

Grokking Nemerle

January 21, 2006

Contents

1	Grokking Nemerle	1
1.1	Don't Panic!	1
2	Grok Base structure of programs	1
2.1	Running the compiler	2
2.2	Methods	2
2.3	Fields	3
2.4	Expressions	3
2.5	A simple function	4
2.6	Imperative loops and value definitions	4
2.7	Local functions	5
2.8	Type inference	6
2.9	External functions	6
2.9.1	Output formatting	7
2.9.2	String interpolation	7
2.10	Arrays	8
2.10.1	Type enforcement and variants of <code>if</code> expression	9
2.10.2	Multidimensional arrays	9
2.11	Miscellaneous information	9
2.12	Exercises – List 1	9
3	Grok Namespaces	10
3.1	Namespaces	10
3.2	Making long names short	11
4	Grok Functionals	11
4.1	Functional values	11
4.2	Lambda expressions	13
4.3	Exercises	14
5	Grok Various data structures	14
5.1	Tuples	14
5.2	List literals	15
6	Grok Variants and matching	15
6.1	Variants	15
6.2	Matching	17
6.3	Other patterns	17
6.4	Using variants as trees	20
6.5	XML trees	20
6.6	Exercises	23
7	Grok The rest	24
7.1	Named parameters	24
7.2	The void literal	25

7.3	Operator overloading	25
7.4	Macros	25
7.4.1	Core language constructs	25
7.4.2	Design by contract	26
7.4.3	Other examples	26
8	Grok Object oriented programming	26
8.1	Back in the Refrigerator	27
8.2	Inheritance	28
8.3	Virtual calls	29
8.4	Interfaces	31
9	Grok Parametric polymorphism	31
9.1	Simple polymorphism	32
9.2	Constraints on type variables	32
10	Grok Exceptions	34
11	Grok Properties, indexers, delegates and events	35
11.1	Properties	35
11.2	Indexers	36
11.3	Delegates	36
11.4	Events	37

1 Grokking Nemerle

"This is as rock; talk in the True Speech." from [A Wizard of Earthsea](#) by [Ursula K. Le Guin](#)

1.1 Don't Panic!

This document contains materials used to teach Nemerle during the "Nemerle Programming Language" course taught at our institute. It should make a good tutorial.

There are references to various languages throughout this document. If you don't know much about these languages – just ignore these remarks. As for remarks for the C language – unless otherwise stated they also apply to C# and Java.

Some links: [grokking?!](#), [don't panic](#).

1. [Grok Base structure of programs](#)
2. [Grok Namespaces](#)
3. [Grok Functionals](#)
4. [Grok Various data structures](#)
5. [Grok Variants and matching](#)
6. [Grok The rest](#)
7. [Grok Object oriented programming](#)
8. [Grok Parametric polymorphism](#)
9. [Grok Exceptions](#)
10. [Grok Properties, indexers, delegates and events](#)

2 Grok Base structure of programs

This page is a part of the [Grokking Nemerle](#) tutorial.

This chapter explains what a basic program looks like in Nemerle. This is the longest and probably the hardest lesson, so don't worry if you don't understand everything at once.

2.1 Running the compiler

In order to run programs written in Nemerle you need to compile them to .NET bytecode first. This is done with the `ncc` (Nemerle Compiler Compiler) command. Assuming the Nemerle is installed properly on your system, you need to do the following:

- write the program text with your favorite text editor and save it as a file with extension `.n`, for example `myfile.n`
- run the Nemerle compiler by typing `ncc myfile.n`
- the output goes to `out.exe`
- run it by typing `out` (Windows) or `mono out.exe` (Linux)

You cannot define either functions or values at the top level in Nemerle. You need to pack them into **classes** or **modules**. For C#, Java and C++ programmers: a module is a class with all members static, there can be no instance of module class.

For now you should consider the module to be a form of packing related functions together.

```
class SomeClass
{
    // ... some comment ...
    some_field : int;

    /* ... some ...
       ... multiline comment ... */
    other_field : string;
}
```

The above example also introduces two kinds of **comments** used in Nemerle. The comments starting with `//` are active until the end of line, while the comments starting with `/*` are active until `*/`. This is the same as in C (well, C99).

2.2 Methods

Classes can contain **methods** (functions) as well as **fields** (values). Both kind of members can be prefixed with access attributes:

- **public** defines a method or a field that can be accessed from outside the module.
- **private** defines a member that is local to the module. This is the default.
- **internal** defines a member that is local to a given library or executable.

In the method declaration header you first write modifiers, then the name of the method, then parameters with a type specification and finally a type of values returned from this function.

Typing constraints are in general written after a colon (`:`).

```
class Foo
{
    public SomeMethod () : void
```

```
{
  // ...
}
private some_other_method (_x : int) : void
{
  // ...
}
private Frobnicate (_x : int, _y : string) : int
{
  // we return some int value here
  0
}
internal foo_bar () : void
{
  // ...
}
}
```

2.3 Fields

Fields define global values inside the module.

Fields accept the same access attributes as methods. However, there is one additional very important attribute for fields – `mutable`.

By default fields are read only, that is they can be assigned values only in the module initializer function (codenamed `this`; we will talk about it later). If you want to assign values to fields in other places you need to mark a field `mutable`.

```
class Bar
{
  public mutable qux : int;
  private quxx : int;
  mutable bar : float;
}
```

2.4 Expressions

Nemerle makes no distinction between an expression and a statement, there are only expressions. The idea is to easily operate on values, which every expression returns as a result of its computation. The exception is a `throw` keyword, which breaks control flow by throwing an exception (the breaking statements were adopted in Nemerle using the `block` construct, which makes jumps localised and fit well into the expressions-only concept).

Also `{ }` sequence is an expression, which have its value - the value of the last expression in it. This eliminates the need for `return` statement in functions. The value returned from a function is the last value computed in its body. You can think that there is an implicit `return` at the end of each function. This is the same as in ML languages.

Despite all that, the most basic example looks almost like in C#. The entry point for a program is a function called `Main`. It is also possible to take command line arguments and/or return integer return code from the `Main` method, consult `.NET` reference for details.

Note that unlike in ML function call requires `()`.

```
class Hello
{
    public static Main () : void
    {
        System.Console.WriteLine ("Hello cruel world!")
    }
}
```

2.5 A simple function

However, the following example (computing **Fibonacci sequence**) looks somewhat different. You can see the usage of a **conditional expression**. Note how the value is returned from the function without any explicit return statement.

Note that this example and the following ones are not complete. To be compiled they need to be packed into the module, equipped with the Main function and so on.

```
fib (n : int) : int
{
    if (n < 2)
        1
    else
        fib (n - 1) + fib (n - 2)
}
```

2.6 Imperative loops and value definitions

It is possible to use regular **imperative loops** like **while** and **for**. Both work as in C. In the example of the **for** loop the first expression is put before the loop, the second expression is the condition (the loop is executed as long as the condition holds) and the last expression is put at the end of the loop.

