

Macros

January 21, 2006

Contents

1	Macros	1
1.1	Intro	1
1.2	Key features	1
1.3	Using macros	1
1.4	Writing macros	2
2	MacroUse	2
2.1	Introduction	2
2.2	Design by contract	2
2.3	Compile-time validation of embedded languages	3
2.4	Partial evaluation	3
2.5	Concurrency constructs	3
2.6	Design patterns	3
2.7	Aspects-Oriented programming	3
2.8	More	4
3	Macros tutorial	4
3.1	What exactly is a macro?	4
3.2	Defining a new macro	4
3.2.1	Compiling a simplest macro	4
3.2.2	Exercise	5
3.3	Operating on syntax trees	5
3.3.1	Quotation operator	6
3.3.2	Matching subexpressions	6
3.3.3	Base elements of grammar	6
3.3.4	Constructs with variable amount of elements	7
3.3.5	Exercise	8
3.4	Adding new syntax to the compiler	8
3.4.1	Exercise	8
3.5	Macros in custom attributes	8
3.5.1	Executing macros on type declarations	8
3.5.2	Manipulating type declarations	9
3.5.3	Execution stages	11
3.6	Reference to more advanced aspects	12
3.6.1	Hygiene and alpha-renaming of identifiers	12
4	Partial evaluation	14
4.1	Partial evaluation through meta-programming	14
4.1.1	Power function - classic example	14
4.1.2	Permutation algorithm	15
4.2	Rewriting interpreter into compiler	17
4.2.1	Virtual machine	17
4.2.2	Language	18
4.2.3	Interpreter	18
4.2.4	Staged interpreter	19

4.2.5	Usage	19
5	Design patterns	20
5.1	Intro	20
5.2	Proxy design pattern	20
5.2.1	What we want to achieve	20
5.2.2	How we do this with a macro	22
5.2.3	Macro included in standard library	23
5.3	Singleton design pattern	23
5.3.1	What do we want to achieve?	23
5.3.2	How do we implement it in macro?	24
6	Syntax extensions	26
6.1	Expression level extensions	26
6.2	Raw token extensions	26
6.2.1	The <i>parentheses tree</i> concept	26
6.2.2	Parentheses tokens	27
6.2.3	Passing token groups to the macro	27
7	Macro tips	27
7.1	Quotations	28
7.1.1	Match cases	28
7.2	Compiler API available from macros	28
8	Defining types from inside macros	29
8.1	Basics	29
8.2	Longer example	30
8.3	Stages of TypeBuilder	31

1 Macros

1.1 Intro

You can think about macros as of a system of compile-time transformations and automatic generation of code with regard to some rules. It can be used either to automate manipulations performed on similar data-types and fragments of code or to add syntax shortcuts to the language, optimize and make some computations safer by moving them from runtime to compile-time.

The idea of making simple inline operations on the code comes from preprocessor macros, which many languages (especially C, C++) have contained since early times of compiler design. We are following them in the direction of much more powerful, and at the same time more secure (type-safe) solutions like Haskell Template Meta-programming.

1.2 Key features

- built-in ability of extending language syntax
- algorithmic generation of code, which may be dependent on external factors, like XML files, script programs, databases or even remote web sites
- completing datatypes with automatically created methods for performing arbitrary operations, e.g. reading datasets from a file, sending them through the network or storing in an SQL database
- using the context of compilation provided by compiler's internal structures, like line numbers (for error reporting), names of processed type, function, class or namespace and accessing datatype definitions by name
- creating functions, whose amount and type of arguments depend on each other (like C printf) with full type-checking during compile-time

1.3 Using macros

You can take a look at the most [appealing examples](#) of macro use. It contains a description of how we incorporate methodologies as Design By Contract, compile-time verification of SQL statements and Aspects-Oriented Programming into Nemerle.

Most useful macros usually gets into the standard tool library of application. Nemerle compiler also have such macros, and they are by default available together with the distribution. Their list is kept on [class library](#) page.

1.4 Writing macros

- [Macros tutorial](#) describing how to begin writing macros
- [Partial evaluation](#) tutorial and examples.
- [Simple expression evaluator](#) example
- Automating code generation in [design patterns](#) usage.
- [Syntax extensions](#) description.
- Some [tips](#) for macro writers.
- [Defining types from inside macros](#)

2 MacroUse

The most interesting features we implemented using macros

2.1 Introduction

This page is dedicated to the features of Nemerle and its library, which have been implemented using our [meta-programming facilities](#).

Understanding how macros work is not necessary for using any of them. It would be useful only for knowing that those examples are just a tip of an iceberg called meta-programming and that you can easily implement even nicer things.

Please refer to [class library](#) documentation for a reference list of implemented macros.

2.2 Design by contract

Languages like [Eiffel](#) or [Spec#](#) incorporate a methodology called [Design by Contract](#) to reason about programs, libraries, methods. It allows to write more secure and correct software and specify its behavior.

A quick example below shows how it looks in Nemerle with its syntax extensions. Note that the **invariant**, **requires** and **ensures** syntax is added to the scope by opening *Nemerle.Assertions* namespace containing design by contract macros.

```
using Nemerle.Assertions;

class BeerControl
invariant m_temperature <= 6.0
{
    public DrinkUsingAStaw (how_much : float) : void
    requires how_much > 0.0
    {
        m_amount -= how_much
    }
}
```

```
}  
  
public DissipationHandler () : void  
ensures m_amount > 0.0  
{  
    when (AReallySmallNumber < m_amount) {  
        m_temperature += (Room.Temperature - m_temperature) / 100.0;  
        m_amount -= 0.1; // loss due to the evaporation  
    }  
}  
  
private mutable m_temperature : float;  
private mutable m_amount : float;  
}
```

See here for [more...](#)

2.3 Compile-time validation of embedded languages

In many programming tasks there is a need for using [domain-specific languages](#) for performing some specialized operations. The examples are regular expressions used for searching and matching parts of text, formatting strings of `printf` function or SQL for obtaining data from database. All those languages have their own syntax and validity rules.

Most of DSLs are used inside a general-purpose language by embedding programs written in them into strings. For example, to query a database about elements in some table, one writes an SQL statement and sends it to the database provider as a string. The common problem with this approach is verifying correctness of embedded programs - if the syntax is valid, if types of the variables used match, etc. Unfortunately, in most cases all those checks are performed at runtime, when a particular program is expected to execute, but fails with a syntax or invalid cast error.

All this happen, because the compiler of our general-purpose language treats DSL programs just as common strings. It is not surprising though - it was not designed to verify any particular domain-specific language - but it would be nice to do it before runtime. In Nemerle we can use macros to handle some of the strings in a special way - for example run a verification function against them.

This mechanism is very general and it is used in some parts of Nemerle standard library (like regular expression matching, `printf`-like functions). Please refer to our [SQL macros](#) description for more information.

2.4 Partial evaluation

[Partial evaluation](#) is a process of specializing given program with some of its inputs, which are known statically. This way we obtain a new program, which is the optimized version of the general one. Implementation of this technique often involves program generation and rewriting.

Macros can be used as a convenient tool for partial evaluation in a similar way to [multi-stage programming](#).

A [tutorial](#) explains how to use macros in such a setting. There are examples describing specialization technique (power function, permutation algorithm) and others.

2.5 Concurrency constructs

Most of the features of [Polyphonic C#](#) were implemented in our library using Nemerle macros. You can use these specialized features for safer multithreading application, by simply importing `Nemerle.Concurrency` namespace.

