

# Meta-programming in Nemerle

Kamil Skalski, Michal Moskal, and Pawel Olszta

University of Wroclaw, Poland  
<http://www.nemerle.org/>

**Abstract.** We present the design of a meta-programming system embedded into Nemerle<sup>1</sup>, a new functional language for the .NET platform. The system enables compile-time operations – generation, transformation and automated analysis of programs by means of hygienic code quotation, syntax extensions, operating on the code like on any other datatype (e.g. listing, adding or changing members of class definition), performing partial typing of a program syntax tree (compiler internal typing procedures are executed by a macro code) and interoperability with the compilation process. All these operations can be fully parametrized with any external data (like a database, a file or a web page).

Our system is a convenient tool for Aspects Oriented Programming with the ability to operate on datatypes, traverse the program code and perform various algorithmic operations on its content.

## 1 Introduction

The idea of compile-time meta-programming has been studied for quite a long time. It was incorporated into several languages, like Lisp macros [3], Scheme hygienic macros [4], C preprocessor-based macros, C++ template system and finally Haskell Template Meta-programming [2]. They vary in their capabilities and ease of use, but generally imply computations during compilation of the program and generating code from some definitions.

During this process programs are treated as *object programs*, which are data supplied to *meta-programs*. They can be then arbitrarily transformed or analyzed and the final result is compiled just like a regular program. These operations may be repeated or take place in stages. In the latter case the generated programs can generate other programs and so on.

*Meta-language* is a language for programming such operations. It usually has its own syntax for describing various constructs of the object language. For example, in our system, `<[ 1 + f (2 * x) ]>` denotes the syntax tree of expression `1 + f (2 * x)`. This idea is called *quasi-quotation*. The prefix *quasi* comes from the possibility of inserting values of meta-language expressions into the quoted context – if `g(y)` is such an expression, we can write `<[ 1 + $(g (y)) ]>`, which describes a syntax tree, whose second part is replaced by the result of evaluation of `g(y)`.

---

<sup>1</sup> The Nemerle project is supported by Microsoft Research ROTOR2 grant.

### 1.1 Our contribution

While we introduce several new ideas, the most important thing about our approach is the unique combination of a powerful meta-programming system together with the language, that possesses industrial strength, object-oriented and imperative capabilities.

The C++ example proves that there is a need for meta-programming systems in the industry – even the quite baroque template system is widely used for compile-time computations. This paper is a study of possible introduction of meta-programming techniques into an industrial environment in a cleaner form. We are therefore focused on making our system easy to use for programmers both writing and using the macros.

Key features of our approach are:

- We develop a uniform, hygiene preserving and simple quasi-quotation system, which does not require learning of internal compiler data structures to generate and transform quite complicated object programs. It also provides an easy way to write variable argument constructs (like tuples or function calls with an arbitrary amount of parameters).
- Using macros is transparent from the user point of view – the meta-program and common function calls are indistinguishable, so the user can use the most complex macros prepared by others without even knowing the idea of meta-programming.
- Flexible definition of syntax extensions allows even more straightforward embedding of macros into the language without interfering with compiler internals.
- Our system can be used to transform or generate practically any fragment of a program, which, composed with .NET object-oriented structure, provides a powerful tool for software engineering methodologies like aspects-oriented programming.
- We allow macros to type fragments of code, which they operate on, during their execution. This allows to parameterize them not only with the syntax of provided expressions, but also with the entire context of the program and types of those expressions.
- The separate compilation of macro definitions provides the clean and manageable stage separation.

### 1.2 Characteristics of Nemerle meta-system

Our meta-system has both *program generation* and *analysis* capabilities [8]. It can easily traverse the abstract syntax tree of the object program and gather information about it as well as change it (often using gathered data).

The system is designed mainly for operating on object programs at compile-time. However, using features of .NET and its dynamic code loading abilities, it is also possible to execute macros during run-time.

The meta-language is *homogeneous*, which means that it is the same as object language. We can use common Nemerle functions within macros and the syntax of generated programs is no different than the one used to write macros.

The quasi-quotation provides a clear separation of the object program from the meta-language. We do this with a manual annotation to distinguish stages of execution in a well understood fashion. Thus it is semantically clear, which part of the code is generated and which is generating. The symbols from the object-code are alpha-renamed so that they do not interfere with the external code.

## 2 First examples

Suppose we want to add some new syntax to our language, like the **for** loop. We could embed it into the compiler, but it is a rather difficult and inelegant way – such addition is quite short and it should not involve much effort to complete. Here a macro can be used.