However, the most important thing about this example is the variable definition used here. **Variables** (values that can be changed) are defined using the **mutable** expression. You do not specify a type, but you do specify an initial value. The type of the defined variable is inferred from the initial value (for example the type of 1 is obviously an **int**). The variable introduced with **mutable** is visible until the end of the current sequence. Lexical scoping rules apply: the definition of a new value with the same name hides the previous one.

Sequence is a list of expressions enclosed in braces (**{}**) and separated with semicolons (**;**). An optional semicolon is allowed at the end of the sequence. Note that the function definition also introduces a sequence (as the function body is written in **{}**).

mutable defines **variables** that can be updated using the assignment operator (**=**). In contrast **def** defines **values** that cannot be updated – in our example we use **tmp** as such value.

Older versions of the Nemerle compiler required a semicolon after the closing brace inside a sequence. This is no longer mandatory: note that for expression's closing brace is not followed by a semicolon. In some rare cases this can introduce compilation errors – remember that you can always put a semicolon there!

```
fib (n : int) : int
{
    mutable last1 = 1;
    mutable last2 = 1;

    for (mutable cur = 1; cur < n; ++cur) {
        def tmp = last1 + last2;
    }
}
```



```
}
```

Notice how the local function is used to organize the loop. This is typical for Nemerle. It is therefore quite important for you to grok this concept. Some external links – [tail recursion](#), [recursion](#).

Here goes another example of a loop constructed with a local function.

```
fib (n : int) : int
{
  def my_loop (last1 : int, last2 : int, cur : int) : int {
    if (cur >= n)
      last2
    else
      my_loop (last2, last1 + last2, cur + 1)
  }

  my_loop (1, 1, 1)
}
```

If you are concerned about performance of such form of writing loops – fear you not. When the function body ends with a call to another function – no new stack frame is created. It is called a **tail call**. Thanks to it the example above is as efficient as the for loop we have seen before.

2.8 Type inference

You can specify types of parameters as well as return types for local functions. However in some (most?) cases the compiler can guess (infer) the types for you, so you can save your fingers by not typing them. This is always safe, that is the program should not in principle change the meaning if type annotations are added.

Sometimes the compiler is unable to safely infer the type information, in which case an error message will be generated to indicate that type annotations are required.

In the following example we have omitted the return type, as well as types of the parameters. Compiler can figure out the types, because literals 1 are used in a few places, which are of the type `int`.

```
fib (n : int) : int
{
  def my_loop (last1, last2, cur) {
    if (cur >= n)
      last2
    else
      my_loop (last2, last1 + last2, cur + 1)
  }

  my_loop (1, 1, 1)
}
```

2.9 External functions

One of the best things about Nemerle is that you can use rich class libraries that come with the Framework as well as the third party libraries. Links to the documentation about .NET class libraries [can be found here](#).

New objects are constructed by simply naming the type and supplying arguments to its constructor. Note that unlike in C# or Java you don't use the `new` keyword to construct new objects. Methods of objects can be invoked later, using the dot operator (`some_object.SomeMethod (some_argument)`). Static methods are

invoked using the `NameSpace.TypeName.MethodName ()` syntax. We will talk more about this object oriented stuff later.

Now some example:

```
the_answer_to_the_universe () : int
{
    // Construct new random number generator.
    // This is object construction, explained in more detail later
    def r = System.Random ();

    // Return new random number from [0, 99] range.
    // This is again object invocation, explained later
    r.Next (100)
}
```

For more information about *The answer to the Ultimate Question of Life, the Universe and Everything* please visit [this site](#). Please note that this program run on a computer not as powerful as Deep Thought will be right only in 1% of cases.

2.9.1 Output formatting

There are several methods of output formatting in Nemerle.

The most basic .NET methods of displaying stuff on the screen is the `System.Console.WriteLine` method. In the simplest form it takes a string to be displayed. If you however supply it with more then one argument, the first one is treated as a format string and occurrences of `{N}` are replaced with the value of (N+2)-th parameter (counting from one).

```
print_answer () : void
{
    def the_answer = the_answer_to_the_universe ();
    System.Console.WriteLine ("The answer to the Ultimate "+
                              "Question of Life, the "+
                              "Universe and Everything "+
                              "is {0}", the_answer)
}
```

There are however other means, for example the `Nemerle.IO.printf` macro that works much like the `printf(3)` C function or `Printf.printf` in OCaml. (Well it doesn't yet handle most formatting modifiers, but patches are welcome ;-)

```
printf_answer () : void
{
    def the_answer = the_answer_to_the_universe ();
    Nemerle.IO.printf ("The answer is %d\n", the_answer);
}
```

2.9.2 String interpolation

If you have ever programmed in Bourne shell, perl, python, php (or some other P-language), you know about string interpolation. It basically takes a string `"answer is $the_answer"` and replaces `$the_answer` with value of the variable `the_answer`. The good news is that we have this feature in Nemerle:

```
interpolate_answer () : void
{
    def the_answer = the_answer_to_the_universe ();
    Nemerle.IO.print ("The answer is $the_answer\n");
}
```

It comes in two flavors – the first one is the `Nemerle.IO.print` (without `f`) function – it does string interpolation on its sole argument and prints the result on `stdout`.

The second version is the `$` operator, which returns a string and can be used with other printing mechanisms:

```
interpolate2_answer () : void
{
    def the_answer = the_answer_to_the_universe ();
    System.Console.WriteLine ($ "The answer is $the_answer");
}
```

Both versions support special `$(...)` quotations allowing any code to be expanded, not just variable references:

```
interpolate3_answer () : void
{
    System.Console.WriteLine (
        $ "The answer is $(the_answer_to_the_universe ())");
}
```

The `$`-expansion works on all types.

2.10 Arrays

The type of array of `T` is denoted `array [T]`. This is a one-dimensional, zero-based array. There are two special expressions for constructing new arrays: `array ["foo", "bar", "baz"]` will construct a 3-element array of strings, while `array (100)` creates a 100-element array of something. The *something* is inferred later, based on an array usage. The empty array is initialized with `0`, `0.0`, etc. or `null` for reference types.

The assignment operator (`=`) can be also used to assign elements in arrays.

Note the `ar.Length` expression – it gets the length of array `ar`. It looks like a field reference in an array object but under the hood it is a method call. This mechanism is called *property*.

Our arrays are subtypes of `System.Array`, so all methods available for `System.Array` are also available for `array [T]`.

```
class ArraysTest {
    static reverse_array (ar : array [int]) : void
    {
        def loop (left, right) {
            when (left < right) {
                def tmp = ar[left];
                ar[left] = ar[right];
                ar[right] = tmp;
                loop (left + 1, right - 1)
            }
        }
        loop (0, ar.Length - 1)
    }
}
```

```

static print_array (ar : array [int]) : void
{
    for (mutable i = 0; i < ar.Length; ++i)
        Nemerle.IO.printf ("%d\n", ar[i])
}

static Main () : void
{
    def ar = array [1, 42, 3];
    print_array (ar);
    Nemerle.IO.printf ("\n");
    reverse_array (ar);
    print_array (ar);
}
}

```

2.10.1 Type enforcement and variants of if expression

One interesting thing about this example is the usage of the type enforcement operator – colon (:). We use it to enforce the left type to be `int`. We could have as well written `def loop (left : int, right) {`. This are simply two ways to achieve the same thing.