Currently you can see some [examples](#).

2.6 Design patterns

Some [design patterns](#) can be greatly automated by macros. Instead of writing massive amount of code, programmer can just generate necessary methods, fields, etc. or transform handwritten classes according to the needs, reducing maintainance costs and bugs introduced by commonly used copy & paste method.

2.7 Aspects-Oriented programming

As for now see [this](#).

2.8 More

More examples can be found at the [class library](#) documentation page.

3 Macros tutorial

Nemerle type-safe macros

3.1 What exactly is a macro?

Basically every macro is a function, which takes a fragment of code as parameter(s) and returns some other code. On the highest level of abstraction it doesn't matter if parameters are function calls, type definitions or just a sequence of assignments. The most important fact is that they are not common objects (e.g. instances of some types, like integer numbers), but their internal representation in the compiler (i.e. syntax trees).

A macro is defined in the program just like any other function, using common Nemerle syntax. The only difference is the structure of the data it operates on and the way in which it is used (executed at compile-time).

A macro, once created, can be used to process some parts of the code. It's done by calling it with block(s) of code as parameter(s). This operation is in most cases indistinguishable from a common function call (like $f(1)$), so a programmer using a macro would not be confused by unknown syntax. The main concept of our design is to make the usage of macros as transparent as possible. From the user point of view, it is not important if particular parameters are passed to a macro, (which would process them at the compile-time and insert some new code in their place), or to an ordinary function.

3.2 Defining a new macro

Writing a macro is as simple as writing a common function. It looks the same, except that it is preceded by a keyword `macro` and it lives at the top level (not inside any class). This will make the compiler know about how to use the defined method (i.e. run it at the compile-time in every place where it is used).

Macros can take zero (if we just want to generate new code) or more parameters. They are all elements of the language grammar, so their type is limited to the set of defined syntax objects. The same holds for a return value of a macro.

Example:

```
macro generate_expression ()
{
  MyModule.compute_some_expression ();
}
```

This example macro does not take any parameters and is used in the code by simply writing `generate_expression ()`; . The most important is the difference between `generate_expression` and `compute_some_expression` - the

first one is a function executed by the compiler during compilation, while the latter is just some common function that must return syntax tree of expressions (which is here returned and inserted into program code by `generate_expression`).

3.2.1 Compiling a simplest macro

In order to create and use a macro you have to write a library, which will contain its executable form. You simply create a new file `mymacro.n`, which can contain for example

```
macro m () {
  Nemerle.IO.printf ("compile-time\n");
  <[ Nemerle.IO.printf ("run-time\n") ]>;
}
```

and compile it with command

```
ncc -r Nemerle.Compiler.dll -t:dll mymacro.n -o mymacro.dll
```

Now you can use `m()` in any program, like here

```
module M {
  public Main () : void {
    m ();
  }
}
```

You must add a reference to `mymacro.dll` during compilation of this program. It might look like

```
ncc -r mymacro.dll myprog.n -o myprog.exe
```

3.2.2 Exercise

Write a macro, which, when used, should slow down the compilation by 5 seconds (use `System.Timers` namespace) and print the version of the operating system used to compile program (use `System.Environment` namespace).

3.3 Operating on syntax trees

Definition of function `compute_some_expression` might look like:

```
using Nemerle.Compiler.Parsertree;

module MyModule
{
  public mutable debug_on : bool;

  public compute_some_expression () : PExpr
  {
    if (debug_on)
      <[ System.Console.WriteLine ("Hello, I'm debug message") ]>
    else
```

```

    <[ () ]>
  }
}

```

The examples above show a macro, which conditionally inlines expression printing a message. It's not quite useful yet, but it has introduced the meaning of compile-time computations and also some new syntax used only in writing macros and functions operating on syntax trees. We have written here the `<[...]>` constructor to build a syntax tree of expression (e.g. `'()'`).

3.3.1 Quotation operator

`<[...]>` is used to both construction and decomposition of syntax trees. Those operations are similar to quotation of code. Simply, everything which is written inside `<[...]>`, corresponds to its own syntax tree. It can be any valid Nemerle code, so a programmer does not have to learn internal representation of syntax trees in the compiler.

```

macro print_date (at_compile_time)
{
  match (at_compile_time) {
    | <[ true ]> => MyModule.print_compilation_time ()
    | _ => <[ WriteLine (DateTime.Now.ToString ()) ]>
  }
}

```

The quotation alone allows using only constant expressions, which is insufficient for most tasks. For example, to write function `print_compilation_time` we must be able to create an expression based on a value known at the compile-time. In next sections we introduce the rest of macros' syntax to operate on general syntax trees.

3.3.2 Matching subexpressions

When we want to decompose some large code (or more precisely, its syntax tree), we must bind its smaller parts to variables. Then we can process them recursively or just use them in an arbitrary way to construct the result.

We can operate on entire subexpressions by writing `$(...)` or `$ID` inside the quotation operator `<[...]>`. This means binding the value of `ID` or the interior of parenthesized expression to the part of syntax tree described by corresponding quotation.

```

macro for (init, cond, change, body)
{
  <[
    $init;
    def loop () : void {
      if ($cond) { $body; $change; loop() }
      else ()
    };
    loop ()
  ]>
}

```

The above macro defines function `for`, which is similar to the loop known from C. It can be used like this

```

for (mutable i = 0, i < 10, i++, printf ("%d", i))

```

Later we show how to extend the language syntax to make the syntax of `for` exactly as in C.

3.3.3 Base elements of grammar

Sometimes quoted expressions have literals inside of them (like strings, integers, etc.) and we want to operate on their value, not on their syntax trees. It is possible, because they are constant expressions and their runtime value is known at the compile-time.

Let's consider the previously used function `print_compilation_time`.

```
using System;
using Nemerle.Compiler.Parsertree;

module MyModule {
  public print_compilation_time () : PExpr
  {
    <[ System.Console.WriteLine ($(DateTime.Now.ToString () : string)) ]>
  }
}
```

Here we see some new extension of splicing syntax where we create a syntax tree of string literal from a known value. It is done by adding `: string` inside the `$(...)` construct. One can think about it as of enforcing the type of spliced expression to a literal (similar to common Nemerle type enforcement), but in the matter of fact something more is happening here - a real value is lifted to its representation as syntax tree of a literal.

Other types of literals (`int`, `bool`, `float`, `char`) are treated the same. This notation can be used also in pattern matching. We can match constant values in expressions this way.

There is also a similar schema for splicing and matching variables of a given name. `$(v : name)` denotes a variable, whose name is contained by object `v` (of special type `Name`). There are some good [reasons](#) for encapsulating a real identifier within this object.

3.3.4 Constructs with variable amount of elements

You might have noticed, that Nemerle has a few grammar elements, which are composed of a list of subexpressions. For example, a sequence of expressions enclosed with `{ .. }` braces may contain zero or more elements.