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

This code creates a special meta-function, which is executed at the compile-time in every place where its original call is placed. Its result is then inserted into the program. Always when something like

```
for (i = 0, i < n, i = i + 2, a[i] = i)
```

is written, the appropriate code is created according to the **for** macro and replaces the original call.

The macros may instruct the compiler to extend the language syntax – for example a macro for the **for** loop with a C-like syntax can be defined. Writing

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

would add a new rule to the parser, which allows using

```
for (i = 0; i < n; i = i + 2) a[i] = i
```

instead of the call mentioned above.

## 2.1 Compiling sublanguages from strings

Macros are very useful for the initial checking and processing of the code written as strings in a program. This relates to many simple languages, like `printf` formatting string, regular expressions or even SQL queries, which are often used directly inside the program.

Let us consider a common situation, when we want to parameterize an SQL query with some values from our program. Most database providers in .NET Framework allow us to write commands with parameters, but neither their syntax is checked during the compilation, nor the consistency of the SQL data types with the program is controlled.

With a well written macro, we could write

```
sql_loop (conn, "SELECT salary, LOWER (name) AS lname"
          " FROM employees"
          " WHERE salary > $min_salary")
printf ("%s : %d\n", lname, salary)
```

to obtain the syntax and type-checked SQL query and the following code

```
def cmd = SqlCommand ("SELECT salary, LOWER (name)"
                     " FROM employees"
                     " WHERE salary > @parm1", conn);

(cmd.Parameters.Add (SqlParameter ("@parm1", DbType.Int32)))
.Value = min_salary;
def r = cmd.ExecuteReader ();
while (r.Read ()) {
  def salary = r.GetInt32 (0);
  def lname = r.GetString (1);
  printf ("%s : %d\n", lname, salary)
}
```

In fact the `printf` function here is another macro, that checks correspondence of supplied parameters to formatting string at compile-time.

The above construct yields safer and more readable code compared to its .NET counterparts.

## 3 Variable amount of arguments

The quotation provides full freedom in constructing any kind of expression. For example, we can decompose a tuple of any size and print its elements.

```

macro PrintTuple (tup, size : int)
{
  def symbols = array (size);
  mutable pvars = [];
  mutable exps = [];

  for (mutable i = size - 1; i >= 0; i--) {
    symbols[i] = NewSymbol ();
    pvars = <[ $(symbols[i] : name) ]> :: pvars;
    exps = <[ WriteLine ($(symbols[i] : name)) ]> :: exps;
  };

  exps = <[ def (.. $pvars) = $tup ]> :: exps;
  <[ {.. $exps } ]>
}

```

Note that here we need a number describing the size of the tuple. We show later, how to obtain the type of the given expression within the macro. This, for example, allows to compute the size of the tuple described by the `tup` variable.

## 4 Pattern matching on programs

The quotation can be used to analyze the program structure as easily as generate it. Standard mechanism of the language, pattern matching, fits perfect for such a purpose.

As an example we show implementation of `<->` operator, which swaps values of two expressions – like in `x <-> arr[2]`. We consider a simple approach first:

```

macro @<-> (e1, e2) {
  <[ def tmp = $e1; $e1 = $e2; $e2 = tmp; ]>
}

```

It has however one drawback of computing both expressions twice. For example trying to swap two random values in an array with `a[rnd()] <-> a[rnd()]` will not yield expected results. We can solve this problem by precomputing parts of expressions to be swapped<sup>2</sup>:

---

<sup>2</sup> The [Hygienic] modifier is discussed later

```
[Hygienic]
cache (e : Expr) : Expr * Expr
{
  | <[ $obj.$mem ]> => (<[ def tmp = $obj ]>, <[ tmp.$mem ]>)
  | <[ $tab [$idx] ]> =>
    (<[ def (tmp1, tmp2) = ($tab, $idx) ]>, <[ tmp1 [tmp2] ]>)
  | _ => (<[ ( ) ]>, e)
}
```

This function returns a pair of expressions. The first one is used to cache values and the second to operate on built equivalent of original expression using those values. Now we can implement `<->` as follows:

```
macro @<-> (e1, e2) {
  def (cached1, safe1) = cache (e1);
  def (cached2, safe2) = cache (e2);
  <[
    $cached1;
    $cached2;
    def tmp = $safe1;
    $safe1 = $safe2;
    $safe2 = tmp;
  ]>
}
```

## 5 Macros operating on declarations

Macros can operate not only on expressions, patterns, types, but also on any part of language, like classes, interfaces, other type declarations, methods etc. The syntax for those operations is quite different. Again, we treat language constructs as data objects, which can be transformed, but this is not done entirely with quotations. We use a special API, designed basing on `System.Reflection`, which is used in .NET for dynamic generation of assemblies. Nemerle type system and operations we are performing on them are not fully compatible with the interface of `Reflection` (we cannot directly derive from its classes), but are quite similar.