Another interesting thing is the `when` expression – it is an `if` without `else`. For symmetry purposes we also have `if` without `then` called `unless`. As you might have already noted `unless` is equivalent to `when` with the condition negated.

In Nemerle the `if` expression always needs to have the `else` clause. It's done this way to avoid stupid bugs with a dangling `else`

```

// C#, misleading indentation hides real code meaning
if (foo)
    if (bar)
        m1 ();
else
    m2 ();

```

If you do not want the `else` clause, use `when` expression, as seen in the example.

2.10.2 Multidimensional arrays

Multidimensional arrays (unlike arrays of arrays) are also available. Type of two-dimensional integer array is `array [2, int]`, while the expression constructing it is `array .[2] [[1, 2], [3, 4]]`. They are accessed using the comma-syntax: `t[1,0]`.

Both single- and multidimensional empty arrays can be created with `array (size)` or `array (size1, size2, ...)`. Empty means that they contain nulls or zeros.

Note that there exist two ways of obtaining multidimensional storage. The first one is the mentioned multidimensional array (type `array [N, int]`). The second one is array of arrays (type `array [array [int]]`), which is just a standard array, but its elements are other arrays. This is useful, because you can initialize only some parts of this array, while multidimensional array allocates all the memory at once. Elements of such a array of arrays are accessed by `x[2][3]`.

2.11 Miscellaneous information

The equality predicate is written `==` and the inequality is `!=` as in C.

The boolean operators (`&&`, `||` and `!`) are all available and work the same as in C.

2.12 Exercises – List 1

1.1. Write a program that prints out to the console:

```
1 bottle of beer.  
2 bottles of beer.  
3 bottles of beer.  
...  
99 bottles of beer.
```

With an appropriate amount of beer instead of `...`. The program source code should not exceed 30 lines.

1.2. Implement **bogo sort** algorithm for an array of integers. (WARNING: you should not implement *destroy the universe* step). Test it by sorting the following array: [4242, 42, -42, 31415].

1.3. As 1.2, but don't use the imperative loops – rewrite them with recursion.

3 Grok Namespaces

This page is a part of the [Grokking Nemerle](#) tutorial.

3.1 Namespaces

Classes and modules can be put in namespaces. Namespaces at the conceptual level prepend a string to names of objects defined within them.

```
namespace Deep.Thought  
{  
    class Answer  
    {  
        public static Get () : int  
        {  
            42  
        }  
    }  
}  
  
namespace Loonquawl  
{  
    class Brain  
    {  
        public static Run () : void  
        {  
            System.Console.WriteLine ("The answer {0}",  
                Deep.Thought.Answer.Get ())  
        }  
    }  
}
```

As you can see, the name of the `Get` function is first prepended with the name of the module (`Answer`) and then with the current namespace (`Deep.Thought`), forming its full name: `Deep.Thought.Answer.Get`.

Another example is the `WriteLine` method of the `Console` module, defined in the `System` namespace.

3.2 Making long names short

In order not to write `System.Console.WriteLine` or `Deep.Thought.Answer.Get` all the time you can import all declarations from the specified namespace into the current scope with the `using` directive.

Thus the `Loonquawl.Brain` module from the example above could be:

```
namespace LoonquawlUsing
{
    using System.Console;
    using Deep.Thought.Answer;

    class Brain
    {
        public static Run () : void
        {
            WriteLine ("The answer is {0}", Get ())
        }
    }
}
```

While we see not much gain from the `using` directive in this example, it can be handy when you use the `WriteLine` method 100 times, and/or your classes are in `Some.Very.Long.Namespaces`.

Note that unlike in C# all methods of a class can be imported into the current namespace. Thus, `using` is not limited to namespaces, but it also works for classes. The new thing is also the fact, that currently opened namespaces are used as prefix in all type lookups - for example if there is `using System;`, then you can write `Xml.XmlDocument ()`, or with an additional `using System.Xml;` you can write `XmlDocument ()`. Neither of these is possible in C#.

It is also possible to create (shorter) aliases for namespaces and types. It is sometimes useful in case when two namespaces share several types, so they cannot be imported with `using` simultaneously. This works in similar way to C#.

```
namespace LoonquawlAlias
{
    using SC = System.Console;
    using DTA = Deep.Thought.Answer;

    class Brain
    {
        public static Run () : void
        {
            SC.WriteLine ("The answer is {0}", DTA.Get ())
        }
    }
}
```

4 Grok Functionals

This page is a part of the [Grokking Nemerle](#) tutorial.

4.1 Functional values

In Nemerle you can pass functions as arguments of other functions, as well as return them as results. This way functions are not worse than any other data types (think about Equal Rights for Functions movement :-)

In C# there are delegates. This concept is quite similar to functional values. However, functional values are far more efficient and their types need not be declared before use.

As a first example consider:

```
// C#
delegate int IntFun (int);

class Delegates {
    static int f(int x)
    {
        return x * 2;
    }

    static int run_delegate_twice(IntFun f, int v)
    {
        return f(f(v));
    }

    static void Main ()
    {
        System.Console.WriteLine("{0}",
            run_delegate_twice(new IntFun (f), 3));
    }
}
```

```
// Nemerle
class Functions {
    static f (x : int) : int
    {
        x * 2
    }

    static run_funval_twice (f : int -> int, v : int) : int
    {
        f (f (v))
    }

    public static Run () : void
    {
        System.Console.WriteLine ("{0}", run_funval_twice (f, 3))
    }
}
```

In this example delegates seem just like function pointers in C. Functional values do not appear any better, except maybe for the shorter syntax. However, the real power of delegates comes from the fact that methods can be used as delegates (thus effectively embedding the this pointer in the delegate). This is much like "functional objects" design pattern in C++. We will not show it here, please refer to the C# manual for details.

Still using methods as delegates does not demonstrate their full power. The funny part begins when we use nested functions as functional values.

```

class MoreFunctions {
    static run_twice (f : int -> int, v : int) : int
    {
        f (f (v))
    }

    static run_adder (x : int) : void
    {
        def f (y : int) : int { x + y };
        System.Console.WriteLine ("{0}", run_twice (f, 1))
    }

    public static Run () : void
    {
        run_adder (1);
        run_adder (2);
    }
}

```

This example prints 3 and 5. Note how `x` is captured in the local function `f`.

Types of functions taking more than one argument are represented as tuples. For example, the following function:

```

some_function (_arg1 : int, _arg2 : string, _arg3 : SomeFoo) : float
{
    // ...
    System.Convert.ToSingle (_arg1)
}

```

has the type `int * string * Foo -> float`. Functions that take no arguments pretend to take one argument of the type `void`. Thus the function:

```

other_function () : string { "kaboo" }

```

possesses the type `void -> string`.

4.2 Lambda expressions

Lambda expressions are just a syntactic sugar to defined unnamed local functions. Unnamed local functions are useful when you need to pass some function to the iterator, so you use it just once.

