Posted October 17, 2010 by [Eric Niebler](http://ericniebler.com), under [Boost](http://cpp-next.com/archive/category/boost/)

From [http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/)

**Expressive C++:   
 A Lambda Library in 30 Lines (Part 2 of 2)**

This entry is part of a series, [Expressive C++»](javascript:;) Entries in this series:

1. [Expressive C++: Introduction](http://cpp-next.com/archive/2010/08/expressive-c-introduction/)
2. [Expressive C++: Playing with Syntax](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/)
3. [Expressive C++: Why Template Errors Suck and What You Can Do About It](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/)
4. [Expressive C++: A Lambda Library in 30 Lines (Part 1 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/)
5. **Expressive C++: A Lambda Library in 30 Lines (Part 2 of 2)**
6. [Expressive C++: Fun With Function Composition](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/)
7. [Expressive C++: Trouble With Tuples](http://cpp-next.com/archive/2010/11/expressive-c-trouble-with-tuples/)
8. [Expressive C++: Expression Optimization](http://cpp-next.com/archive/2011/01/expressive-c-expression-optimization/)

Welcome to the latest installment of Expressive C++, a series of articles devoted Embedded Domain-Specific Languages (EDSLs) and Boost.Proto, a library for implementing them in C++. In the [last installment](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/), we started developing a very simple library for creating inline, anonymous function objects: lambdas. In this installment we’ll finish the job. By the end of this article, you’ll know how make your EDSL come alive by giving expressions domain-specific behaviors. When we’re done, we’ll have a very tiny lambda library that is actually useful.

**Quick Recap**

Last week we wrote out the very beginnings of a lambda library. Since we’ll be referring back to it, I’ll reproduce the complete example here:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <cassert>  #include <boost/proto/proto.hpp>  using namespace boost;    struct arg1\_tag {};  proto::terminal<arg1\_tag>::type const arg1 = {};    // A Proto "algorithm": a grammar with embedded transforms for  // evaluating lambda expressions (explained in the previous article):  struct Lambda  : proto::or\_<  // When evaluating a placeholder terminal, return the state.  proto::when< proto::terminal<arg1\_tag>, proto::\_state >  // Otherwise, do the "default" thing.  , proto::otherwise< proto::\_default< Lambda > >  >  {};    int main()  {  // Evaluate the lambda arg1 + 42, replacing arg1 with 1  int i = Lambda()( arg1 + 42, 1 );  assert( i == 43 );  } |

***Data/Algorithm Separation***

*Proto encourages you to think of your data as separate from your algorithm, like the STL. In this case, “data” is an expression tree, and “algorithm” is the code that traverses the tree and does something. The algorithm is where you put your program logic; the data has no knowledge of how it will be evaluated. But in the case of lambda expression trees, you want the tree itself to “know” how to evaluate itself. With Proto, you can still develop data and algorithm separately, and then wire them together at the end. This article shows you how.*

With this simple program, we can evaluate arbitrarily complicated expression trees involving the arg1 placeholder. Not too bad for such a small program.

But as a lambda library, this solution leaves much to be desired. The biggest problem is that the expression arg1 + 42 is a dumb tree with no particular meaning; it just sits there until it is passed to the Lambda algorithm for evaluation. It’d be way better if we could evaluate the expression tree by passing arguments to the lambda directly, like (arg1 + 42)(1). That way, we could pass arg1 + 42 to algorithms like std::transform. We’ll see how in a minute; but first, I have to head off some potential confusion.

**Expressions: What’s In A Name?**

I’ve been carelessly throwing this word “expression” around a lot. “Expression” can mean many different things, and its important to keep it all straight, so forgive me while I make a brief digression. At the lowest level is syntactic conformance to the rules for valid C++ expressions. 1 << 8 is a C++ expression in the trivial sense that a C++ compiler can parse it. Here, << is just a binary operator with certain associativity and precedence.

Things get more interesting when we think about semantics. What does << *mean?* It takes the left operand and left-shifts it by the amount of the right operand, right? Interestingly, nobody thinks to wonder what it means to left-shift std::cout by the amount "hello world". In the context of output stream operations, << means something else entirely. (To this day, some folks have a hard time accepting that. Those folks should read no further.)