When splicing values of some expressions, we would like to decompose or compose such constructs in a general way - i.e. obtain all expressions in a given sequence. It is natural to think about them as if a list of expressions and to bind this list to some variable in meta-language. It is done with special syntax `..`:

```
mutable exps = [ <[ printf ("%d ", x) ]>, <[ printf ("%d ", y) ]> ];
exps = <[ def x = 1 ]> :: <[ def y = 2 ]> :: exps;
<[ {.. $exps } ]>
```

We have used `{ .. $exps }` here to create the sequence of expressions from list `exps : list<Expr>`. A similar syntax is used to splice the content of tuples (`(.. $elist)`) and other constructs, like `array []`:

```
using Nemerle.Collections;

macro castedarray (e) {
  match (e) {
    | <[ array [.. $elements] ]> =>
      def casted = List.Map (elements, fun (x) { <[ ($x : object) ]> });
      <[ array [.. $casted] ]>
    | _ => e
  }
}
```

If the exact number of expressions in tuple/sequence is known during writing the quotation, then it can be expressed with

```
<[ $e.1; $e.2; $e.3; x = 2; f () ]>
```

The `..` syntax is used when there are `e_i : Expr` for $1 \leq i \leq n$.

3.3.5 Exercise

Write a macro `rotate`, which takes two parameters: a pair of floating point numbers (describing a point in 2D space) and an angle (in radians). The macro should return a new pair – a point rotated by the given angle. The macro should use as much information as is available at the compile-time, e.g. if all numbers supplied are constant, then only the final result should be inlined, otherwise the result must be computed at runtime.

3.4 Adding new syntax to the compiler

After we have written the `for` macro, we would like the compiler to understand some changes to its syntax. Especially the C-like notation

```
for (mutable i = 0; i < n; --i) {
    sum += i;
    Nemerle.IO.printf ("%d\n", sum);
}
```

In order to achieve that, we have to define which tokens and grammar elements may form a call of `for` macro. We do that by changing its header to

```
macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")") body)
```

The `syntax` keyword is used here to define a list of elements forming the syntax of the macro call. The first token must always be a unique identifier (from now on it is treated as a special keyword triggering parsing of defined sequence). It is followed by tokens composed of operators or identifiers passed as string literals or names of parameters of macro. Each parameter must occur exactly once.

Parsing of syntax rule is straightforward - tokens from input program must match those from definition, parameters are parsed according to their type. Default type of a parameter is `Expr`, which is just an ordinary expression (consult Nemerle grammar in [Reference](#)). All allowed parameter types will be described in the extended version of reference manual corresponding to macros.

3.4.1 Exercise

Add a new syntactic construct `forpermutation` to your program. It should be defined as the macro

```
macro forp (i, n : int, m : int, body)
```

and introduce syntax, which allows writing the following program

```
mutable i = 0;
forpermutation (i in 3 to 10) Nemerle.IO.printf ("%d\n", i)
```

It should create a random permutation `p` of numbers `x_j`, $m \leq x_j \leq n$ at the compile-time. Then generate the code executing body of the loop `n - m + 1` times, preceding each of them with assignment of permutation element to `i`.

3.5 Macros in custom attributes

3.5.1 Executing macros on type declarations

Nemerle macros are simply plugins to the compiler. We decided not to restrict them only to operations on expressions, but allow them to transform almost any part of program.

Macros can be used within custom attributes written near methods, type declarations, method parameters, fields, etc. They are executed with those entities passed as their parameters.

As an example, let us take a look at `Serializable` macro. Its usage looks like this:

```
[Serializable]
class S {
    public this (v : int, m : S) { a = v; my = m; }
    my : S;
    a : int;
}
```

From now on, `S` has additional method `Serialize` and it implements interface `ISerializable`. We can use it in our code like this

```
def s = S (4, S (5, null));
s.Serialize ();
```

And the output is

```
<a>4</a>
<my>
  <a>5</a>
  <my>
    <null/>
  </my>
</my>
```

The macro modifies type `S` at compile-time and adds some code to it. Also inheritance relation of given class is changed, by making it implement interface `ISerializable`

```
public interface ISerializable {
    Serialize () : void;
}
```

3.5.2 Manipulating type declarations

In general, macros placed in attributes can do many transformations and analysis of program objects passed to them. To see `Serializable` macro's internals and discuss some design issues, let's go into its code.

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance, Nemerle.MacroTargets.Class,
    Inherited = true)]
macro Serializable (t : TypeBuilder)
{
    t.AddImplementedInterface (<[ ISerializable ]>)
}
```

First we have to add interface, which given type is about to implement. But more important thing is the phase modifier `BeforeInheritance` in macro's custom attribute. In general, we separate three [stages of execution](#) for attribute macros. `BeforeInheritance` specifies that the macro will be able to change subtyping information of the class it operates on.

So, we have added interface to our type, we now have to create `Serialize ()` method.

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers, Nemerle.MacroTargets.Class,
                    Inherited = true)]
macro Serializable (t : TypeBuilder)
{
    /// here we list its fields and choose only those, which are not derived
    /// or static
    def fields = t.GetFields (BindingFlags.Instance | BindingFlags.Public |
                             BindingFlags.NonPublic | BindingFlags.DeclaredOnly);

    /// now create list of expressions which will print object's data
    mutable serializers = [];

    /// traverse through fields, taking their type constructors
    foreach (x : IField in fields) {
        def tc = x.GetMemType ().TypeInfo;
        def nm = Macros.UseSiteSymbol (x.Name);
        if (tc != null)
            if (tc.IsValueType)
                /// we can safely print value types as strings
                serializers = <[
                    printf ("<%s>", $(x.Name : string));
                    System.Console.Write ($(nm : name));
                    printf ("</%s>\n", $(x.Name : string));
                ]>
                :: serializers
            else
                /// we can try to check, if type of given field also implements ISerializable
                if (x.GetMemType ().Require (<[ ttype: ISerializable ]>))
                    serializers = <[
                        printf ("<%s>\n", $(x.Name : string));
                        if ($(nm : name) != null)
                            $(nm : name).Serialize ()
                        else
                            printf ("<null/>\n");
                        printf ("</%s>\n", $(x.Name : string));
                    ]>
                    :: serializers
                else
                    /// and finally, we encounter case when there is no easy way to serialize
                    /// given field
                    Message.FatalError ("field '" + x.Name + "' cannot be serialized")
            else
                Message.FatalError ("field '" + x.Name + "' cannot be serialized")
        };
    /// after analyzing fields, we create method in our type, to execute created
    /// expressions
    t.Define (<[ decl: public Serialize () : void
                implements ISerializable.Serialize {
                    .. $serializers
                }
            ]>);
}
```

```
}

```

3.5.3 Execution stages

Analysing object-oriented hierarchy and class members is a separate pass of the compilation. First it creates inheritance relation between classes, so we know exactly all base types of given type. After that every member inside of them (methods, fields, etc.) is being analysed and added to the hierarchy and its type annotations are resolved. After that also the rules regarding implemented interface methods are checked.

For the needs of macros we have decided to distinguish three moments in this pass at which they can operate on elements of class hierarchy. Every macro can be annotated with a stage, at which it should be executed.

- **BeforeInheritance** stage is performed after parsing whole program and scanning declared types, but before building subtyping relation between them. It gives macro a freedom to change inheritance hierarchy and operate on parse-tree of classes and members
- **BeforeTypedMembers** is when inheritance of types is already set. Macros can still operate on bare parse-trees, but utilize information about subtyping.
- **WithTypedMembers** stage is after headers of methods, fields are already analysed and in bound state. Macros can easily traverse entire class space by reflecting type constructors of fields, method parameters, etc. Original parse-trees are no longer available and signatures of class members cannot be changed.

Parameters of attribute macros Every executed attribute macro operates on some element of class hierarchy, so it must be supplied with an additional parameter describing the object, on which macro was placed. This way it can easily query for properties of that element and use compiler's API to reflect or change the context in which it was defined.