Lambda expressions are defined using the `fun` keyword, followed by formal parameters, an optional return type and the function body.

```

def x = Nemerle.Collections.List.Map ([1, 2, 3], fun (x) { 2 * x });
def y = Nemerle.Collections.List.FoldLeft (x, 0, fun (val : int, acc) { acc + val })
assert (y == 12);

```

In general:

```

fun (parms) { exprs }

```

is equivalent to:

```
{
  def tmp (parms) { exprs };
  tmp
}
```

This feature is similar to anonymous delegates in C# 2.0.

4.3 Exercises

2.1. Write a function

```
string_pow : int * (string -> string) * string -> string;
```

The call `string_pow (3, f, "foo")` should result in calling `f` three times, i.e. returning result of `f (f (f ("foo")))`. When you are done, write a second version of `string_pow` using recursion (if you used imperative loop) or imperative loop (otherwise).

Test it by passing function that replaces "42" with "42ans". You can try passing the functional argument to `string_pow` as a lambda expression. **This method** might be useful.

5 Grok Various data structures

This page is a part of the [Grokking Nemerle](#) tutorial.

5.1 Tuples

Tuples are forms of nameless data structures. They are usable when you need to return two or three values from a function.

A tuple is constructed with:

```
(expr1, expr2, ..., exprN)
```

and deconstructed with:

```
def (id1, id2, ..., idN) = expr;
```

This `def` thing defines values called `id1` through `idN` and puts respective tuple members into them.

Tuple types are written using the `*` operator. For example, pair of integers (42, 314) has the type `int * int`, while ("Zephod", 42.0, 42) has the type `string * float * int`.

An example follows:

```
/** Parses time in HH:MM format. */
parse_time (time : string) : int * int
{
  def arr = time.Split (array [':']);
  (System.Int32.Parse (arr[0]), System.Int32.Parse (arr[1]))
}

seconds_since_midnight (time : string) : int
{
```

```
def (hours, minutes) = parse_time (time);
(hours * 60 + minutes) * 60
}

foo () : void
{
  def secs = seconds_since_midnight ("17:42");
  ...
}
```

Another example could be:

```
// split (3.7) => (3, 0.7)
split (x : double) : int * double
{
  def floor = System.Math.Floor (x);
  (System.Convert.ToInt32 (floor), x - floor)
}
```

5.2 List literals

List literals are special forms of writing lists. Lists are data structures that are used very often in Nemerle - often enough to get their own syntax. Lists in Nemerle are somewhat different than the `ArrayList` type in .NET.

- the lists in Nemerle are immutable (cannot be changed once created)
- items can be appended at the beginning only (constructing new lists)

Of course you are free to use the .NET `ArrayList`.

Anyway, to construct a list consisting of a given **head** (first element) and a **tail** (rest of elements, also a list), write:

```
head :: tail
```

To construct a list with specified elements write:

```
[ element_1, element_2, ..., element_n ]
```

This way you can also construct an empty list (`[]`).

6 Grok Variants and matching

This page is a part of the [Grokking Nemerle](#) tutorial.

Variants (called data types or sum types in [SML](#) and [OCaml](#)) are forms of expressing data of several different kinds.

Matching is a way of destructuring complex data structures, especially variants.

6.1 Variants

The simplest example of variants are enum types known from C.

```
// C
enum Color {
  Red,
  Yellow,
  Green
}
```

```
// Nemerle
variant Color {
  | Red
  | Yellow
  | Green
}
```

Note that you can define C#-like `enum` types in Nemerle anyway.

```
// Nemerle
enum Color {
  | Red
  | Yellow
  | Green
}
```

However, the variant options might be more useful because they can carry some extra data with them:

```
variant RgbColor {
  | Red
  | Yellow
  | Green
  | Different {
    red : float;
    green : float;
    blue : float;
  }
}
```

So if color is neither red, yellow nor green, it can be represented with RGB. You can create variant object just like any other object, by using its constructor (which is always implicitly provided):

```
// ...
def _blue = RgbColor.Different (0f, 0f, 1f);
def _red = RgbColor.Red ();
```

You can think about variants as of a union with a selector in C. In OO world sometimes modeling variants with sub classing can be seen sometimes:

```
// C#
class Color {
  class Red : Color { }
  class Green : Color { }
  class Yellow : Color { }
  class Different : Color {
    float red;
  }
}
```

```

float green;
float blue;

public Different (float red, float green, float blue) {
    this.red = red;
    this.green = green;
    this.blue = blue;
}
}
}

```

Of course you need to write a constructor, mark fields public and so on. When you're done – using this kind of stuff is quite hard – you need to use lots of runtime type checks.

On the other hand, Nemerle provides an easy and convenient method of dealing with variants – pattern matching.

6.2 Matching

Pattern matching is accomplished with the match expression. Its semantics are to check each pattern in turn, from top to bottom, and execute expression after the first pattern that matched. If no pattern matched, the exception is raised. This is like the switch statement known from C, but using a large dose of steroids.

```

string_of_color (color : Color) : string
{
    match (color) {
        | Color.Red => "red"
        | Color.Yellow => "yellow"
        | Color.Green => "green"
        | Color.Different (r, g, b) =>
            System.String.Format ("rgb({0},{1},{2})", r, g, b)
    }
}

```

The main idea behind patterns is that they match values that look like them. For example, the Nemerle compiler creates a default constructor for the `Different` variant option with the following body:

```

public this (red : float, green : float, blue : float)
{
    this.red = red;
    this.green = green;
    this.blue = blue;
}

```

Therefore, the constructor call `Color.Different (r, g, b)` creates a new variant option instance with specified arguments. The pattern looks the same – it binds actual values of `red`, `green` and `blue` fields to `r`, `g` and `b` respectively. You can also spell the field names explicitly:

```

| Color.Different (red = r, green = g, blue = b) =>
    System.String.Format ("rgb({0},{1},{2})", r, g, b)

```

6.3 Other patterns

We have already seen a so called **”constructor pattern”** in action. It is used to match over variants. The constructor pattern consists of variant option name (starting with an uppercase letter) followed by optional

tuple or record pattern.

```
// examples of constructor patterns
// plain one, without sub-pattern:
Color.Red
// followed by tuple pattern:
Color.Different (r, g, b)
// followed by record pattern:
Color.Different (red = r, green = g, blue = b)
```

The **variable pattern** matches any value, and binds it to a specified variable. The variable pattern is an identifier starting with a lowercase letter. They are used mostly inside other patterns, but here we give a (rather pointless) example of using them as the top-level pattern:

```
// it prints 42
match (42) {
  | x => // here x is bound to 42
    printf ("%d\n", x)
}
```

The **throw-away pattern**, written `_`, matches any value and has no further effects. It is a way of specifying the default: case in matching.

```
match (color) {
  | Color.Red => "red"
  | _ => "other"
}
```

The **tuple pattern** consists of one or more patterns separated by commas and surrounded by parens. We have already used them in the [section about tuples](#) above. There they were used in `def` expressions.

```
// matches any pair, binding its elements
// to specified variables
(first_element, second_element)

// matches a pair whose first element is Foo
(Foo, _)
```

The **record pattern** consists of class name and zero or more named patterns separated by commas enclosed in parentheses. It matches a class, whose field values are matched by sub-patterns.

```
class Foo {
  public number : int;
  public name : string;
}

StringOfFoo (f : Foo) : string
{
  if (f.name == "")
    f.number.ToString ()
  else
    f.name
}

// do the same as above
```

```
StringOfFooMatch (f : Foo) : string
{
  match (f) {
    | Foo where (name = "", number = k) =>
      k.ToString ()
    | Foo where (name = s) =>
      s
  }
}
```

It might be doubtful if `string_of_foo_match` is any better than `string_of_foo`. Record patterns are mostly useful when used inside a complex pattern, or when they contain complex patterns.