Are there other ways to understand expressions? Consider the C++ type system and object model. A C++ expression has a type and a value that can be completely independent of its syntax and semantics. std::cout << "hello world" has type std::ostream & and value std::cout. That’s mostly orthogonal to the fact that it has the effect of writing "hello world" to an output stream.

More? How about the conceptual level? If cont is an STL container, then we know (A) that cont.begin() parses as valid C++, (B) that it means “get the begin iterator”, and (C) that it surely has a type and a value; but we also know that its type models a [*concept*](http://www.sgi.com/tech/stl/Iterators.html) like *ForwardIterator* or *RandomAccessIterator*. In short, expressions are not as simple as one might think!

**Back to EDSL Design**

What does this have to do with EDSL design? Writers and readers of EDSLs in C++ must be able to move throught these different levels when reading and writing domain-specific code. arg1 + 42 parses as C++, has a type and a value (a tree that represents the expression), and its type models a concept called *ProtoExpression*. So when I casually say that arg1 + 42 is a “Proto expression”, I’m really saying all of the above.

*Models of ProtoExpression have certain properties; for example, you can test to see if a Proto expression represents a terminal or non-terminal; if it’s a non-terminal, you can ask for its child expressions (which are themselves Proto expressions); etc. All Proto expressions support these operations.*

There’s not much to say about the semantics of Proto expressions besides the fact that they tend to clump together to form larger and larger Proto expressions. In particular, arg1 + 42 does not yet have “lambda” semantics in and of itself; you must pass it to the Lambda algorithm which *interprets* it as a lambda expression. We’d like to define a *LambdaExpression* concept that refines *ProtoExpression* by adding operator(), and we’d like arg1 + 42 to model that new concept. Now, when I say “lambda expression”, all that information—syntax, semantics, types, values, concepts—comes along for the ride.

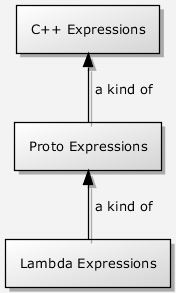
[](http://cpp-next.com/wp-content/uploads/2010/10/05-fig3.dot_.notugly.png)

Figure 1: Relationships between different kinds of expressions.

There’s a simple relationship between the expressions I’ve been discussing: a lambda expression is a kind of Proto expression, which is a kind of C++ expression. These relationships are shown in Figure 1. At the lowest level—syntax—they’re all the same thing. At higher levels, they diverge, and it’s up to the EDSL user to keep it straight. This doesn’t have to be hard; after all, when was the last time you looked at std::cout << "hello world" and wondered what it meant?

**Conceptual Difficulties**

As defined above, arg1 + 42 is a Proto expression. But we’re building a lambda library, dammit—we don’t want no stinkin’ Proto expressions, we want lambda expressions! How do we create a type that models the concept *LambdaExpression*? We can easily add operator() to arg1 such that arg1(1) returns 1. But we haven’t told anybody about our operator(), so as far as Proto is concerned arg1 is still just a *ProtoExpression*. Worse, arg1 + 42 is also just a *ProtoExpression*, without our special operator() member. Proto has effectively sliced it off!

*Why doesn’t arg1 + 42 have an operator()? Although you created arg1 and are free to tweak its interface, arg1 + 42 is an object that Proto has created (the + in arg1 + 42 is provided by Proto). Unless Proto knows (A) that its operator+ should return a lambda expression, and (B) how to create one, it won’t.*

There needs to be a way to “sub-class” Proto expressions to create lambda expressions in such a way that they combine into larger lambdas and not degenerate back into boring old Proto expressions. In Proto, the process of sub-classing Proto expressions is called *expression extension*, and it involves telling Proto a little something about your *domain*.

**Proto Domains**

In Proto, you can extend expressions by defining a *domain* and assigning properties (like extra member functions) to expresions within that domain. What’s a domain? In the abstract domain-specific-language sense, a domain is a kind of intelectual scope; problems within a domain all tend to share characteristics, as do their solutions. Linear algebra is a domain; text manipulation is a domain; etc. A Proto domain is similar: all Proto expressions that share a Proto domain have the same properties, like extra member functions.