For example a method macro declaration would be

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers,
                    Nemerle.MacroTargets.Method)]
macro MethodMacro (t : TypeBuilder, f : MethodBuilder, expr)
{
    // use 't' and 'f' to query or change class-level elements
    // of program
}
```

Macro is annotated with additional attributes specifying respectively the stage in which macro will be executed and the macro target.

The available parameters contain references to class hierarchy elements that given macro operates on. They are automatically supplied by compiler and they vary on the target and stage of given macro. Here is a little table specifying valid parameters for each stage and target of attribute macro.

- Attribute macro targets and parameters
- | MacroTarget | MacroPhase.BeforeInheritance | MacroPhase.BeforeTypedMembers | MacroPhase.WithTypedMembers
- | Class | TypeBuilder | TypeBuilder | TypeBuilder
- | Method | TypeBuilder, ParsedMethod | TypeBuilder, ParsedMethod | TypeBuilder, MethodBuilder
- | Field | TypeBuilder, ParsedField | TypeBuilder, ParsedField | TypeBuilder, FieldBuilder
- | Property | TypeBuilder, ParsedProperty | TypeBuilder, ParsedProperty | TypeBuilder, PropertyBuilder
- | Event | TypeBuilder, ParsedEvent | TypeBuilder, ParsedEvent | TypeBuilder, EventBuilder

- | Parameter | TypeBuilder, ParsedMethod, ParsedParameter | TypeBuilder, ParsedMethod, ParsedParameter | TypeBuilder, MethodBuilder, ParameterBuilder
- | Assembly | (none) | (none) | (none)

The intuition is that every macro has parameter holding its target and additionally objects containing it (like TypeBuilder is available in most of the attribute macros).

After those implicitly available parameters there come standard parameters explicitly supplied by user. They are the same as for expression level macros.

3.6 Reference to more advanced aspects

3.6.1 Hygiene and alpha-renaming of identifiers

Problem with names capture Identifiers in quoted code (object code) must be treated in a special way, because we usually do not know in which scope they would appear. Especially they should not mix with variables with the same names from the macro-use site.

Consider the following macro defining a local function `f`

```
macro identity (e) { <[ def f (x) { x }; f($e) ]> }
```

Calling it with `identity (f(1))` might generate confusing code like

```
def f (x) { x }; f (f (1))
```

To preserve names capture, all macro generated variables should be renamed to their unique counterparts, like in

```
def f_42 (x_43) { x_43 }; f_42 (f (1))
```

Hygiene of macros The idea of separating variables introduced by a macro from those defined in the plain code (or other macros) is called ‘hygiene’ after Lisp and Scheme languages. In Nemerle we define it as putting identifiers created during a single macro execution into a unique namespace. Variables from different namespaces cannot bind to each other.

In other words, a macro cannot create identifiers capturing any external variables or visible outside of its own generated code. This means, that there is no need to care about locally used names.

The Hygiene is obtained by encapsulating identifiers in special `Name` class. The compiler uses it to distinguish names from different macro executions and scopes (for details of implementation consult [paper about macros](#)). Variables with appropriate information are created automatically by quotation.

```
def definition = <[ def y = 4 ]>;
<[ def x = 5; $definition; x + y ]>
```

When a macro creates the above code, identifiers `y` and `x` are tagged with the same unique mark. Now they cannot be captured by any external variables (with a different mark). We operate on the `Name` class, when the quoted code is composed or decomposed and we use `<[$(x : name)]>` construct. Here `x` is bound to an object of type `Name`, which we can use in other place to create exactly the same identifier.

An identifier can be also created by calling method `Macros.NewSymbol()`, which returns `Name` with an unique identifier, tagged with a current mark.

```
def x = Macros.NewSymbol ();
<[ def $(x : name) = 5; $(x : name) + 4 ]>
```

Controlled breaking hygiene Sometimes it is useful to generate identifiers, which bind to variables visible in place where a macro is used. For example one of macro's parameters is a string with some identifiers inside. If we want to use these as real identifiers, then we need to break automatic hygiene. It is especially useful in embedding domain-specific languages, which reference symbols from the original program.

As an example consider a `Nemerle.IO.sprint (string literal)` macro (which have the syntax shortcut `$"some text $id "`). It searches given string literal for `$var` and creates a code concatenating text before and after `$var` to the value of `var.ToString ()`.

```
def x = 3;
System.Console.WriteLine ("My value of x is $x and I'm happy");
```

expands to

```
def x = 3;
System.Console.WriteLine ({
  def sb = System.Text.StringBuilder ("My value of x is ");
  sb.Append (x.ToString ());
  sb.Append (" and I'm happy");
  sb.ToString ()
});
```

Breaking of hygiene is necessary here, because we generate code (reference to `x`), which need to have the same context as variables from invocation place of macro.

To make given name bind to the symbols from macro usesite, we use `Nemerle.Macros.UseSiteSymbol (name : string) : Name` function, or special splicing target `usesite` in quotations. Their use would be like in this simplified implementation of macro

```
macro sprint (lit : string)
{
  def (prefix, symbol, suffix) = Helper.ExtractDollars (lit);
  def varname = Nemerle.Macros.UseSiteSymbol (symbol);
  <[
    def sb = System.Text.StringBuilder ($(prefix : string));
    sb.Append ($(varname : name).ToString ());
    // or alternatively $(symbol : usesite)
    sb.Append ($(suffix : string));
    sb.ToString ()
  ]>
}
```

Note that this operations is 'safe', that is it changes context of variable to the place where macro invocation was created (see paper for more details).

Unhygienic variables Sometimes it is useful to completely break hygiene, where programmer only want to experiment with new ideas. From our experience, it is often hard to reason about correct contexts for variables, especially when writing class level macros. In this case it is useful to be able to easily break hygiene.

Nemerle provides it with `<[$("id" : dyn)]>` construct. It makes produced variable break hygiene rules and always bind to the nearest definition with the same name.

4 Partial evaluation

4.1 Partial evaluation through meta-programming

We will discuss one of the possible applications of Nemerle meta-programming system in [partial evaluation](#). This process is based on specializing given algorithm with some of its inputs, which are known statically (at compile time).

This is particularly useful when we have some general algorithm for solving a problem, but most of the time we use its special case. For example we have a procedure *search* to search for arbitrary pattern in a string, but 98% of its uses are with "what is The Answer" pattern. It would be nice to have the *search_answer* procedure, which would be highly optimized version of our general algorithm.

The methodology we will use here is general and can be used for various problems like: specializing numerical algorithms, pattern matching, generating compilers from interpreters, efficient embedding of domain-specific languages.

4.1.1 Power function - classic example

Let us consider a general [power function](#). One of the most efficient ways of computing it is the logarithmic [power algorithm](#).

Its implementation in Nemerle is presented below:

```
static power (x : double, n : int) : double
{
  if (n == 0)
    1.0
  else
    if (n % 2 == 0) // even
      Sqr (power (x, n / 2))
    else
      x * power (x, n - 1)
}
```

where *Sqr* is a standard method defined like:

```
static Sqr (x : double) : double { x * x }
```

As we can see it divides *n* by 2 if it is even and decrements by 1 if it is odd. In former case it performs square operation and in latter multiplication by *x* parameter. Note that the pattern of those operation depends only on *n - x* and *n* parameters are independent.

Here we come to the point - when we want to have specialized power function for some *n*, then we know exactly what operations it will perform on second argument. For example optimal pattern for $x^{i \supset 5j / \supset j}$ is $x^{i \supset 2j / \supset j} i \supset 2j / \supset j$. Now we want compiler to be able to generate this optimal set of operations for us. Here is the code to perform it:

```
macro power1 (x, n : int)
{
  def pow (n) {
    if (n == 0)
      <[ 1.0 ]>
    else
      if (n % 2 == 0) // even
        <[ Sqr ($(pow (n / 2))) ]>
      else
```

```

    <[ $x * $(pow (n - 1)) ]>
  }
  pow (n);
}

```

We will give two different, but in general equivalent descriptions of what is happening here.

First you can view this as **staged computation**. We defer performing of *Sqr* and *** operations until *x* is known at runtime, but we are free to perform all the others like comparisons of *n* and recursion at earlier stage. In the result only the optimal pattern of multiplication and square operations will be executed at runtime. The *power* macro is just a function, which performs first stage of computation (at compile time). The result of using this macro is inserting the second stage operations at the place of usage. This way the `<[]>` brackets can be considered as markers of second stage computation.

The other way of understanding the example is by its implementation in Nemerle compiler. It is a function generating code, which will be substituted at the place of its usage. The *pow* function simply generates the code of arithmetic expression composed of multiplications and square operations. As mentioned earlier we utilize the fact, that *n* is independent from *x* and basing on it we know exactly how resulting expression should look like. Here we can view `<[]>` brackets as what they really are - syntax for constructing parts of new Nemerle code.

The macro above can be easily used to create specialized method

```

static power74 (x : double) : double
{
  power1 (x, 74);
}

```

or directly in code (which would yield direct inlining of expressions and might be not a good idea for large chunks of generated code).

Just a little digression. The power macro could be written as:

```

macro power2 (x, n : int)
{
  if (n == 0)
    <[ 1.0 ]>
  else
    if (n % 2 == 0) // even
      <[ Sqr (power2 ($x, $(n / 2 : int))) ]>
    else
      <[ $x * power2 ($x, $(n - 1 : int)) ]>
}

```

The resulting generated code would be identical, but here we see usage of some different features of macro system. In single macro execution we generate code, which contains invocation of this very macro. Note that it is not a direct recursive call, like in previous implementation. Although the result is the same, we encourage the former style, as it encapsulates all the macro's computations in single macro expansion, while the second way forces compiler to interleave many macro expansions. The latter way is slower and infinite loops here are harder to notice / debug.

4.1.2 Permutation algorithm

Now we will show a little more evolved example, which shows that sometimes it is useful to explicitly store parts of code in collection, creating final computation from it.

Let us consider a function for computing **permutation** of given array. It takes input array as first parameter and array specifying the permutation as second parameter:

```
permute (data : array [int], permutation : array [int]) : void { ... }
```

where *permutation* array describes permutation as positions at which elements of input should appear in output (like for **3, 2, 1** and permutation **2, 1, 3**, the output will be **2, 3, 1**). The algorithm utilizes the fact that this representation directly exposes cycles of permutation and to permute elements we must simply move them by one position on its cycle. It is presented below:

```
permute (data : array [int], permutation : array [int]) : void
{
  def visited = array (permutation.Length);

  // we visit each cycle once using this top loop
  for (mutable i = 0; i < permutation.Length; i++)
  {
    mutable pos = i;
    // we walk through one cycle
    while (!visited [pos])
    {
      visited [pos] = true;
      // moving its elements by one position
      def next_pos = permutation [pos];
      unless (visited [next_pos]) {
        data [pos] <-> data [next_pos];
        pos = next_pos;
      }
    }
  }
}
```

As we can see this algorithm does some operations of *data* only in one line

```
data [pos] <-> data [next_pos]
```

which is the swap operation on elements of array. The rest of its steps are performed only basing on contents of *permutation*. This quickly leads us to conclusion, that if we statically know this array, we can have highly optimized version of *permute*, which performs only sequence of swaps.

This is exactly what partial evaluation does, removing overhead of computations dependant only on static parameters. How do we code it using macros in Nemerle? It is almost as simple as deferring \leftrightarrow operation with $\langle [] \rangle$, but two more technical issues must be addressed.

First we must obtain the value of permutation array inside our first stage function (a macro). The simplest way would be to store it in some static field visible to the macro, but we chose to pass it directly as its parameter. Second one is that original *permute* function uses imperative style, so we must defer computation also in this style, explicitly building sequence of final computations.

```
macro permute1 (data, p_expr)
{
  def permutation = expr_to_array (p_expr); // new

  def visited = array (permutation.Length);
  mutable result = []; // new

  for (mutable i = 0; i < permutation.Length; i++) {
    mutable pos = i;
    while (!visited [pos]) {
```

```

    visited [pos] = true;
    def next_pos = permutation [pos];
    unless (visited [next_pos]) {
      result = <[
        $data [$(pos : int)] <-> $data [$(next_pos : int)]
      ]> :: result;          // new
      pos = next_pos;
    }
  }
}
<[ {..$result } ]>      // new
}

// technical function used to decompose expression
// holding constant array of ints
expr_to_array (expr : PExpr) : array [int]
{
  // we must convert syntax tree of array into array itself
  | <[ array [..$p_list] ]> =>
    def permutation = array (p_list.Length);
    mutable count = 0;
    foreach (<[ $(x : int) ]> in p_list) {
      permutation [count] = x;
      count++;
    }
    permutation

  | _ => throw System.ArgumentException ("only constant arrays are allowed")
}

```

As we can see the function didn't change much. We must have added the variable *result*, which is the list of resulting expressions to execute. It is used at the end in `<[{.. $result }]>` expression - the sequence of swaps on *data* is the result of macro.

permute and *permute'* can be used as follows:

```

permute_specialized (data : array [int]) : void
{
  permute1 (data, array [10, 7, 11, 0, 12, 5, 14, 6, 9, 4, 13, 2, 1, 8, 3]);
}

public Run (data : array [int]) : void {
  def perm = array [10, 7, 11, 0, 12, 5, 14, 6, 9, 4, 13, 2, 1, 8, 3];
  permute (arr, perm);
  permute_specialized (arr);
}

```

4.2 Rewriting interpreter into compiler

4.2.1 Virtual machine

```

public class Robot
{
  public mutable Orientation : byte;
}

```

```

public mutable X : int;
public mutable Y : int;

public IsDown : bool
{
  get { Orientation == 1 }
}

public override ToString () : string
{
  $"($X, $Y)"
}
}

```

4.2.2 Language

```

public variant Expr {
  | MoveBy { steps : int; }
  | Left
  | Right
  | Value { prop : string; }
  | If { cond : Expr.Value; then : Expr; els : Expr; }
}

```

4.2.3 Interpreter

```

public module Scripts
{
  public Run (obj : Robot, expr : Expr) : void
  {
    def check_value (val) {
      System.Convert.ToBoolean (obj.GetType ().GetProperty (val.prop).GetValue (obj, null))
    }

    match (expr) {
      | Expr.MoveBy (steps) =>
        match (obj.Orientation) {
          | 0 => obj.X += steps
          | 1 => obj.Y += steps
          | 2 => obj.X -= steps
          | _ => obj.Y -= steps
        }

      | Expr.Left => obj.Orientation = ((obj.Orientation + 3) % 4) :> byte;
      | Expr.Right => obj.Orientation = ((obj.Orientation + 1) % 4) :> byte;

      | Expr.Value as val => _ = check_value (val)

      | Expr.If (val, e1, e2) =>
        if (check_value (val))
          Run (obj, e1)
        else
          Run (obj, e2)
    }
  }
}

```