The **literal pattern** is an integer, character or string constant. It matches the exact specified value.

```
StringOfInt (n : int) : string
{
  match (n) {
    | 0 => "null"
    | 1 => "one"
    | 2 => "two"
    | 3 => "three"
    | _ => "more"
  }
}

IntOfString (n : string) : int
{
  | "null" => 0
  | "one" => 1
  | "two" => 2
  | "three" => 3
  | _ => 42
}
```

Note lack of `match` in the second example. When the function body starts with `|` – the `match` expression is automatically inserted.

The **as pattern** tries to match a value with a pattern enclosed within it, and in case of success binds the value that matched to a specified variable. We will show it later.

There are also two patterns related to types. The first one is the **type enforcement pattern**. **Its job is to give a hint to type inference engine**. It consists of a pattern followed by a colon and a type, like: `x : Foo` or `(x,y) : Foo * Bar`. It requires type of the matched value to be statically known to subtype the type specified.

The other pattern related to type is the **type check pattern**. It is used to check whether the value matched has the specified type. It consists of a variable followed by the keyword `is` and a type. If the runtime type of the value matched is subtype of the type specified, the branch is taken and the matched value is bound to specified variable.

```
match (some_value) {
  | x is SomeType =>
    // use x
  | x is SomeOtherType =>
    // use x
  | _ => ...
```

```
}
```

6.4 Using variants as trees

The example above, while simple, is not the best usage of variants. Variants are best at handling tree-like data structures. A common example of tree data structures are XML documents. However, we will deal with plain binary trees first.

The following example defines the type of trees of integers (representing sets).

```
variant Tree {
  | Node {
    left : Tree;
    elem : int;
    right : Tree;
  }
  | Null
}

// return tree t with element e inserted
Insert (t : Tree, e : int) : Tree
{
  match (t) {
    | Tree.Node (l, cur, r) =>
      if (e < cur)
        Tree.Node (insert (l, e), cur, r)
      else if (e > cur)
        Tree.Node (l, cur, insert (r, e))
      else
        // node already in the tree,
        // return the same tree
        t
    | Tree.Null =>
      Tree.Node (Tree.Null (), e, Tree.Null ())
  }
}

// check if specified integer is in the tree
Contains (t : Tree, e : int) : bool
{
  match (t) {
    | Tree.Node (l, cur, r) when e < cur =>
      Contains (l, e)

    | Tree.Node (l, cur, r) when e > cur =>
      Contains (r, e)

    | Tree.Node => true
    | Tree.Null => false
  }
}
```

6.5 XML trees

As you can see binary trees are not very interesting, so we will go to XML. Whether XML is interesting remains a doubtful question, but at least it is somewhat more practical.

```
variant Node {
  | Text {
    value : string;
  }
  | Element {
    name : string;
    children : list [Node];
  }
}
```

This variant defines a simplistic data structure to hold XML trees. An XML node is either a text node with a specified text inside, or an element node, with a name and zero or more children. A sequence of children is represented as a Nemerle list data structure (Nemerle has even a [special syntax for lists](#)). The type is written here `list [Node]` – a list of nodes. We will learn more about polymorphic variants later.

For example the following tree:

```
<tree>
  <branch>
    <leaf/>
  </branch>
  <branch>
    Foo
  </branch>
</tree>
```

would be represented by:

```
Node.Element ("tree",
  [Node.Element ("branch", [Node.Element ("leaf", [])]),
  Node.Element ("branch", [Node.Text ("Foo")])])
```

Of course XML by itself is just a data format. Using data in the above form wouldn't be too easy. So we want some different internal representation of data, and use XML only to save it or send it over the network.

```
class Refrigerator
{
  minimal_temperature : float;
  content : list [RefrigeratorContent];
}

variant RefrigeratorContent
{
  | Beer { name : string; volume : float; }
  | Chips { weight : int; }
  | Ketchup
}
```

Now we'll write simple XML parsing function.

```

ParseRefrigerator (n : Node) : Refrigerator
{
  | Node.Element ("refrigerator",
    Node.Element ("minimal-temperature", [Node.Text (min_temp)]
      :: content) =>
    Refrigerator (System.Float.Parse (min_temp),
      ParseRefrigeratorContent (content))
  | _ =>
    throw System.ArgumentException ()
}

ParseRefrigeratorContent (nodes : list [Node])
: list [RefrigeratorContent]
{
  | [] => []

  | node :: rest =>
    def food =
      match (node) {
        | Node.Element ("ketchup", []) =>
          RefrigeratorContent.Ketchup ()

        | Node.Element ("beer",
          [Node.Element ("name", [Node.Text (name)]),
          Node.Element ("volume", [Node.Text (volume)])]) =>
          RefrigeratorContent.Beer (name, System.Float.Parse (volume))

        | Node.Element ("chips",
          [Node.Element ("weight", [Node.Text (weight)])]) =>
          RefrigeratorContent.Chips (System.Int32.Parse (weight))

        | _ =>
          throw System.ArgumentException ()
      };
    food :: ParseRefrigeratorContent (rest)
}

```

The reader will easily note that a) this code looks a bit like a junk, b) it can be generated automatically and c) in C# it would be even worse. Later we will learn how to write macros to generate this kind of code automatically.

But let's leave the ideology behind. There are probably few interesting things about this example. The first is the usage of list patterns and constructors. We can check if a list is empty, and if not, deconstruct it with the following code:

```

match (l) {
  | [] =>
    // the list is empty
    // ...
  | head :: rest =>
    // the list isn't empty, the first element
    // of the list is bound to the 'head' variable
    // and the rest of the list to 'rest'
    // ...
}

```

We can also construct new lists with `::` operator – it prepends an element to an existing list. If we know all list elements in advance we can use the `[...]` thing in both expressions and patterns.

The second interesting thing is that we throw an exception in case of problems. We will talk about it later, for now assume, it just terminates the program with an error message.

