN;

    // Generated constructor:
    public constructor (field1, field2, ..., fieldN)
    {
        this.__field1 = field1;
        this.__field2 = field2;
        ...
        this.__fieldN = fieldN;
    }

    // Generated properties:
    public property field1 => this.__field1;
    public property field2 => this.__field2;
    ...
    public property fieldN => this.__fieldN;

    // A protected property that returns a tuple of all declared fields:
    protected property __members => (this.__field1, this.__field2, ..., this.__fieldN);

    // Generated methods:
    public function equals(other) => other is recordName && this.__members == other.__members;
    public function hashCode() => this.__members.hashCode();
    public function toString(format = '') => 'recordName' + this.__members.toString(format);
    public function clone() => new recordName(..this.__members);

    // Generated operators:
    public operator ==(other) => this.equals(other);
    public operator !=(other) => !this.equals(other);
```

```
    // Any additional members are inserted here
}
```

Example

```
record Point(x, y);

p1 = new Point(10, 20);
p2 = new Point(30, 40);
p3 = new Point(10, 20);
println($'p1 = {p1}');
println($'p2 = {p2}');
println($'p3 = {p3}');
println($'p1 == p2: {p1 == p2}');
println($'p1 != p2: {p1 != p2}');
println($'p1 == p3: {p1 == p3}');
println($'p1 != p3: {p1 != p3}');
println();

record Vector(x, y) {
  public property length => sqrt(this.x * this.x + this.y * this.y);
  public function toPoint() => new Point(this.x,  this.y);
}

v1 = new Vector(10, 20);
println($'v1 = {v1}');
println($'v1.length = {v1.length}');
println($'v1.toPoint() = {v1.toPoint()}');
println($'p1 == v1: {p1 == v1}');
println($'p1 != v1: {p1 != v1}');
println($'p1 == v1.toPoint(): {p1 == v1.toPoint()}');
println($'p1 != v1.toPoint(): {p1 != v1.toPoint()}');
```

Output

```
p1 = Point(10, 20)
p2 = Point(30, 40)
p3 = Point(10, 20)
p1 == p2: false
p1 != p2: true
p1 == p3: true
p1 != p3: false

v1 = Vector(10, 20)
v1.length = 22,360679774997898
v1.toPoint() = Point(10, 20)
p1 == v1: false
p1 != v1: true
p1 == v1.toPoint(): true
p1 != v1.toPoint(): false
```

MUTABLE COPY OF A RECORD

In AddyScript, you can create a mutable copy of a record by using the **with** operator. This operator takes a record as its left operand and a set of field assignments as its right operand. The result of the operation is a new record with the same fields as the original record, but with the values of the specified fields replaced by the corresponding values from the right operand. The following example shows how to use the **with** operator to create a mutable copy of a record:

```
record Point(x, y);

p1 = new Point(10, 20);
p2 = p1 with {x = 30, y = 40};
p3 = p1 with {y = 50};

println($'p1 = {p1}');
println($'p2 = {p2}');
println($'p3 = {p3}');
```

Output

```
p1 = Point(10, 20)
p2 = Point(30, 40)
p3 = Point(10, 50)
```

3.7.2 Inheritance and polymorphism

Inheritance

Inheritance is an object-oriented programming concept that allows you to define new classes that extend existing ones, that is, the new class inherits all the members of its ancestor and adds some new members. Instances of the child class are also considered by AddyScript to be instances of the parent class (meaning that if B inherits from A, where b is an instance of B, then the expression `b is A` evaluates to **true**). In the example below, we will illustrate how inheritance is handled in AddyScript by creating a subclass of the Person class introduced in the previous section. The example also shows how to invoke the parent class's constructor (to initialize inherited fields).

```
class Employee : Person
{
    public constructor(name, sex, age, hireDate, department, jobTitle)
         : super(name, sex, age)
    {
        this.hireDate = hireDate;
        this.department = department;
        this.jobTitle = jobTitle;
    }

    public property hireDate;

    public property department;

    public property jobTitle;
}

// Usage:
steve = new Employee('Steve', 'Male', 32, `2002-04-18`, 'IT', 'Senior Analyst');
println('{0} works for us since {1} years', steve.summary(), now().subtract(steve.hireDate, 'year'));
println('He is currently a {0} at the {1} department', steve.jobTitle, steve.department);
```

Output