With such a tool we can analyze, generate or change any declaration in the program. For example, dump a definition of each data structure to an XML file, create serialization methods or automatically generate fields or methods from an external description.

Such macros are not called like ordinary functions, but are added as attributes in front of the declaration, similarly as `C#` attributes.

```
[SerializeBinary ()]
public module Company {
  [ToXML ("Company.xml")]
```

```

    public class Employee {
        ...
    }
    [FromXML ("Product.xml"), Comparable ()]
    public class Product { }
}

```

### 5.1 Transforming types

Macros that operate on declarations change them in the imperative fashion. Their first parameter is always an object representing the given declaration.

```

macro ToXML (ty : TypeBuilder, file : string) {

```

We can easily list the data contained in the provided object, like fields or methods of a class and add a new method using their names.

```

    def fields = ty.GetFields (BindingFlags.Instance %|
                               BindingFlags.Public %|
                               BindingFlags.DeclaredOnly);
    def list_fields =
        List.Map (fields, fun (x) { <[
            xml_writer.WriteAttributeString
                (x.Name : string),
                (x.Name : usesite).ToString ()
        ]> }
    );
    ty.Define (<[ decl:
        public ToXML () : void {
            def xml_writer = XmlTextWriter ($(file : string), null);
            { ..$list_fields };
            xml_writer.Close ();
        }
    ]>);

```

With the macro above (perhaps modified to do some additional formatting) we can generate serialization methods for any class simply by adding the `[ToXML ("file.xml")]` attribute.

## 6 Aspects-oriented programming

AOP has been proposed as a technique for separating different *concerns* in software. It is often a problem, when they crosscut natural modularity of the rest of implementation and must be written together in code, leading to tangled code.

## 6.1 Join points and pointcuts

*Join points* are well-defined points in the execution of a program. In contrary to many other systems, which define fixed *join points* in the code (like AspectJ [10]), macros allow to place the user code in arbitrary parts of program. One can write a macro traversing all the classes and adding some behavior to their bodies.

A *pointcut* picks out join points. In Nemerle there are no hard-coded design restrictions placed on pointcuts and the user can create any selection function. Particularly we can write a program which enables the same join points and pointcuts as those well established in AOP community.

## 6.2 Advices

*Advice* is a code executed at each join point picked out by a pointcut. It is straightforward for a macro to add *advices* to the code by simply transforming and combining it from given subprograms.

Consider the following code, adapted from the AspectJ tutorial:

```
after(Object o) throwing (Error e): publicInterface(o) {
    log.Write (o, e);
}
```

Given some traversal strategy, one can detect places where exceptions of some type are thrown and add their logging.

```
match (e) {
    | <[ throw $e ]> when IsAppropriateType (e) =>
        <[ log.Write ($o, $e); throw $e ]>
    | _ => e
}
```

## 6.3 User interface to AOP features

In order to make the aspects oriented paradigm production-ready in Nemerle, we have to develop an easy user interface for it. In general programmers would not use entire power of macro transformations in every day development. It is better to adapt existing interfaces and progressively add new power of expressiveness to systems like AspectJ.

We think that it should be possible to implement most Aspects-oriented and Adaptive-programming system designs with more or less complex macros. This field is our future research direction and we will present more details after finishing the implementation of all the necessary features.

## 7 Details of our design

In this section a more formal definition of the macro and our meta-system is provided.

A macro is a top-level function prefixed with the **macro** keyword, which may have access modifiers like other methods (**public**, **private** etc.) and resides in `.NET/Nemerle` namespace. It is used within the code like any other method, but it is treated in a special way by the compiler.

Its formal parameter types are restricted to the set of Nemerle grammar elements (including simple literal types). Parameters of the macro are always passed as their syntax trees, which for some types are partially decomposed. For example, literal types appear within the macro as real values (`int`, `string`, etc.), but they are passed as syntax trees, so they must be given as constant literals (it is obvious since these values must be known at the compile-time).