6.6 Exercises

2.2 (2 points). Write a function that reads XML from specified files and puts it into the `Node` variant defined above. Then write a function to dump your data in a lisp format, something like:

```
(tree
  (branch
    (leaf)
  )
  (branch
    ($text "Foo")
  )
)
```

For an extra point implement indentation of output.

```
(tree
  (branch
    (leaf))
  (branch
    ($text "Foo")))
```

Then copy the Parser functions from above, fix any errors you find in them and try to parse the following file:

```
<?xml version='1.0' encoding='utf-8' ?>
<refrigerator>
  <minimal-temperature>-3.0</minimal-temperature>
  <beer>
    <name>Hyneken</name>
    <volume>0.6</volume>
  </beer>
  <beer>
    <name>Bydweisser</name>
    <volume>0.5</volume>
  </beer>
  <beer>
    <name>Plsner</name>
    <volume>0.5</volume>
  </beer>
  <chips>
    <weight>500</weight>
  </chips>
  <ketchup/>
</refrigerator>
```

Warning: you do not need to write the XML parser. You should not do it, actually. Use the `System.Xml` namespace from the .NET Framework. In order to link with the `System.Xml` library you need to compile with the `-r:System.Xml` option. For example:

```
ncc -r System.Xml myprogram.n
```

should do.

7 Grok The rest

This page is a part of the [Grokking Nemerle](#) tutorial.

7.1 Named parameters

This feature (in this form) comes from python. When you call a function you can name some of its parameters (which allows putting them in a different order than in the definition).

```
frobnicate (foo : int, do_qux : bool,
           do_baz : bool,
           do_bar : bool) : int
{
    // this is completely meaningless
    if (do_qux && !do_baz) foo * 2
    else if (do_bar) foo * 7
    else if (do_baz) foo * 13
    else 42
}

Main () : void
{
    // Parameters' names can be omitted.
    def res1 = frobnicate (7, true, false, true);

    // This is the intended usage -- the first
    // (main) parameter comes without a name
    // and the following flags with names
    def res2 = frobnicate (7, do_qux = true,
                          do_baz = false,
                          do_bar = true);

    // You can however name every parameter:
    def res3 = frobnicate (foo = 7, do_qux = true,
                          do_baz = false, do_bar = true);

    // And permute them:
    def res3 = frobnicate (do_qux = true, do_bar = true,
                          do_baz = false, foo = 7);

    // You can also omit names for any number of leading
    // parameters and permute the trailing ones.
    def res2 = frobnicate (7, true,
                          do_bar = true,
                          do_baz = false);

    ()
}
```

The rules behind named parameters are simple:

- named parameters must come after all unnamed (positional) parameters
- positional parameters are assigned first
- the remaining parameters are assigned based on their names

Named parameters can be used only when names of parameters are known (which basically mean they do not work in conjunction with functional values).

7.2 The void literal

The void literal is written: `()`.

The void literal is quite a tricky thing, since it represents the only value of type `void`, which, judging from the name, should be ergh... `void`. In fact, in some other functional languages this type is called `unit`, but in Nemerle the name comes from `System.Void` and the `void` type is an alias for it.

In C# the `void` type is used as a return type of functions. It does not make much sense to use it elsewhere. It could be needed, however, if, for example, you want to use the `Hashtable [a, b]` type as a set representation of strings. You can use `Hashtable [string, void]` then. And this is the place to use the void value – when you call a set method, you need to pass something as a value to set – and you pass the void value.

You can also use the void value to return it from a function – as you remember, the return value of function is the last expression in its body. However, in most cases the last expression will already have the right `void` type.

7.3 Operator overloading

You can overload existing operators or add new ones simply by specifying them as a static method in some type. The name of method should be some quoted (with `@`) operator. For example, the following class definition:

```
class Operand {
    public val : int;
    public this (v : int) { val = v }

    public static @<-< (x : Operand, y : Operand) : Operand {
        Operand (x.val + y.val);
    }
}
```

contains the `<-<` binary operator processing `Operand` objects. It can be used like:

```
def x = Operand (2);
def y = Operand (3);
def z = x <-< y;
assert (z.val == 5);
```

Unary operators can be created by giving only one parameter to the method.

7.4 Macros

Nemerle has very powerful code-generating macros. They are more akin to Lisp macros than macros found in C preprocessor. We are not going to explain here how to write macros (if you are curious, please see [macros tutorial](#)), but will describe a few often used macros.

7.4.1 Core language constructs

First of all, `if`, `while`, `for`, `foreach`, `when`, `using`, `lock`, etc. are all macros.

When you write code like:

```
using (stream = System.IO.FileStream ("file.txt"))
{
    lock (this) {
        ...
    }
}
```

it first gets transformed into

```
def stream = System.IO.FileStream ("file.txt", System.IO.FileMode.Open);
try {
    System.Threading.Monitor.Enter (this);
    try {
        ...
    } finally {
        System.Threading.Monitor.Exit (this)
    }
} finally {
    def disp = (stream : System.IDisposable);
    when (disp != null)
        disp.Dispose ()
}
}
```

The transformation is done by macros, which also introduce *using* and *lock* syntax to the language.

The greatest thing is that programmer is allowed to define his own macros, introducing his **own syntax**.

7.4.2 Design by contract

Design by contract macros allows you to specify assertions about your program. In other languages they are usually laying around in comments, while in Nemerle you can explicitly decorate code. This way *contracts* are instantly checked during program execution.

7.4.3 Other examples

Other examples of macros:

- `printf`, `sprintf` – uses a similar syntax to the same function in C, but is checked at the compile time
- `scanf` – likewise
- `print` – it does the `$`-expansion known from shell or perl
- `assert` – much like the C macro

You can see a [page about macro usage](#) for some other high level examples.

8 Grok Object oriented programming

This page is a part of the [Grokking Nemerle](#) tutorial.

Once again a [definition](#) from Wikipedia.

OOP is all about **objects**. Objects consist of some data and methods to operate on this data. In functional programming we have functions (algorithm) and data. The things are separate. One can think about objects as of records (structures) with attached functions.

Nemerle allows programmers to program in this style. Moreover, the class library is very deeply object oriented. Therefore OOP is unavoidable in Nemerle.

8.1 Back in the Refrigerator

While talking about XML, we have shown an example of a refrigerator. It was a degenerated object – a record. Record is a bunch of data, or an object without methods. Now we will try to extend it a bit.

```
class Refrigerator
{
    public mutable minimal_temperature : float;
    public mutable content : list [RefrigeratorContent];

    public AddContent (elem : RefrigeratorContent) : void
    {
        content = elem :: content
    }
}

variant RefrigeratorContent
{
    | Beer { name : string; volume : float; }
    | Chips { weight : int; }
    | Ketchup
}
```

Now, in addition to fields with content and temperature, the refrigerator has a method for adding new content. The definition of method looks much like the definition of a function within a module.

It is quite important to understand the difference between classes and objects. Classes are type definitions, while objects (class instances) are values. Classes define templates to create new objects.

Non static methods defined in class **C** have access to a special value called **this**. It has type **C** and is immutable. It refers to the current object. You can use the dot (.) operator to access fields of current object. You can see how **this** is used in the **AddContent** method.

The **this** value is quite often used in object oriented programming. Therefore it can be omitted for brevity. For example:

```
public AddContent (elem : RefrigeratorContent) : void
{
    content = elem :: content
}
```

However, if the method had a formal parameter called **content**, or a local value with such a name, one would need to use **this.content** to access the field.