```

}

public Run (obj : Robot, name : string) : void
{
  def script = GetScript (name);
  foreach (e in script) Run (obj, e)
}
}

```

4.2.4 Staged interpreter

```

public module Scripts
{
  public GenerateRun (obj : PExpr, expr : Expr) : PExpr
  {
    def check_value (val) {
      <[ $obj.$(val.prop : dyn) ]>
    }

    match (expr) {
      | Expr.MoveBy (steps) =>
        <[ match ($obj.Orientation) {
          | 0 => $obj.X += $(steps : int)
          | 1 => $obj.Y += $(steps : int)
          | 2 => $obj.X -= $(steps : int)
          | _ => $obj.Y -= $(steps : int)
        } ]>

      | Expr.Left => <[ $obj.Orientation = (($obj.Orientation + 3) % 4) :> byte ]>;
      | Expr.Right => <[ $obj.Orientation = (($obj.Orientation + 1) % 4) :> byte ]>;
      | Expr.Value as val => <[ _ = $(check_value (val)) ]>

      | Expr.If (val, e1, e2) =>
        <[ if ($(check_value (val)))
          $(GenerateRun (obj, e1))
        else
          $(GenerateRun (obj, e2))
        ]>
    }
  }
}

macro GenerateRun (obj, name : string)
{
  def script = Scripts.GetScript (name);
  def exprs = List.Map (script, fun (e) { Scripts.GenerateRun (obj, e) });
  <[ { ..$exprs } ]>
}

```

4.2.5 Usage

```

def x = Robot ();
Scripts.Run (x, "myscript1");

```

```
System.Console.WriteLine (x);
GenerateRun (x, "myscript1");
System.Console.WriteLine (x);
```

5 Design patterns

5.1 Intro

Design patterns are meant to ease programmers' work by distinguishing various common schemas of creating software and solving most often occurring problems. Most of them involve some particular interaction between objects in program, which can be easily identified and named.

The downside of many design patterns is that they often require implementing massive amount of code, which tends always to be the same or almost the same. Some patterns just point out how particular objects could communicate or what should be the inheritance hierarchy of classes representing given model. These patterns are just hints for programmers what is the preferred way of structuring their programs in choosen situations. Unfortunately others often imply large dumb work for programmers trying to follow them. They need to override n -th method from the k -th class by adding the same code as in all $0..n-1$ previous methods. This kind of job is not only boring, but also is a waste of developer's time and will probably be a nightmare to maintain.

There comes a solution from metaprogramming. We could just write a generator for all those methods, wrappers, auxiliary instances, etc. In this document we present examples of choosen design patterns, where often repeating code may be generated instead of hand written.

5.2 Proxy design pattern

Proxy pattern is based on forwarding calls to some object A into calls to another object B . Usually the object B is contained in an instance field of A . The point of this is to imitate behavior of B in a new class. We can even implement B 's interface in A by simply passing all calls to B . We can then override some of these methods with new behaviour.

The solution presented is also very similar to **implicit interface implementation through aggregation**, one of many niched suggestions about C# language.

5.2.1 What we want to achieve

Suppose we have an interface `IMath` and an object `Math` implementing this interface. `Math` will be the object stored on server and we will create a representative object that controls access to it in a different *AppDomain*.

```
using System;
using System.Runtime.Remoting;

// "Subject"

public interface IMath
{
    // Methods
    Add( x : double, y : double ) : double;
    Sub( x : double, y : double ) : double;
    Mul( x : double, y : double ) : double;
    Div( x : double, y : double ) : double;
}

// "RealSubject"
```

```

class Math : MarshalByRefObject, IMath
{
    // Methods
    public Add(x : double, y : double ) : double { x + y; }
    public Sub(x : double, y : double ) : double { x - y; }
    public Mul(x : double, y : double ) : double { x * y; }
    public Div(x : double, y : double ) : double { x / y; }
}

```

Now for accessing `Math` in different `AppDomain` we create a `MathProxy`, which will provide full functionality of `IMath`. Internally it will use `Math` instance to implement all that functionality.

The point is that forwarding every call in `IMath` requires a considerable amount of code to be written. We will use a macro, which generates needed methods automatically.

```

// Remote "Proxy Object"
class MathProxy : IMath
{
    // the stubs implementing IMath by calling math.* are automatically generated
    [DesignPatterns.Proxy (IMath)]
    math : Math;

    // Constructors
    public this()
    {
        // Create Math instance in a different AppDomain
        def ad = System.AppDomain.CreateDomain( "MathDomain",null, null );
        def o =
            ad.CreateInstance("Proxy_RealWorld", "Math", false,
                System.Reflection.BindingFlags.CreateInstance, null,
                null, null,null,null );
        math = ( o.Unwrap() :> Math);
    }
}

```

As the comment says, for each method in `IMath` interface the implementation will be automatically created in a way similar to

```

public double Add( x : double, y : double ) {
    math.Add(x,y);
}

```

so we can then use `MathProxy` as

```

public class ProxyApp
{
    public static Main() : void
    {
        // Create math proxy
        def p = MathProxy();

        // Do the math
        Console.WriteLine( "4 + 2 = {0}", p.Add( 4.0, 2.0 ) );
        Console.WriteLine( "4 - 2 = {0}", p.Sub( 4.0, 2.0 ) );
        Console.WriteLine( "4 * 2 = {0}", p.Mul( 4.0, 2.0 ) );
    }
}

```

```

    Console.WriteLine( "4 / 2 = {0}", p.Div( 4.0, 2.0 ) );
  }
}

```

5.2.2 How we do this with a macro

In code above we used an attribute `[DesignPatterns.Proxy (IMath)]`. Actually it is a macro, which is invoked by compiler and will do what we need.

The implementation is presented below. It uses some of the compiler's API, but we will briefly describe what happens here.

```

using Nemerle.Compiler;
using Nemerle.Collections;

namespace DesignPatterns
{
  [Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers,
                      Nemerle.MacroTargets.Field)]
  macro Proxy (t : TypeBuilder, f : FieldBuilder, iface)
  {
    // find out the real type specified as [iface] parameter
    def interfc = match (Nemerle.Macros.ImplicitCTX().BindType (iface))
    {
      | MType.Class (typeinfo, _) when typeinfo.IsInterface => typeinfo
      | _ => Message.FatalError ("expected interface type")
    }
    foreach (meth :> IMethod in interfc.GetMembers ())
    {
      // prepare interface method invocation arguments
      def parms = List.Map (meth.GetParameters (), fun (p) {
        <[ $(t.ParsedName.NewName (p.name) : name) : $(p.ty : typed) ]>
      });
      // prepare function parameters of created method
      def fparms = List.Map (parms, Parsetree.Fun_parm);