```
Mr. Steve, Male person aged 32 works for us since 23 years
He is currently a Senior Analyst at the IT department
```

The so defined *Employee* class inherits all the members of the *Person* class in addition to those that it declares itself.

Polymorphism

Polymorphism is nothing more than the ability to override inherited members in derived classes. Since AddyScript uses duck typing (i.e.: objects are generally taken for what they appear to be without further checking), it handles polymorphism without any additional semantics. Any class method is

overridable unless it is marked as **private**, **final**, or **static** in the parent class. Also, when the parent class is **abstract**, any subclass that is not abstract must override its abstract properties and methods.

Example

```
// A base abstract class with a single abstract method
abstract class Pet
{
    public abstract function cry();
}

// A subclass with a custom cry
class Cat : Pet
{
    public function cry() => println('As a cat, I meow');
}

// Another subclass with a custom cry
class Dog : Pet
{
    public function cry() => println('As a dog, I bark');
}

// Another subclass with a custom cry
class Pig : Pet
{
    public function cry() => println('As a pig, I grunt');
}

// a set of pets
pets = {new Dog(), new Cat(), new Pig()};

// making them all cry
foreach (pet in pets)
    pet.cry();
```

Output

```
As a dog, I bark
As a cat, I meow
As a pig, I grunt
```

Note: the above example would work even if *Cat*, *Dog* and *Pig* were not declared as subclasses of *Pet*. That is a consequence of AddyScript's **duck typing** philosophy: an instance of any class that exposes a public instance *cry* method without any parameters can be used where a Pet is expected.

The super keyword

Even after you have overridden a method, you still can refer to its original implementation as `super::methodName` (methodName being the name of that method). The same rule applies for properties and indexers.

Protocols

AddyScript has a number of protocols that, if implemented by a user-defined class allow instances of that class to behave like some of the predefined data types in certain aspects.

THE ITERATOR PROTOCOL:

First approach: moveFirst, hasNext, moveNext

The iterator protocol makes it possible to iterate over instances of a user-defined class with a foreach loop. For a class to implement this protocol, it must expose 3 methods named **moveFirst**, **hasNext**, and **moveNext** respectively. The role of **moveFirst** is to position the internal cursor on the first logical element of the collection (assuming your class is a custom collection). **hasNext** is supposed to return a boolean indicating whether the

iteration can continue or not. Finally, **moveNext** is responsible for moving the internal cursor forward, returning the value pointed to by the cursor at each step.

Example

```
class Range
{
   private start;
   private end;
   private step;
   private current;

   public constructor(start, end = 0, step = 1)
   {
      // Swap start and end to be in the correct order
      if (start > end) (start, end) = (end, start);
      // Ensure that step is always positive
      if (step <= 0) throw 'step has to be positive';

      this.start = start;
      this.end = end;
      this.step = step;
   }

   public function moveFirst() => this.current = this.start;

   public function hasNext() => this.current < this.end;

   public function moveNext()
   {
      var current = this.current;
      this.current += this.step;
      return current;
   }
}

foreach (item in new Range(5, 25, 5))
   println(item);
```

Output

```
5
10
15
20
```

Second approach: iterator method with yield statements

Alternatively, a class can implement the iterator protocol simply by exposing an **iterator** method whose body contains a succession of **yield** statements (presumably in a loop). The **yield** statement has a syntax similar to that of a **throw** statement or a parameterized **return** statement, but it is not actually a jump. It only tells AddyScript what value the iterator being constructed should return on each iteration.

Example

```
class Range
{
   private start;
   private end;
   private step;

   public constructor(start, end = 0, step = 1)
   {
      // Swap start and end to be in the correct order
      if (start > end) (start, end) = (end, start);
      // Ensure that step is always positive
      if (step <= 0) throw 'step has to be positive';

      this.start = start;
      this.end = end;
      this.step = step;
   }

   public function iterator()
   {
      for (var current = this.start; current < this.end; current += this.step)
         yield current;
   }
}