There is one special method in a class called a *constructor*. It is called whenever you request creation of new instance of an object. It the responsibility of the constructor to setup values of all fields within an object. Fields start with value of **null**, **0** or **0.0**.

```
public this ()
{
    minimal_temperature = -273.15;
    content = [];
}
```

Constructors can take parameters. For example, if we wanted to set the `minimal_temperature` already at the object construction stage, we could write:

```
public this (temp : float)
{
    minimal_temperature = temp;
    content = [];
}
```

For variant options Nemerle provides a default constructor, that assigns each field. If you do not provide a constructor for a regular class, an empty one is generated. If you need the field-assigning constructor, you can use `[Record]` attribute, like this:

```
[Record]
class Foo
{
    x : int;
    y : float;
    z : string;
}
```

The following constructor is generated:

```
public this (x : int, y : float, z : string)
{
    this.x = x;
    this.y = y;
    this.z = z;
}
```

The constructor is called with the name of the class when creating new objects. Other methods are called using the dot operator. For example in `refr.AddContent (Ketchup ())` the `refr` is passed to the `AddContent` method as the `this` pointer and `Ketchup ()` is passed as `elem` formal parameter.

```
module Party {
    Main () : void
    {
        def refr = Refrigerator ();
        refr.AddContent (Beer ("Tiskie", 0.60));
        refr.AddContent (Ketchup ());
    }
}
```

Fortunately, objects are not just a fancy notation for a function application on records.

8.2 Inheritance

Classes can **inherit** from other classes. The fact that a class B inherits from a class A has a few consequences. The first one is that B has now all the fields and methods of A. The second one is that B is now **subtype** of A. This means that all the functions operating on A can now also operate on B.

Class A is often called **parent** or **base** class of B (which is **derived** class).

In the following example we can see how we can call methods defined in the base class (`AddContent`), as well as from the derived class (`MoveToBedroom`).

Static methods and the constructor are not derived. The parameterless constructor is defined in this example. As the first thing to do, it calls parameterless constructor of the base class. It does it, so the derived fields are initialized first. Then it initializes the new fields.

The call to the parameterless parent constructor is in fact redundant. When there is no call to the parent class constructor, such a parameterless parent constructor call is assumed in the first line of a constructor.

```
class RefrigeratorWithRolls : Refrigerator
{
    public mutable position_x : int;
    public mutable position_y : int;

    public MoveBy (dx : int, dy : int) : void
    {
        position_x += dx;
        position_y += dy;
    }

    public MoveToBedroom () : void
    {
        position_x = 42;
        position_y = -42;
    }

    public this ()
    {
        base ();

        position_x = 0;
        position_y = 0;
    }
}

class TheDayAfter
{
    static Main () : void
    {
        def refr = RefrigeratorWithRolls ();
        for (mutable i = 0; i < 10; ++i)
            refr.AddContent (Beer ("Liech", 0.5));
        refr.MoveToBedroom ();
        // drink
    }
}
```

8.3 Virtual calls

The funny part begins where objects can react to calling some methods in a way dependent on the class of the object. It is possible to define virtual methods, which means they can be redefined in a derived class. Then when we have a function working on the base class, whenever it calls the virtual method, an appropriate method is selected based on actual object type.

This feature is called polymorphism in object-oriented world. We will, however, mostly use this word for another kind of polymorphism – parametric polymorphism.

When one wants to override a virtual method from a base class, it needs to be declared with the `override` modifier.

```
using Nemerle.IO;

class Refrigerator
{
    public mutable minimal_temperature : float;
    public mutable content : list [RefrigeratorContent];

    public virtual AddContent (elem : RefrigeratorContent) : void
    {
        content = elem :: content
    }

    public this ()
    {
        minimal_temperature = -273.15;
        content = [];
    }
}

class RestrictedRefrigerator : Refrigerator
{
    public override AddContent (elem : RefrigeratorContent) : void
    {
        match (elem) {
            | Ketchup =>
                // don't add!
                printf ("Ketchup is not healthy!\n")
            | _ =>
                content = elem :: content
        }
    }

    public this ()
    {
    }
}

```

Here we can see how the `AddKetchup` calls different methods depending on actual object type. The first call adds ketchup, the second call refuses to do so.

```
module Shop
{
    AddKetchup (refr : Refrigerator) : void
    {
        refr.AddContent (Ketchup ())
    }
}

```

```
}

Main () : void
{
    def r1 = Refrigerator ();
    def r2 = RestrictedRefrigerator ();
    AddKetchup (r1);
    AddKetchup (r2);
}
}
```

8.4 Interfaces

The .NET Framework supports only single inheritance. This means that any given class can derive from just one base class. However, it is sometimes needed for a class to be two or more different things depending on context. .NET supports it (just like Java) through interfaces. An interface is a contract specifying a set of methods given class should implement. A class can implement zero or more interfaces (in addition to deriving from some base class).

Implementing interface implies subtyping it. That is if you have a class A implementing I and method taking I as parameter, then you can pass A as this parameter.

Interfaces most commonly state some ability of type. For example, the ability to convert itself to some other type or to compare with some other types.

```
using Nemerle.IO;

interface IPrintable {
    Print () : void;
}

class RefrigeratorNG : Refrigerator, IPrintable
{
    public Print () : void
    {
        printf ("I'm the refrigerator!\n")
    }

    public this ()
    {
    }
}

module RP {
    PrintTenTimes (p : IPrintable) : void
    {
        for (mutable i = 0; i < 10; ++i)
            p.Print ()
    }

    Main () : void
    {
        def refr = RefrigeratorNG ();
        PrintTenTimes (refr)
    }
}
```

The base class must come first after the colon in class definition. Then come interfaces in any order.

9 Grok Parametric polymorphism

This page is a part of the [Grokking Nemerle](#) tutorial.

Parametric polymorphism is a wise way to say that function can operate on values of any type. Kind of `System.Object` and/or `void*` on steroids. This is very much like Generics in C# 2.0 or Java 5.0 and somewhat less like templates in C++.

Both functions and types can be parameterized over types.

9.1 Simple polymorphism

Here we define a list of values of any type (it is also defined in Nemerle standard library but this is not the point here).

```
variant list [T] {
  | Cons { hd : T; tl : list [T]; }
  | Nil
}
```

Here we used `T` as a type parameter to the list type. The ML-lovers would rather write `'a` instead and read it *alpha* (it is a convention to use identifiers starting with an apostrophe for type parameters). They are allowed to do so, as the apostrophe is allowed in identifiers in Nemerle. We will however stick to a C++-like convention.

In the body of the `list` definition `T` can be used like any other type name. It is called **type variable** in this scope.

Next we define the method parameterized on a type (it is reflected by listing `[something]` after `Head`). Since the algorithm of taking the head out of the list does not depend on the type of the actual values stored in the list, we can use the same `T`, but we could have used `'b` or `foobar42` as well. This method for `list[int]`, `list[string]` and even `list[list[int]]` would look exactly the same. Therefore we can use generic `T` type.

You can see that the type of elements of the list (a parameter in `list[T]`) is used as return type of this method. This way we can ensure that we take an `int` out of `list[int]` and not some generic `System.Object`.

```
class List {
  public static Head[T] (l : list[T]) : T
  {
    match (l) {
      | Cons (hd, _) => hd
      | Nil =>
        throw System.ArgumentException ()
    }
  }
}
```

9.2 Constraints on type variables

It is sometimes necessary for types to be substituted for type variables to conform to some specific interface. This concept is known as F-bounded polymorphism. We will address this issue in more detail, as it is probably new for most readers.