### 7.1 Macro as a function

A macro cannot be called recursively or passed as a first-class citizen (although it can generate the code, which contains the calls of this macro). Still, it can use any function from the rest of the program in a standard ML fashion, so we consider this as a minor disadvantage. If complex computations on syntax trees are necessary, one must simply put them into some arbitrary functions and run the entire computation from within the macro. Such design allows to define easily which functions are run at the compile-time without requiring any special annotations at their use site.

### 7.2 Names binding in quotation

A very important property of the meta-system is called *hygiene* and relates to the problem with names capture in Lisp macros, resolved later in Scheme. It specifies that variables introduced by a macro may not bind to variables used in the code passed to this macro. Particularly variables with the same names, but coming from different contexts, should be automatically distinguished and renamed.

Consider the following example:

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

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

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

To prevent 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))
```

In general, names in the generated code bind to definitions visible within their scope. The binding is done after all transformations during the execution of the macro are finished. This means that a variable used in a quotation may not necessarily refer to the definition visible directly in the place where it is written. Everything depends on where it occurs in the finally generated code. Consider the following example:

```
def d1 = <[ def x = y + foo (4) ]>;
def d2 = <[ def y = $(Bar.Compute() : int) ]>
<[ $d2;
  def foo (x) { x + 1 };
  $d1;
  x * 2   ]>
```

As macros might get large and complex it is frequently very useful to compute parts of the expression independently and then compose the final code from them. Still, names in such macro are alpha-renamed so that they do not capture any external definitions. The renaming is defined as putting names created in the single macro execution into the same “namespace”, which is mutually exclusive with all other “namespaces” and the top-level code. This is exactly the hygiene rule – neither the macro can capture names used in the macro-use place nor it can define anything colliding with the external code.

This is an opposite approach to Template Haskell [2], where the lexical scoping means binding variables from the object code immediately to definitions visible at the construction site of the quotation. We find our approach more flexible, as we can transform the code with much more freedom, while still keeping the system hygienic. This is of course just a design decision, naturally associated with certain costs. Sometimes it is not obvious to recreate bindings simply by looking at the code, but here we assume that a programmer of a macro knows the structure of the code to be generated. We also lose the ability to detect some errors earlier, but as they are always detected during the compilation of the generated code, we believe it is a minor disadvantage.

One can think that putting all identifiers from entire macro invocation into a single “namespace” is not a good idea, especially when we use some general purpose code generating function from a library, which should generate only its own unique names. To obtain such independent, hygienic functions we write

```
[Hygienic] f (x : Expr) : Expr { ... }
```

The `[Hygienic]` attribute is a simple macro, which transforms `f` to enable its own context during execution. This way function receives the same semantics as macro with regards to hygiene. We consider this behavior not good by default, because code is often generated by some tool functions, especially defined locally in macro and they should not change their context.

### 7.3 Breaking hygiene

Sometimes it is useful to share some names between several macro executions. It can be done safely by generating a unique identifier independent of macro executions. We support it by function `NewSymbol ()` whose return value can be stored in a variable, providing the hygiene preserving solution.

There are also situations, where we know the exact name of the variable used in the code passed to the macro. If we wanted to define a name referring to it, we would have to change the scope to our macro use site. As it breaks hygiene, it should be done in a controlled manner. Consider a macro introducing **using** keyword (C# keyword, simplified for the purpose of this paper):

```
macro using (name : string, val, body) {
  <[
    def $(name : usesite) = $val;
    try { $body } finally { $v.Dispose () }
  ]>
}
```

It should define a symbol binding to variables of the same name in `body`. But if it contained some other external code, like in:

```
macro bar(ext) {
  <[ using ("x", Foo (), { $ext; x.Compute () }) ]>
}
```

some inadvertent capture of variables in `ext` might happen, if `x` was just a plain dynamically scoped variable.

Although it is not recommended, also nonhygienic symbols can be created, by `$(x : dyn)`, where `x` is of type `string`. They are bound to the nearest definition with the same name appearing in the generated code, regardless of the context it comes from.

### 7.4 Lexical scoping of global symbols

The object code often refers to variables, types or constructors imported from other modules (e.g. .NET standard library or symbols defined in the namespace where the macro resides). In normal code we can omit the prefix of the full name, by including **using** keyword, which imports symbols from given namespace. Unfortunately this feature used in the object code like