A Proto domain is just a C++ type. Like a trait, a Proto domain tells you things about an EDSL. All Proto expressions have an associated Proto domain that describes what makes expressions in that domain special. You can query for it with proto::domain\_of, but your rarely need to. Since it’s awkward to say, “expression types have associated domain types,” we just say “expressions are *in* domains.”

*How does Proto know what domain a new expression should have? Proto checks the domains of the child expressions. If they match, the parent expression also has that domain. What about 42? Obviously, 42 is not in any particular domain, so Proto assigns it to the so-called “default” domain. Expressions in the default domain are not very assertive; they assume the domain of whatever expression they find themselves in.*

The most important job of a Proto domain is to communicate to Proto how it should create new expressions in that domain. By default, expressions are in proto::default\_domain, and there’s nothing special Proto needs to do when building new expressions. But if you assign a custom domain to an expression (we’ll see how in a minute), then you can tell Proto to wrap all new expressions with a custom expression wrapper where you can add extra behaviors—like an operator() member function, for instance.

The domain-specific expression wrappers are known as *expression extension classes*. You tell Proto about them via a domain with a type called a *generator*, described below. So in short, we have:

**Domain**

A type associated with a Proto expression. A Proto domain is a trait with a number of associated types that tell Proto about an EDSL. Larger expressions share the domain of the smaller ones from which they’re composed. One of the types associated with a Proto domain is a generator.

**Generator**

A function object that accepts accepts a Proto expression object and returns a new object—a lambda, perhaps—that *extends* the expression. Usually, this simply involves wrapping it in a custom wrapper.

**Expression Extensions**

Custom wrappers are called *expression extensions*, and they are where all the domain-specific stuff like custom member functions live.

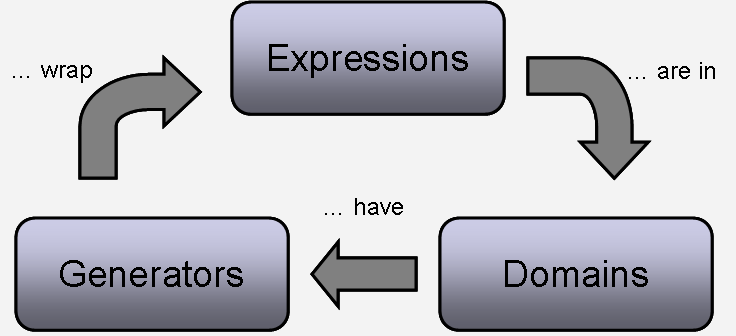
[](http://cpp-next.com/wp-content/uploads/2010/10/05-concepts.png)

Figure 2: Relationship between Proto expressions, domains and generators.

There’s a loopy relationship between these three concepts: for instance, lambda expressions are in the lambda domain, which has an associated lambda generator, which creates lambda expressions, closing the circle. The relationship between expressions, domains and generators is shown in Figure 2 to the right.

Some code will make this a little more real.

**The Lambda Domain, Generator, and Extension**

In the code below, you will see how the three concepts described above—domains, generators, and expression extensions—play together to create lambda expression. First, we define lambda\_domain:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Forward-declare our custom expression wrapper  template<typename ProtoExpression> struct lambda\_expr;    // Define a lambda domain and its generator, which  // simply wraps all new expressions our custom wrapper  struct lambda\_domain  : proto::domain< proto::generator<lambda\_expr> >  {}; |

*We described the loopy relationship between expressions, generators and domains. That loopiness manifests itself in our code by forcing us to forward-declare lambda\_expr. That’s how we break the loop.*

On line 2 we forward-declare our lambda expression wrapper. That lets us use it on line 7 when defining lambda\_domain. The lambda\_domain struct is how we tell Proto how we would like it to post-process all new lambda expressions; namely, to pass them through proto::generator<lambda\_expr>. That’s a function object that wraps expressions in the (yet to be defined) lambda\_expr.