For example the elements stored in a tree need to provide a comparison method. Thus, we can define an appropriate interface and then require `'a` in `Tree['a]` to conform to it:

```

interface IComparable ['a] {
  CompareTo (elem : 'a) : int;
}

variant Tree['a]
  where 'a : IComparable['a]
{
  | Node {
    left : Tree['a];
    elem : 'a;
    right : Tree['a];
  }
  | Tip
}

```

In fact the `IComparable` interface is already defined in the standard library, but that is not the point.

Now, once we ensured that elements in the tree conform to `IComparable`, we can use the `CompareTo` method. For example, to insert a thing into the tree we can use the following function:

```

module TreeOperations {
  public Insert['a] (t : Tree['a], e : 'a) : Tree['a]
    where 'a : IComparable['a]
  {
    match (t) {
      | Node (l, c, r) =>
        if (e.CompareTo (c) < 0)
          Node (Insert (l, e), c, r)
        else if (e.CompareTo (c) > 0)
          Node (r, c, Insert (r, e))
        else
          Node (r, e, l)
      | Tip =>
        Node (Tip (), e, Tip ())
    }
  }
}

```

The people familiar with C# or Java will ask why not simply use something like:

```

interface IComparable {
  CompareTo (elem : IComparable) : int;
}

variant Tree
{
  | Node {
    left : Tree;
    elem : IComparable;
    right : Tree;
  }
  | Tip
}

```

But this is only half good. The most often case for using a tree is to store elements of some specific type, for example strings. We don't want integers and strings to be stored in the same tree, for the very simple reason

that we cannot compare integer with string in a reasonable way. Well, even if we could, we plainly cannot predict what other types beside integers and strings implement `IComparable` and thus can be passed to string's `CompareTo`.

But the design above makes it impossible to ensure statically whether we're using the tree with correct types. When inserting nodes to the tree we upcast them all to `IComparable`. We will get runtime exception when string's `CompareTo` is passed integer argument. The second drawback is that when we extract elements out of the tree, we need to downcast them to a specific type. This is second possibility for runtime errors.

To fully understand this issue please look at the following example:

```
interface IFrobincatable {
    Froblicate (x : int) : void;
}

class C1 : IFrobincatable
{
    public this () {}
    public Froblicate (_ : int) : void {}
}

class C2 : IFrobincatable
{
    public this () {}
    public Froblicate (_ : int) : void {}
}

module M {
    f1['a] (o : 'a) : 'a
        where 'a : IFrobincatable
    {
        o.Froblicate (3);
        o
    }

    f2 (o : IFrobincatable) : IFrobincatable
    {
        o.Froblicate (3);
        C1 ()
    }

    Main () : void
    {
        def x1 = f1 (C1 ()); // x1 : C1
        def x2 = f1 (C2 ()); // x2 : C2
        def x3 = f2 (C1 ()); // x3 : IFrobincatable
        def x4 = f2 (C2 ()); // x4 : IFrobincatable

        ()
    }
}
```

In the `Main` function you can see what types get the `x1`, `x2` etc values.

10 Grok Exceptions

This page is a part of the [Grokking Nemerle](#) tutorial.

In Nemerle you can use exceptions in a similar way you would use them in C#. The only difference is the syntax for the catch handlers – it looks more or less like matching, you can even use the `_` to denote any exception that has to be caught.

```
some_function (foo : string) : void
{
    when (foo == null)
        throw System.ArgumentException ("foo");
    // do something
}

some_other_function () : void
{
    try {
        some_function ()
    }
    catch {
        | e is System.ArgumentException =>
            printf ("a problem:\n%s\n", e.Message)
    }
    finally {
        printf ("yikes")
    }
}

some_other_function.2 () : void
{
    try {
        some_function ()
    } catch {
        | e is System.ArgumentException =>
            printf ("invalid argument\n")
        | _ =>
            // just like:
            // | _ is System.Exception =>
            printf ("a problem\n")
    }
}

some_other_function.3 () : void
{
    def f = open_file ();
    try {
        some_function ()
    }
    finally {
        f.close ()
    }
}

class MyException : System.Exception {
    public this () {}
}
```

11 Grok Properties, indexers, delegates and events

This page is a part of the [Grokking Nemerle](#) tutorial.

11.1 Properties

Properties are syntactic sugar for get/set design pattern commonly found in Java. Accessing a property looks like accessing a field, but under the hood it is translated to a method call.

```
class Button {
  text : string;
  public Text : string {
    get { text }
    set { text = value; Redraw () }
  }
}

def b = Button ();
b.Text = b.Text + "..."
```

11.2 Indexers

Indexers are other form of properties, but instead of exposing field access syntax array access syntax is used. All .NET indexers have names, though one indexer can be marked as the **default indexer** (well, in fact one name of indexer is marked as default, but there can be several indexers with that name, subject to the regular overloading rules).

For example the `System.Hashtable` default indexer is called `Item` and `System.String` one is called `Chars`. The C# language allows definition of default indexer only. Nemerle allows also other indexers (like VB.NET). In the current release default indexer is always `Item`.

```
class Table {
  store : array [array [string]];
  public Item [row : int, column : int] : string
  {
    get { store[row][column] }
    set { store[row][column] = value }
  }
}

def t = Table ();
t[2, 3] = "foo";
t[2, 2]
```

11.3 Delegates

Delegates are half-baked functional values. In fact there is little place for usage of delegates in Nemerle itself. However other .NET languages all speak delegates, so they are good way to expose functional values for them and *vice versa*.

Delegates are in essence named functional types. They are defined with `delegate` keyword:

```
delegate Foo (- : int, - : string) : void;
delegate Callback () : void;
```

Later delegate name can be used to construct delegate instances. Any functional value of corresponding type can be used to create delegate instance (local functions and lambda expressions in particular).

Delegate instance can be invoked using regular function call syntax, as well as the `Invoke` special method. Please consult class library documentation for details.

The `+=` operator has special meaning on delegate instances – it calls the `System.Delegate.Combine` method that makes one function that calls two other in sequence. This operator is probably more useful with events.

```
module X {
  some_fun () : void
  {
    mutable f = Foo (fun (_, _) {});
    f (3, "foo"); // same as f.Invoke (3, "foo");
    f += fun (_, s) { System.Console.WriteLine (s) };
    f (42, "bar");
  }
}
```

11.4 Events

Events are kind of specialized properties for delegates. They are used in a GUI system for connecting signals (setting function to call when a button is pressed etc.). They can be also used to call user defined function on certain events, like class being loaded by the runtime system.

Events are defined like fields, but they are marked with the `event` keyword, and always have a delegate type. Inside the class events are seen as fields of these delegate type, but outside only the `+=` and `-=` operators are available. They are used to connect and disconnect delegates. Implicit conversion from functional values to delegate instances is provided.

```
class Button {
  public event OnClick : Callback;
}

def b = Button ();
b.OnClick += fun () { ... }
```