```
using System.Text.RegularExpressions;
using Finder;

macro finddigit (x : string) {
  <[
    def numreg = Regex (@"\d+-\d+");
```

```

        def m = numreg.Match (current + x);
        m.Success ();
    ]>
}

public module Finder {
    public static current : string;
}

```

brings some dependency on currently imported namespaces. We would like the generated code to behave alike no matter where it is used, thus the `Regex` constructor and the `current` variable should be expanded to their full name – `System.Text.RegularExpressions.Regex` and `Finder.current`, respectively. This operation is automatically done by the quotation system. When a symbol is not defined locally (and with the same context as described in the previous section), its binding is looked for in global symbols imported at the quotation definition site.

Note that there is no possibility to override security permissions this way. Access rights from the lexical scope of the macro are not exported to the place where the generated code is used. .NET policies do not allow this, thus the programmer must not generate a code breaking the static security.

## 7.5 Accessing compiler internals

It is vital for meta-functions to be able to use all benefits they have from running at the compile-time. They can retrieve information from the compiler, use its methods to access, analyze and change data stored in its internal structures.

**Retrieving declarations** For example, we can ask the compiler to return the *type declaration* of a given *type*. It will be available as syntax tree, just like as we put a special macro attribute (Section 5) before that declaration. Certainly, such a declaration must not be an external type and has to be available within the compiled program as a source code.

```

def decl = Macros.GetType (<[ type: Person ]>);
xmlize (decl);    // we can use macros for declarations

```

## 8 Typing during execution of macro

Some more advanced techniques are also possible. They involve a closer interaction with the compiler, using its methods and data structures or even interweaving with internal compilation stages.

For example, we can ask the compiler to type some of object programs, which are passed to the macro, retrieve and compare their types, etc. This means that we can plug between many important actions performed by the compiler, adding

our own code there. It might be just a nicer bug reporting (especially for macros defining complex syntax extensions), making code generation dependent on input program types or improving code analysis with additional information.

### 8.1 Example

Let us consider the following translation of the `if` condition to the standard ML matching:

```
macro @if (cond, e1, e2)
syntax ("if", "(" , cond, ")", e1, "else", e1) {
  <[
    match ($cond) {
      | true => $e1
      | false => $e2
    }
  ]>
}
```

When `if ("bar") true else false` was written, the compiler would complain that type of matched expression is not `bool`. Such an error message could be very confusing, because the programmer may not know, that his `if` statement is being transformed to the `match` statement. Thus, we would like to check such errors during the execution of the macro, so we can generate a more verbose message.

### 8.2 Usage

Instead of directly passing object expressions to the result of the macro, we can first make the compiler type them and then find out if they have the proper type. The body of the above `if` macro should be

```
def tcond = TypedExpr (cond);
def te1 = TypedExpr (e1);
def te2 = TypedExpr (e2);
if (tcond.Type == <[ ttype: bool ]> ) {
  <[
    match ($(tcond : typed)) {
      | true => $(te1 : typed)
      | false => $(te2 : typed)
    }
  ]>
}
else
  FailWith ("‘if’ condition must have type bool, " +
           "while it has " + tcond.Type.ToString())
```

Note that typed expressions are used again in the quotation, but with a special splicing tag “typed”. This means that the compiler does not have to perform the typing (in fact, it cannot do this from now on) on the provided syntax trees. Such a notation introduces some kind of laziness in typing, which is guided directly by a programmer of the macro.

### 8.3 PrintTuple example

As mentioned in remarks to the `PrintTuple` function, a macro which is able to run typing of its parameters can obtain the size of a supplied tuple. We just need to add the following lines:

```
match (TypedExpr (tup).Type) {
  | <[ ttype: (..args) ]> =>
    def size = List.Length (args);
    ..
}
```

## 9 How it works

We will now describe how our meta-programming system works internally.

Each macro is translated to a separate class implementing a special *IMacro* interface. It provides a method to run the macro, which in most cases involves passing it the list of Nemerle grammar elements (untyped syntax trees of object programs).

Therefore at the compiler level a macro is a function operating on syntax trees. There are several kinds of syntax trees used in Nemerle compiler. We will focus on parse trees and typed trees.

Parse trees are generated by the parser as well as the quotations during macro execution. They are mostly one-to-one with the grammar of the language.

Typed trees contain less language construct (particularly no macro invocations). These constructs are however more explicit. In particular they contain inferred types of expressions.

The process of typing in compiler generally involves rewriting parse trees into typed trees.

The typing function, when it encounters a macro invocation, executes the *Run* method of respective macro object. The macro invocation looks like a regular function call, so we distinguish these two cases by looking up name of invoked function in list of currently loaded macros (the *IMacro* interface has a special *GetName* method).

To support typing parts of macro parameters, we have a special node in parse tree, that simply holds typed tree node. Typing function simply strips parse tree box, leaving content intact.

## 9.1 Quotations

The quotation system is just a shortcut for explicitly constructing syntax trees from compiled data types. For example `f(x)` expression is internally represented by `E_call (E_ref ("f"), [Parm (E_ref ("x"))])`, which is equivalent to `<[ f (x) ]>`. Translating the quotation involves “lifting” the syntax tree by one more level – we are given an expression representing a program (its syntax tree) and we must create a representation of such expression (a larger syntax tree). This implies building a syntax tree of the given syntax tree, like