Now we define lambda\_expr, our custom expression wrapper. This is where we put our domain-specific goodies (explained below):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | // A lambda is an expression with an operator() that  // evaluates the lambda.  template<typename ProtoExpression>  struct lambda\_expr  : proto::extends<ProtoExpression, lambda\_expr<ProtoExpression>, lambda\_domain>  {  lambda\_expr(ProtoExpression const &expr = ProtoExpression())  : lambda\_expr::proto\_extends(expr)  {}    // So that boost::result\_of can be used to calculate  // the return type of this lambda expression.  template<typename Sig> struct result;    template<typename This, typename Arg>  struct result<This(Arg)>  : boost::result\_of<Lambda(This const &, Arg const &)>  {};    // Evaluate the lambda expressions  template<typename Arg1>  typename result<lambda\_expr(Arg1)>::type  operator()(Arg1 const & arg1) const  {  return Lambda()(\*this, arg1);  }  }; |

Line 5—inheritance from proto::extends—is where the associations between the lambda domain, generator, and extension class are established. The three template parameters are: the Proto expression we’re extending, the lambda expression we’re defining, and the domain with which all lambda expressions are associated. The constructor on line 7 is some necessary boilerplate, and the rest is up to you.

*And if we were being good citizens, we would* [*validate the template parameter*](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/) *of operator() before passing the current expression to Lambda. Checks omited for brevity’s sake.*

Line 23 is where we (finally!) give lambda expressions their operator(). On line 25, we use the Lambda algorithm that we defined last time. Note that we pass \*this as the first parameter. The Lambda algorithm expects to be passed a model of the *ProtoExpression* concept. Since \*this refers to an instantiation of lambda\_expr which models *LambdaExpression*, and *LambdaExpression* refines *ProtoExpression*, this works.

The only other interesting thing is the nested result class template and the use of boost::result\_of to calculate the return type of the Lambda algorithm. Lambda is a valid [TR1-style function object](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1454.html) (inheritance from proto::or\_ makes it so). And with the addition of the nested result class template, all lambda\_expr objects are valid TR1-style function objects as well.[1](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/#fn:decltype)

**Making arg1 A Lambda Expression**

We’re almost done. All that’s left is to make arg1 a lambda expression so that the expressions in which it appears are also lambda expressions. To do that, we just wrap it in lambda\_expr:

// Define arg1 as before, but wrapped in lambda\_expr

typedef lambda\_expr<proto::terminal<arg1\_tag>::type> arg1\_type;

arg1\_type const arg1 = arg1\_type();

With this last change, every expression involving arg1 gets infected with lambda-ness and has an operator() that evaluates the lambda expression. *Now* we’re done. (For now.)

**Complete Solution**

When all the parts are assembled, our little lambda library looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | #include <boost/proto/proto.hpp>  using namespace boost;    struct arg1\_tag {};    // Forward-declare our custom expression wrapper  template<typename ProtoExpression> struct lambda\_expr;    // Define a lambda domain and its generator, which  // simply wraps all new expressions our custom wrapper  struct lambda\_domain  : proto::domain< proto::generator<lambda\_expr> >  {};    // A Proto "algorithm": a grammar with embedded transforms for  // evaluating lambda expressions (explained in the previous article):  struct Lambda  : proto::or\_<  // When evaluating a placeholder terminal, return the state.  proto::when< proto::terminal<arg1\_tag>, proto::\_state >  // Otherwise, do the "default" thing.  , proto::otherwise< proto::\_default< Lambda > >  >  {};    // A lambda is an expression with an operator() that  // evaluates the lambda.  template<typename ProtoExpression>  struct lambda\_expr  : proto::extends<ProtoExpression, lambda\_expr<ProtoExpression>, lambda\_domain>  {  lambda\_expr(ProtoExpression const &expr = ProtoExpression())  : lambda\_expr::proto\_extends(expr)  {}    // So that boost::result\_of can be used to calculate  // the return type of this lambda expression.  template<typename Sig> struct result;    template<typename This, typename Arg>  struct result<This(Arg)>  : boost::result\_of<Lambda(This const &, Arg const &)>  {};    // Evaluate the lambda expressions  template<typename Arg1>  typename result<lambda\_expr(Arg1)>::type  operator()(Arg1 const & arg1) const  {  return Lambda()(\*this, arg1);  }  };    // Define arg1 as before, but wrapped in lambda\_expr  typedef lambda\_expr<proto::terminal<arg1\_tag>::type> arg1\_type;  arg1\_type const arg1 = arg1\_type();    // End lambda library here. Begin test code.  #include <cassert>    int main()  {  int i = (arg1 + 42)(1);  assert( i == 43 );  } |