      // define the wrapper method
      t.Define (<[ decl:
        public virtual $(meth.Name : dyn) (..$fparms) : $(meth.ReturnType : typed) {
          this.$(f.Name : dyn).$(meth.Name : dyn) (..$parms)
        }
      ]>)
    }
  }
}

```

Our macro takes as a parameter the name of interface, for which we need to create methods. This is just a plain expression, so we need to ask compiler what it really is and if it really describes an interface. We get an instance of `Nemerle.Compiler.TypeInfo` this way.

Then we iterate over members of this interface type, casting them to `IMethod` interface. From this interface we can obtain all information needed about the method. Its name and parameters.

The process of generating new method's declaration is a little bit complex. We must separately create expressions describing its parameters and expressions for method invocation. Finally we create code for calling method on the field for which our macro was used.

5.2.3 Macro included in standard library

Since version 0.9.2 of Nemerle, the slightly modified version of Proxy macro has been included into standard macros library. It can be accessed via *Nemerle.DesignPatterns.ProxyPublicMembers*. It creates wrappers for public members like methods and properties from given field to current type.

So if you have a class

```
class Foo ['a] {
  public Length : int {
    get { .. }
  }
  public Fire (x : int) : void { }
  public Gene (x : 'a) : 'a { x }
}
```

you can automatically duplicate its methods in your class like

```
[Record]
class Bar {
  [Nemerle.DesignPatterns.ProxyPublicMembers ()]
  my_foo : Foo [int];
}

def bar = Bar (Foo ());
_ = bar.Length;
bar.Fire (1);
_ = bar.Gene (1);
```

5.3 Singleton design pattern

Singleton Design Pattern ensures a class has only one instance and provide a global point of access to it. It lets you encapsulate and control the creation process by making sure that certain prerequisites are fulfilled or by creating the object lazily on demand.

5.3.1 What do we want to achieve?

We want to have a class, which will be automatically created when requested and then the created instance will be stored for any further requests. We just want to specify the name of property by which we will access class' instance and write the needed logic. Management of lazy creation and storage should be hidden, because it does not introduce any valuable information.

```
using System;
using Nemerle.Collections;
using System.Threading;

// "Singleton"

[DesignPatterns.Singleton (GetLoadBalancer)]
class LoadBalancer
{
  private servers : Vector [string] = Vector (5);
  private random : Random = Random ();

  // Constructors (protected)
```

```

protected this ()
{
    // List of available servers
    servers.Add( "ServerI" );
    servers.Add( "ServerII" );
    servers.Add( "ServerIII" );
    servers.Add( "ServerIV" );
    servers.Add( "ServerV" );
}

// Properties
public Server : string
{
    get
    {
        // Simple, but effective random load balancer
        def r = random.Next (servers.Count);
        servers [r];
    }
}
}

```

[DesignPatterns.Singleton (GetLoadBalancer)] attribute specifies that given class will be a singleton and that its instance will be accessed through property *GetLoadBalancer*.

```

public class SingletonApp
{
    public static Main() : void
    {
        def b1 = LoadBalancer.GetLoadBalancer;
        def b2 = LoadBalancer.GetLoadBalancer;
        def b3 = LoadBalancer.GetLoadBalancer;
        def b4 = LoadBalancer.GetLoadBalancer;

        // Same instance?
        when ((b1 : object == b2) && (b2 : object == b3) && (b3 : object == b4))
            Console.WriteLine( "Same instance" );

        // Do the load balancing
        Console.WriteLine( b1.Server );
        Console.WriteLine( b2.Server );
        Console.WriteLine( b3.Server );
        Console.WriteLine( b4.Server );
    }
}

```

5.3.2 How do we implement it in macro?

The task of a macro will be to

- create a field for storing the only instance
- create the property for accessing it, which will lazily call the constructor
- make sure that there is only one constructor and make sure it is protected

```

using Nemerle.Compiler;
using Nemerle.Collections;

namespace DesignPatterns
{
  [Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance,
                      Nemerle.MacroTargets.Class)]
  macro Singleton (t : TypeBuilder, getter)
  {
    def mems = t.GetParsedMembers ();
    // find constructor, which we will need to call
    // to create instance
    def ctor = List.Filter (mems, fun (x) {
      | <[ decl: ..$_ this (..$_) $_ ]> => true
      | _ => false
    });
    match (ctor) {
      | [ <[ decl: ..$_ this (..$parms) $_ ]> as constructor ] =>
        match (getter) {
          | <[ $(getter_name : name) ]> =>
            // we must prepare expressions for invoking constructor
            def invoke_parms = List.Map (parms, fun (x) {
              <[ $(x.ParsedName : name) ]>
            });

            // first define the field, where a single instance will be stored
            t.Define (<[ decl:
              private static mutable instance : $(t.ParsedName : name);
            ]>);

            // finally, define getter
            t.Define (<[ decl:
              public static $(getter_name : name) : $(t.ParsedName : name) {
                get {
                  // lazy initialization in generated code
                  when (instance == null)
                    instance = $(t.ParsedName : name) (..$invoke_parms);
                  instance;
                }
              }
            ]>);

            // make sure constructor is protected
            constructor.Attributes |= NemerleAttributes.Protected;

            | _ =>
              Message.FatalError ($"Singleton must be supplied with a simple name for getter, got $getter")
          }
        }
      | _ => Message.Error ("Singleton design pattern requires exactly one constructor defined")
    }
  }
}

```

First we search through members of the class and find all constructors. We make sure there is only one, read its parameters. After building expressions for invoking constructor we define needed class members. Finally constructor is marked as *protected*.

6 Syntax extensions

Nemerle has builtin syntax extension capabilities. They are limited to some fixed elements of language grammar, but their ability to defer parsing of entire fragments of token stream makes them quite powerful and usable.

Note: Remember that in order **to compile** anything that has a macro in it, you must use `ncc -r Nemerle.Compiler.dll foo.nj/codej` or similar.

6.1 Expression level extensions

We have mainly focused on the ability of extending the basic syntactic entity in Nemerle, which is the expression.

The new rules of parsing are triggered by the set of user definable distinguished keywords and operators. When the parser encounters one of those at the valid position of expression beginning, then it executes a special parsing procedure for syntax extension related to the distinguished token.

All syntax extensions are specified by **macro** definitions. Each macro can optionally have syntax definition, which describes how given macro should be called when the syntax occurs in program. For example

```
macro while_macro (cond, body)
syntax ("while", "(", cond, ")", body) {
  <[
    def loop () {
      when ($cond) {
        $body;
        loop ()
      }
    }
    loop ()
  ]>
}
```

creates a macro introducing *while* loop construct and syntax to the language.

6.2 Raw token extensions

Nemerle has a very powerful method for introducing virtually arbitrary syntax into language. It allows for specifying that given part of program input will not be interpreted by main Nemerle parser, but will be passed in easy to use representation to some macro.

6.2.1 The *parentheses tree* concept

The parser and all syntax extensions operate on token streams created by the lexer and the so-called pre-parse stage. The lexing phase simply transforms program text into a stream of tokens (like identifiers, numbers, operators, distinguished keywords, etc.).

The next phase groups this stream into a tree of parentheses. We have distinguished four types of them (`{}` `()` `[]` `<[]>`). Tokens inside those parentheses are also divided into groups separated by special separator tokens. For example the following program fragment

```
fun f (x : string) {
  def y = System.Int32.Parse (x);
  y + 1
}
```

is after the pre-parse stage represented as the token tree