```
E_call (E_ref ("f"), [Parm (E_ref ("x"))])
=>
E_call ("E_call",
  [Parm (E_call ("E_ref", [Parm (E_literal
    (L_string ("f"))]))]);
  Parm (E_call ("Cons", [Parm (E_call ("Parm",
    [Parm (E_call ("E_ref",
      [Parm (E_literal (L_string ("x"))]))]))]))]))])
```

or using the quotation

```
<[ f (x) ]>
=>
<[ E_call (E_ref ("f"), [Parm (E_ref ("x"))]) ]>
```

Now splicing means just “do not lift”, because we want to pass the value of the meta-language expression as the object code. Of course it is only valid when such an expression describes (is type of) the syntax tree. Operator `..` inside the quotation is translated as the syntax tree of list containing lifted expressions from the provided list (which must occur after `..`).

## 9.2 Making identifiers hygienic

In this section we describe how we achieve hygiene in our system. As said before, we use distinct “namespace” or “color” for each macro invocation. Our coloring system is quite simple. All plain identifiers introduced in quotation receive the color of the current macro invocation. The identifiers marked with `$(id : name)` receive the color of the code that called the macro. This can be the color of the top-level object code, as well as the color of some earlier quotation.

We describe our approach formally using a few simple inference rules.

The entire language is flattened to plain flat terms. We are only interested here in identifiers and macro invocations. Macro invocation are written as terms of the form `macro(name, parameter)`, while identifiers are denoted `id(v, c, g)` where `v` is the name of identifier, `c` is its color, and `g` is global environment for a given symbol.

A name is a term representation of strings written in the program text. A color is a term representation of integer, but it can take two special forms

`current()` and `usesite()`. The global environment is a list of namespaces opened with the `using` declaration in scope where given identifier was defined. We only consider terms, where neither name nor color nor environment contain `macro()` or `id()`.

After all macros are expanded and colors resolved we can say what each name binds to. Regular lexical scoping rules apply – some terms define symbols, some other use it. Use refers to the nearest preceding declaration with the same color. If there is no such declaration – the symbol is looked up in global environment enclosed in each symbol.

$\Gamma$  is a function describing dynamic semantics of compiled macros. It takes the name of the macro and its parameter as an input and returns the result of macro application on this parameter. If the macro needs more parameters they can be easily encoded using cons-like terms.

Identifiers introduced by macros in a hygienic way are denoted  $id(v, current(), g)$  while identifiers introduced with `UseSiteSymbol` are marked  $id(v, usesite(), g)$  in result of this function. The global environment  $g$  there comes from the context within which the macro was defined. Top-level object code is already colored with a single unique color.

$$\frac{e_1 \rightarrow e'_1 \dots e_n \rightarrow e_n}{\mathcal{F}(e_1, \dots, e_n) \rightarrow \mathcal{F}(e'_1, \dots, e'_n)} (Mon) \text{ where } \mathcal{F} \notin \{macro, id\}$$

$$\frac{}{id(v, x, g) \rightarrow id(v, x, g)} (MonId)$$

$$\frac{\Gamma(m, e) \Rightarrow_c^{(u, g')} e' \quad e' \rightarrow e''}{macro(id(m, u, g'), e) \rightarrow e''} (Expand) \text{ where } c \text{ is a fresh color}$$

$$\frac{e_1 \Rightarrow_c^{(u, g')} e'_1 \dots e_n \Rightarrow_c^{(u, g')} e_n}{\mathcal{F}(e_1, \dots, e_n) \Rightarrow_c^{(u, g')} \mathcal{F}(e'_1, \dots, e'_n)} (ColMon) \text{ where } \mathcal{F} \notin \{id\}$$

$$\frac{}{id(v, usesite(), g) \Rightarrow_c^{(u, g')} id(v, u, g')} (ColUse)$$

$$\frac{}{id(v, current(), g) \Rightarrow_c^{(u, g')} id(v, c, g)} (ColCur)$$

$$\frac{}{id(v, x, g) \Rightarrow_c^{(u, g')} id(v, x, g)} (ColSkip) \text{ where } x \notin \{current(), usesite()\}$$

**Definition 1.** We say that  $e$  is valid if it does not contain terms of the form  $id(v, current(), g)$  and  $id(v, usesite(), g)$ .

**Definition 2.** We say that  $e$  is in normal form if it is valid and does not contain macro in head of any term.

Normal form is thus an object code without macro invocation and with full color information.

**Theorem 1.** *For any valid  $e$ , there exists  $e'$  in normal form, such that it can be proven that  $e \rightarrow e'$ .*

*Proof.* Rules for both  $\rightarrow$  and  $\Rightarrow$  are syntax-directed and defined for all terms. Thus, for any  $e$  there exists  $e'$ , such that  $e \rightarrow e'$ . Moreover, usage of  $\rightarrow$  eliminates all occurrences of *macro()*, usage of  $\Rightarrow$  guarantees elimination of all *current()* and *usesite()* introduced by macros.  $\square$