foreach (item in new Range(5, 25, 5))
   println(item);
```

Output

```
5
10
15
20
```

3.7.3 Introspection

Introspection (also called reflection) allows you to discover the type of an object and its members at runtime and manipulate them dynamically. This is useful for various scenarios, such as serialization, object mapping, dependency injection, and dynamic proxies. AddyScript provides a comprehensive introspection API that enables you to inspect types, fields, properties, methods, and events.

Type Information

You can obtain type information from a class using the `typeof` operator, which returns a **TypeInfo** object representing the specified type. You can also get the type of an instance using the `type` property of the object. The **TypeInfo** class provides various properties and methods to inspect the type. They are listed in the table below:

Member	Nature	Description
`string name { read; }`	Property	Gets the name of the type.
`TypeInfo superType { read; }`	Property	Gets the base type of the type.
`bool isIntegral { read; }`	Property	Indicates whether the type is an integral type.
`bool isNumeric { read; }`	Property	Indicates whether the type is a numeric type.
`bool isTemporal { read; }`	Property	Indicates whether the type is a temporal type.
`bool isSequential { read; }`	Property	Indicates whether the type is a sequential type.
`bool isCollection { read; }`	Property	Indicates whether the type is a collection type.
`bool isSubclassOf(TypeInfo otherType)`	Method	Determines whether the current type is a subclass of the specified type.
`bool isAssignableTo(TypeInfo otherType)`	Method	Determines whether instances of the current type can be assigned to variables of the specified type.
`bool isAssignableFrom(TypeInfo otherType)`	Method	Determines whether instances of the specified type can be assigned to variables of the current type.
`MethodInfo $constructor { read; }`	Property	Gets the constructor method of the type.
`PropertyInfo indexer { read; }`	Property	Gets the indexer property of the type, if any.
`map fields { read; }`	Property	Gets a map of fields defined in the type identified by their names.
`map properties { read; }`	Property	Gets a map of properties defined in the type identified by their names.
`map methods { read; }`	Property	Gets a map of methods defined in the type identified by their names.
`map events { read; }`	Property	Gets a map of events defined in the type identified by their names.
`list attributes { read; }`	Property	Gets a list of attributes applied to the type.
`any newInstance(..args)`	Method	Creates a new instance of the type by invoking its constructor.

Member Information

The members of a type can be inspected using various classes. All are based on a common **MemberInfo** class that provides basic information about a member, such as its name, scope, and modifier.

MEMBERINFO

The **MemberInfo** class provides the following members:

Member	Nature	Description
`string name { read; }`	Property	Gets the name of the member.
`string fullName { read; }`	Property	Gets the fully qualified name of the member.
`TypeInfo holder { read; }`	Property	Gets the type that declares the member.
`string scope { read; }`	Property	Gets the scope of the member (Private, Protected, Public).
`string modifier { read; }`	Property	Gets the modifier of the member (Static, Final, Abstract, etc.).
`list attributes { read; }`	Property	Gets a list of attributes applied to the member.

The following subclasses of **MemberInfo** provide additional information specific to the type of member.

FIELDINFO

The **FieldInfo** class represents a field in a type and provides the following additional members:

Member	Nature	Description
`any sharedValue { read; }`	Property	Gets the value of the static field.
`any getValue(any target)`	Method	Gets the value of the field for the specified target object.
`void setValue(any target, any value)`	Method	Sets the value of the field for the specified target object.

PROPERTYINFO

The **PropertyInfo** class represents a property in a type and provides the following additional members:

Member	Nature	Description
`bool canRead { read; }`	Property	Indicates whether the property has a reader.
`bool canWrite { read; }`	Property	Indicates whether the property has a writer.
`MethodInfo reader { read; }`	Property	Gets the reader method of the property.
`MethodInfo writer { read; }`	Property	Gets the writer method of the property.
`any getValue(any target)`	Method	Gets the value of the property for the specified target object.
`void setValue(any target, any value)`	Method	Sets the value of the property for the specified target object.
`any getItem(any target, any index)`	Method	Gets the value of the property at the given index for the specified target object.
`void setItem(any target, any index, any value)`	Method	Sets the value of the property at the given index for the specified target object.

METHODINFO

The **MethodInfo** class represents a method in a type and provides the following additional members:

Member	Nature	Description
`list parameters { read; }`	Property	Gets a list of parameters defined for the method.
`any invoke(any target, ..args)`	Method	Invokes the method on the specified target object with the given arguments.

EVENTINFO

The **EventInfo** class represents an event in a type and provides the following additional members:

Member	Nature	Description
`list parameters { read; }`	Property	Gets a list of parameters defined for the event.

PARAMETERINFO

The **ParameterInfo** class represents a parameter of a method or event and provides the following members:

Member	Nature	Description
`string name { read; }`	Property	Gets the name of the parameter.
`bool byRef { read; }`	Property	Indicates whether the parameter is passed by reference.
`bool vaList { read; }`	Property	Indicates whether the parameter is a variadic parameter.
`bool canBeEmpty { read; }`	Property	Indicates whether the parameter can be a null reference or an empty string or collection.
`any defaultValue { read; }`	Property	Gets the default value of the parameter, if any.

Introspection showcase

The following example shows how this functionality is handled in AddyScript.

Introspection demo