```
[ fun , f , ( [ x , ':' , string ] ) , {
  [ def , y , '=' , System , '.' , Int32 , '.' , Parse , ( [ x ] ) ] ,
  [ y , '+' , 1 ]
} ]
```

where matched parentheses groups are distinguished with () {} [] and their elements are separated with ,. Note that groups like () and {} contain tokens enclosed by [], which represents loose token groups - divisions of tokens split by separators (, for () [] and ; for {} <[]>).

6.2.2 Parentheses tokens

So, according to the description above we have following kinds of special tokens, which represents whole fragments of unparsed code:

- Token.BracesGroup - for { }
- Token.RoundGroup - for ()
- Token.SquareGroup - for []
- Token.QuoteGroup - for <[]>, used in macro code quotation
- Token.LooseGroup - list of tokens grouped inside one of above brackets and separated by separator token specific for each of bracket kinds

All the available tokens produced by lexer can be viewed [here](#)

6.2.3 Passing token groups to the macro

Those raw grouping tokens can be passed as a parameter of macro. We simply have to name it when specifying macro parameter:

```
macro BuildXml (group : Token)
syntax ("xml", group)
{
  ...
}
```

in code, where such a macro was imported we can use the new syntax:

```
foo () : void {
  def doc = xml (<node name="foo">My name is foo</node>);
  // macro produced some XmlNode for us, we can use it
  print (doc.InnerXml);
}
```

Inside such macro we can use our own specialized parser. For example some small domain specific language can be embedded easily inside Nemerle program provided a simple syntax extension.

7 Macro tips

Some quick notes about using macros and quotations. This should really be merged with [macros' documentation page](#), but it will wait, because our parsetree structure is going to be changed soon and this will also impact quotations (simplify them) a lot.

7.1 Quotations

For best reference of how to write quotations, take a look at algorithm used to translate them in [the sources](#). This code is quite self explaining (at least if you just need to know how to write quotations).

7.1.1 Match cases

So first, why can't I write

```
<[ | Some (x) => x ]>
```

Unfortunately

```
| Some (x) => x
```

is a match case, not an expression, and standard quotation is used by default for expressions. To make a quoted match case, you must simply add the `case:` target to the quotation. So it would finally look like

```
<[ case: | Some (x) => x ]>
```

Unfortunately `parsetree` is not yet unified enough to look very consistent and `try` statement use different syntax for quotations. You write

```
<[ try $body catch { $exn : $exn_ty => $handler } ]>
```

This quotation allows only one handler in catch, but you can nest others in body of try block.

7.2 Compiler API available from macros

Macros are arbitrary functions and they can reference any external classes. It is sometimes also useful to use Nemerle compiler API from within a macro. It is usually done using two methods

- Using static helper functions from *Nemerle.Compiler* namespace
- Using the instance of *typer* to make more advanced things, like typing some code fragment, asking for defined local variables in current scope, etc.

The second operation requires obtaining an instance of *Nemerle.Compiler.Typer* within a macro. Actually, every macro has it as a hidden parameter (we just didn't want to pollute macros syntax with this parameter, because it is rarely used) and it can be obtained using *Nemerle.Macros.ImplicitCTX ()* function (it is a macro returning the hidden parameter).

Consider following code:

```
macro print_visible_vars () {
  def locals = Nemerle.Macros.ImplicitCTX ().GetLocals ();
  def amount = locals.Fold (0, fun (n : Name, loc : LocalValue, acc) {
    System.Console.WriteLine ($"var ${n.Id} is visible");
    acc + 1
  })
  System.Console.WriteLine ($"seen $amount variables");
}
```

```
}
```

This is the way how you can get some compiler's internals for your own usage. Feel free to ask for new useful methods to be created in compiler if you need them.

8 Defining types from inside macros

8.1 Basics

Macros can define new types, as well as add members to existing types.

There are two ways of defining types, you can either:

- define a nested type inside some other type (using `DefineNestedType` method of the `TypeBuilder`) or
- define a new top level type (using `Define` method of the `GlobalEnv` class).

Both methods take the same argument – just a single quotation of the type `decl`. `TypeBuilder` object can be obtained from a parameter of a macro-on-declaration or from `Nemerle.Macros.ImplicitCTX ().CurrentTypeBuilder`. Current `GlobalEnv` is available in `Nemerle.Macros.ImplicitCTX ().Env`.

```
macro BuildClass ()
{
  def ctx = Nemerle.Macros.ImplicitCTX ();
  def builder = ctx.Env.Define (<[ decl:
    internal class FooBar
    {
      public static SomeMethod () : void
      {
        System.Console.WriteLine ("Hello world");
      }
    }
  ]>);

  builder.Compile ();

  <[ FooBar.SomeMethod () ]>
}
```

This macro will add a new class inside the current namespace, the class will be named `FooBar`. The macro will return a call to a function inside this class. That is this code:

```
module Some {
  Main () : void
  {
    BuildClass ();
  }
}
```

will print the famous message.

One gotcha here is that the following code:

```
module Some {
  Main () : void
```

```
{
  BuildClass ();
  BuildClass ();
}
```

will give redefinition error instead of a second message. Macros do not guarantee hygiene of global symbols.

Another gotcha is the `builder.Compile()` call. If you forget it, then the compiler will throw ICE when the macro is used.

8.2 Longer example

```
macro BuildClass ()
{
  def ctx = Nemerle.Macros.ImplicitCTX ();
  def builder = ctx.Env.Define (<[ decl:
    internal class FooBar
    {
      public static SomeMethod () : void
      {
        System.Console.WriteLine ("Hello world");
      }
    }
  ]>);

  builder.Define (<[ decl: foo : int; ]>);
  builder.Compile ();
  builder.CannotFinalize = true;

  builder.Define (<[ decl:
    public Foo : int
    {
      get { foo }
    }
  ]>);

  def nested_builder = builder.DefineNestedType (<[ decl:
    internal class SomeClass { }
  ]>);
  nested_builder.Compile ();

  builder.CannotFinalize = false;

  <[ FooBar () ]>
}
```

The `GlobalEnv.Define` (as well as `TypeBuilder.Compile`) return a fresh `TypeBuilder` object, that can be used to add new members using the `Define` and `DefineNestedType` methods. There is also a `DefineAndReturn` method, that works much like `Define`, but returns the added member (as option `[IMember]`).

As you can see, you can add new member to already built class, but this needs additional hassle with the `CannotFinalize` property.

8.3 Stages of TypeBuilder

To understand the `CannotFinalize` stuff properly we need to talk a bit about the internals of the compiler.

During compilation it first scans through the entire program to look for global definitions. Then there are several passes dealing with them. You can plug macros in most places of this process. Once the global iteration passes are done, the compiler proceeds with typing and code generation for each method in turn. Then the regular macros are called.

Now we want to add new types during typing. However the passes setting up various things in types have already been run.

Therefore the `Define/DefineNestedType` call a few functions right after the type is created. The most important are:

- setting the `CannotFinalize` property to true
- resolving type names used in definitions
- adding default constructor if no constructor was found
- running any macros attached to the type and definitions within it

Next the `Compile()` call does a few other things:

- set `CannotFinalize` property to false
- check implemented interfaces (that is, if you declared that you will implement some interface and failed to provide implementation for some methods, you will get an error here)
- add SRE declarations (this in particular means that the members of this type can be used during expression typing **only** after `Compile()` is called)
- queue compilation of methods inside the type (the compilation cannot occur just at the moment `Compile()` is called, because we support only one method compilation at a time, and `Compile()` can be called during compilation of some method)

Now about the `CannotFinalize` property. If it's true, the type won't be finalized, that is finished up. This can happen if it gets to regular typer queues, so if you want to add members after `Compile()`, better set it to true. But don't forget to set it to false before the end of the compilation, otherwise you'll get an ICE.