### 9.3 Compiling and loading

A key element of our system is the execution of meta-programs during the compile-time. To do this they must have an executable form and be compiled before they are used.

Macros after compilation are stored in assemblies (compiled libraries of code). All macros defined within an assembly are listed in its metadata. Therefore, when linking an assembly during the compilation is requested by user, we can construct instances of all macro classes and register them by names within the compiler.

Each macro resides in a namespace. The name of the macro is prefixed with the name of the namespace. To use short name of macro it is necessary to issue the **using** declaration for the respective namespace. This works exactly the same as for regular functions. If a macro defines a syntax extension, it is activated only if **using** for respective namespace is in force.

This is an industrial strength design. It allows macro usage without namespace pollution, syntax extensions can be loaded on demand, and all macros can be used by programmers unaware of the very idea of meta-programming.

### 9.4 Separate Compilation

The current implementation requires macros to be compiled in a separate pass, before the compilation of the program that uses them. This results in inability to define and use a given macro in the same compilation unit. While we are still researching the field of generating and running macros during the same compilation our current approach also has some advantages.

The most important one is that it is simple and easy to understand – one needs first to compile the macros (probably being integrated into some library), and then load them into the compiler and finally use them. This way the stages of compilation are clearly separated in a well understood fashion – an important advantage in the industrial environment where meta-programming is a new and still somewhat obscure topic.

The main problem with ad-hoc macros (introduced and used during the same compilation) is that we need to first compile transitive closure of types (classes with methods) used by given macro. This very macro of course cannot be used in these types.

This issue may be hard to understand by programmers (“why doesn’t my program compile after I added new field to this class?!”). On the other hand,

such a dependency-closed set of types and macros can be easily split out of the main program into the library.

Experiences of the Scheme community show [5] how many problems arise with systems that do not provide clear separation of compilation stages. Indeed, to avoid them in large programs, manual annotations describing dependencies between macro libraries are introduced.

## 10 Related work

### 10.1 Scheme hygienic macros

Our system has much in common with modern Scheme macro expanders [1]:

- Alpha-renaming and binding of variables is done after macro expansion, using the context stored in the macro in use site
- Macros can introduce new binding constructs in a controlled way, without possibility to capture third party names.
- Call site of macros has no syntactic baggage, the only place where special syntax appears is the macro definition – this implies simple usage of macros by programmers not aware of meta-programming.

Still maintaining the above features we embedded the macro system into a statically typed language. The generated code is type-checked after expansion. We also provide a clear separation of stages – meta-function must be compiled and stored in a library before use.

Works on Scheme last quite long, and many interesting features have been proposed. For example first-class macros in [9] seem to be possible to implement in Nemerle by simply passing functions operating on object code.

### 10.2 Template Haskell

There are interesting differences between Template Haskell [2] and Nemerle macros:

- Resolving bindings during translating of quotations brings ability to reason about type-correctness of object code, before it is used. It allows detecting errors much earlier. Nevertheless, the presence of `$` splicing construct makes typing postponed to next stage of compilation, in which case new bindings must be dynamic.
- Template Haskell macros are completely higher-order, like any other function: they can be passed as arguments, partially applied, etc. This however requires manually annotating which code should be executed at compile-time. We decided to make macros callable simply by name (like in Scheme), so their usage looks like calling an ordinary function. We are still able to use higher-order functions in meta-language (functions operating on object code can be arbitrary), so only the top meta-function (prefixed with `macro` is triggering compile-time computations.