Excluding comments, blank lines and the test code (which isn’t part of the library), this is 34 lines by my count. Now that we have it working, let’s take it for a spin:

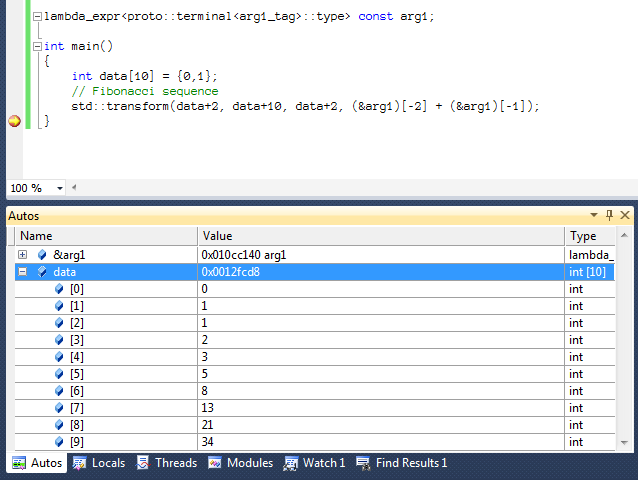
[](http://cpp-next.com/wp-content/uploads/2010/10/05-fig1.png)

Figure 3: Computing the Fibonacci Numbers

The above code uses our 34-line lambda library to fill an array with the first ten [Fibonacci numbers](http://en.wikipedia.org/wiki/Fibonacci_number). Not shabby. If you’re confused about how this lambda works, perhaps you’re thrown by the expression &arg1. Proto has overloaded unary operator& so that &arg1 builds a Proto expression tree. The tree for (&arg1)[-2] + (&arg1)[-1] looks like Figure 4.

*Imagine how std::transform evaluates this lambda for the elements [data+2, data+10):*

*// Standard implementation of transform algo*

*for(; begin != end; ++begin, ++out )*

*\*out = fun(\*begin);*

*Substituting data+2 for begin, data+10 for end, data+2 for out and our lambda expression for fun, we get:*

*int \*begin = data+2, \*out = data+2;*

*for(; begin != data+10; ++begin, ++out )*

*\*out = ((&arg1)[-2] + (&arg1)[-1])(\*begin);*

*Substituting \*begin for arg1 (which is handled by the Lambda algorithm), we get:*

*int \*begin = data+2, \*out = data+2;*

*for(; begin != data+10; ++begin, ++out )*

\*out = (&\*begin)[-2] + (&\*begin)[-1];

*Now the classic Fibonacci algorithm emerges: the next element in the sequence is the sum of the previous two.*[*2*](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/#fn:fib_caveat)

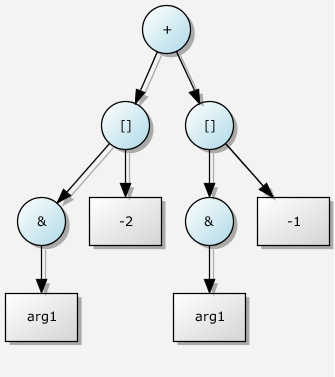
[](http://cpp-next.com/wp-content/uploads/2010/10/05-fig2.dot_.notugly.png)

Figure 4: Expression tree for (&arg1)[-2] + (&arg1)[-1]

**Order of Initialization**

There is one small problem with our solution, but unless you were a language lawyer, I wouldn’t expect you to spot it: the arg1 placeholder requires *dynamic initialization*. This means that it must be constructed at runtime. The problem is that there could be initialization order dependencies that cause arg1 to be used before its constructor is executed. That’s not good.

We can avoid the order-of-initialization problem by making arg1 [POD](http://en.wikipedia.org/wiki/Plain_old_data_structure) and using *static initialization*. In C++03, that means lambda\_expr can’t have a constructor or use inheritance.[3](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/#fn:pod) No problem; Proto offers another mechanism for extending expressions: BOOST\_PROTO\_EXTENDS and friends. We can redefine lambda\_expr as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // A lambda is an expression with an operator() that  // evaluates the lambda.  template<typename ProtoExpression>  struct lambda\_expr  {  BOOST\_PROTO\_BASIC\_EXTENDS(ProtoExpression, lambda\_expr, lambda\_domain)  BOOST\_PROTO\_EXTENDS\_ASSIGN()  BOOST\_PROTO\_EXTENDS\_SUBSCRIPT()    // So that boost::result\_of can be used to calculate  // the return type of this lambda expression.  template<typename Sig> struct result;    template<typename This, typename Arg>  struct result<This(Arg)>  : boost::result\_of<Lambda(This const &, Arg const &)>  {};    // Evaluate the lambda expressions  template<typename Arg1>  typename result<lambda\_expr(Arg1)>::type  operator()(Arg1 const & arg1) const  {  return Lambda()(\*this, arg1);  }  }; |

We had to replace inheritance from proto::extends and the lambda\_expr constructor with three macros. The first on line 6 establishes the relationships between the lambda domain, generator and extension class. The second and third macros define tree-building operator= and operator[] member functions, respectively. (Admittedly, the macros are distasteful but necessary with C++03. The relaxed rules for POD types and constexpr constructors in C++0x will eliminate the need for this hoop-jumping.)

Now that we have changed lambda\_expr to allow static initialization, we can use it when initializing the arg1 placeholder as follows:

lambda\_expr<proto::terminal<arg1\_tag>::type> const arg1 = {};

The curly brace initialization (called *aggregate* initialization) reflects the fact that lambda\_expr no longer has a constructor. And since initializing lambda\_expr objects now requires a different syntax, we must change lambda’s generator when defining lambda\_domain as follows:

// Define a lambda domain and its generator, which

// simply wraps all new expressions our custom wrapper

struct lambda\_domain

: proto::domain< proto::pod\_generator<lambda\_expr> >

{};

Notice that we replaced proto::generator with proto::pod\_generator. The only purpose of this change was to get Proto to use curly-braces when initializing new lambda\_expr objects instead of the regular constructor-call syntax.

[Click here to view the complete solution.](http://cpp-next.com/wp-content/uploads/2010/10/05-mini-lambda.txt)

**Conclusions and What’s To Come**

With the above changes to fix the order-of-initialization problem (and the qualifications above), our mini lambda library now has 32 lines of code by my count. OK, so it’s not quite 30 as promised. Bygones? I hope you’ve enjoyed this introduction to Proto grammars, transforms, and extension classes. We’ll be seeing a lot more about grammars and transforms in future articles, but at this point, you have most of what you need to start building your own embedded languages with Proto.

In the next article, we’ll address the one major shortcoming of this lambda EDSL: the fact that it only accepts a single argument. Accepting additional arguments is a simple matter, and along the way we’ll see how to compose Proto transforms, making them far more flexible and powerful.

See you then!

**Acknowledgements**

Thanks to [Bartosz Milewski](http://bartoszmilewski.wordpress.com) and [David Abrahams](http://www.boostpro.com) for their valuable feedback about this post.

1. On C++0x compilers, the nested result class template is unnecessary because std::result\_of uses decltype to deduce the return type of operator(). [↩](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/#fnref:decltype)
2. Since this lambda works by using pointer manipulation, it only works on contiguous data. Careful! [↩](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/#fnref:fib_caveat)
3. In C++0x, the situation is different. Some types that use inheritance and constructors still qualify for static initialization (see [standard layout types](http://en.wikipedia.org/wiki/C%2B%2B0x#Modification_to_the_definition_of_plain_old_data)). But if you want your EDSL to be portable to C++03 compilers, you need to respect the stricter C++03 rules. [↩](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/#fnref:pod)

Posted Sunday, October 17th, 2010 under [Boost](http://cpp-next.com/archive/category/boost/)