- We do not restrict declaration splicing to the top-level code and we also do not introduce a special syntax for macros introducing them. This seems a good way of taking advantage of binding names after macro expansion and imperative style present in Nemerle. It is natural for an imperative programmer to think about introduced definitions as about side effects of calling macros, even if these calls reside within quoted code.
- We introduce macros operating on type declarations, which are able to imperatively modify them. Moreover, they look like attributes attached to type definitions, so again programmer does not have to know anything about meta-programming to use them.

There are still many similarities to Template Haskell. We derive the idea of quasi-quotation and splicing directly from it. Also the idea of executing functions during compilation and later type-checking their results is inspired by Template Haskell.

### 10.3 C++ templates

C++ templates [11] are probably the most often used meta-programming system in existence. They offer Turing complete, compile time macro system. However, it is argued if the Turing-completeness was intentional, and expressing more complex programs entirely in the type system, not designed for this purpose is often quite cumbersome. Yet, the wide adoption of this feature shows need for meta-programming systems in the industry.

There are a few lessons we have learned from C++ example. First is to keep the system simple. Second is to require macro precompilation, not to end up with various problems with precompiled headers (a C++ compiler must-have feature, because of performance).

### 10.4 CamlP4

CamlP4 [12] is a preprocessor for OCaml. Its LL(1) parser is completely dynamic, thus allowing quite complex grammar extensions to be expressed. Macros (called syntax extensions) need to be compiled and loaded into the parser prior to be used. Once run, they can construct new expressions (using quotation system), but only at the untyped parse tree level. It is not possible to interact with compiler in more depth.

### 10.5 MacroML

MacroML [6], the proposal of compile-time macros on top of an ML language, has similar assumptions to Template Haskell by means of binding names in quotations before any expansion. It additionally enables pattern for introducing hygienic definition capturing macro use-site symbols (similar to our `UseSiteSymbol()`). All this is done without need to break typing of quotation before expansion.

Macros in MacroML are limited to constructing new code from given parts, so matching and decomposing of code is not possible.

## 10.6 MetaML

MetaML [7] inspired both Template Haskell and MacroML by introducing quasi-quotation and idea of typing object code. It was developed mainly to operate on code and execute it during runtime, so it represents a little different field of research than ours.

## 11 Further work

- Runtime program generation is still very wide research area for our meta-system. It would be useful to optimizations based on runtime only accessible data.
- We will focus on implementing interface to Aspects-Oriented programming features in Nemerle. It seems to be a good way of introducing meta-programming paradigm to commercial environment.
- Early typing of object code – to detect as many errors as possible during macro compilation (as opposite to macro execution) we would support special kind of typing function. Of course we cannot tell the type of the  $\$( )$ -splices in the object code (it is obviously undecidable). Additionally we are restricted by binding of identifiers made at post-expansion time, but still we can reject programs like `<[ 1 + (x : string) ]>`.

*Acknowledgments* We would like to thank Marcin Kowalczyk for very constructive discussion about hygienic systems, Lukasz Kaiser for useful opinions about quotation system and Ewa Dacko for corrections of this paper.

## References

1. Dybvig, R. K., Hieb, R., Bruggeman, C.: Syntactic Abstraction in Scheme. Lisp and Symbolic Computations, 1993
2. Sheard, T., Jones, S. P.: Template Meta-programming for Haskell. Haskell Workshop, Oct. 2002, Pittsburgh
3. Steele, Jr., Guy, L. Common Lisp, the Language. Digital Press, second edition (1990)
4. Clinger, William, Rees, Jonathan et al. The revised report on the algorithmic language Scheme. LISP Pointers, 4, 3 (1991)
5. Flatt, M.: Composable and Compilable Macros. ICFP, Oct. 2002, Pittsburgh
6. Ganz, S., Sabry, A., Taha, W.: Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. Preceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-2001), New York, Sep. 2001
7. Sheard, T., Benaissa, Z., Martel, M.: Introduction to multistage programming: Using MetaML.
8. Sheard, T.: Accomplishments and Research Challenges in Meta-Programming. 2001
9. Bawden, A.: First-class Macros Have Types. Symposium on Principles of Programming Languages, 2000
10. <http://eclipse.org/aspectj/>
11. Veldhuizen, T.: Using C++ template metaprograms. C++ Report Vol. 7 No. 4 (May 1995), pp. 36-43.
12. Rauglaudre, D.: Camlp4 – Reference Manual. <http://caml.inria.fr/camlp4/>