Posted October 04, 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-one/>

## Expressive C++: A Lambda Library in 30 Lines (Part 1 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)**
5. [Expressive C++: A Lambda Library in 30 Lines (Part 2 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/)
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/)

In this installment and the next, we’ll see a bit more about grammars, introduce transforms for evaluating expressions, and dig deeply into expression extension: giving Proto expressions—dumb, static trees up till now—interesting behaviors of their own. Doing this will open a world of expressive opportunities. By the end of this article, you’ll know how to use grammars and transforms to evaluate expressions in interesting ways. After the next article, you’ll know how to define a Proto domain and give your expressions domain-specific behaviors that make your EDSL come alive. Along the way, we’ll define a little lambda library[1](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/#fn:1) that, despite its micro size, is actually useful.

**UPDATE**

In the course of this series, I have made every effort to keep my posts accessible and interesting to intermediate-level C++ programmers. Based on some feedback, I’ve realized that I missed the mark with this one. Look for more of these “Update” notes as I try to clarify some of the more opaque concepts and arcane syntax that appears in this post.

### STL Algorithms, Function Objects, and Lambdas

The STL brought separation of data structures and algorithms, allowing the same algorithm to work efficiently on lists, maps, vectors and even user-defined types. It also brought a proliferation of one-off function objects to customize the behavior of the algorithms, and a small, exotic zoo of binders and adaptors. For instance, if you want to add a constant to every element in a sequence, you can either define your own function object for use with std::transform or use the standard binders, as below:

#include <algorithm>

#include <functional>

// Define a function object for adding a constant to a value

struct add\_constant

{

typedef int result\_type;

explicit add\_constant(int i)

: i\_(i)

{}

int operator()(int i) const

{

return i + i\_;

}

private:

int i\_;

};

int main()

{

int data[10] = {0,1,2,3,4,5,6,7,8,9};

// Add 42 to each element in "data" using a stand-alone function object.

std::transform( data, data+10, data, add\_constant(42) );

// Same as above, but using standard binders function function objects.

std::transform( data, data+10, data, std::bind1st( std::plus<int>(), 42 ) );

}

Each call to std::transform above adds 42 to each element in the data array. The problem with defining stand-alone function objects like add\_constant is that it requires a lot of boilerplate. Also, it needlessly separates the action from the invocation. In comparison, the next line that uses the binder and the function object is a paragon of terseness. (That’s not saying much.) But as function objects get more complicated, the readability of binders and function objects drops precipitously, and in many cases, no combination of binders, adaptors, negators and function objects will suffice.

C++0x has a much better solution: first-class lambda functions. The above example is tidier with lambdas:

// Add 42 to each element in an array using C++0x lambdas.

std::transform( data, data+10, data, [] (int j) { return j + 42; } );

C++0x lambdas scale much better than binders when expressions get complicated, and I heartily endorse them. If your compiler doesn’t support them yet, you have a few options:

Do C++0x lambdas obviate the need for library solutions like [Boost.Phoenix](http://www.boost.org/libs/spirit/phoenix/doc/html/index.html)? Not entirely. One reason is that, in contrast to C++0x lambdas, Phoenix lambdas are polymorphic: you don’t have to say up front what kinds of arguments they accept. In other words, the same Phoenix expression can be used to add 42 to arrays of int and MyNiftyBigInt, something C++0x lambdas can’t do. But the real reason Phoenix will continue to rock even in C++0x is that a Phoenix expression has a tree structure that you can inspect and manipulate at compile time. Much more on that in future articles.

* Run out and get one that does.
* Use [Boost.Phoenix](http://www.boost.org/libs/spirit/phoenix/doc/html/index.html)[2](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/#fn:2).
* Roll your own lambda library in about 30 lines of code with Boost.Proto.

Though there are good reasons to not roll your own, doing so is the first step on a journey of discovery that, over the course of this article series, will shed light on all the inner workings of Boost.Phoenix and put the full power of Proto in your hands.

### Lambda in a Library

Using Boost.Phoenix, the solution to the above problem looks like this:

// Use Boost.Phoenix to add 42 to all the elements of an array.

using boost::phoenix::arg\_names::arg1;

std::transform(data, data+10, data, arg1 + 42 );

The expression arg1 + 42 creates an object with a function call operator: a function object. The object arg1 defined by Phoenix is a so-called placeholder that will be substituted with actual values when the function object is invoked. In other words: (arg1 + 42)(1) ⇒ (1 + 42) ⇒ (43).

How does that work? If you’ve read previous articles in this series, you can probably guess that the expression arg1 + 42 builds some kind of a tree representation, but it’s a funny kind of tree: one that has an operator() that does actual work. That is, it’s a tree with domain-specific behavior. So far, the trees Proto has built for us have been totally boring and domain-independent. Somehow we need to get Proto to build not just a tree, but a lambda tree.

But before we learn that new trick, let’s get the expression arg1 + 42 compiling first (explanation to follow).

#include <boost/proto/proto.hpp>

using namespace boost;

struct arg1\_tag {};

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

int main()

{

arg1 + 42;

}

That was pretty easy. The expression arg1 + 42 builds a boring, static tree for us. The expression looks like Figure 1 below.

**UPDATE: Why Is arg1\_tag Empty?**

A common misconception among new users of libraries like Boost.Lambda and Boost.Phoenix is that placeholder objects like arg1 are actually mutable, that at some point they will be assigned the values for which they stand in. This is not the case. Rather, the entire lambda expression, arg1 included, is immutable and unchanged even while the lambda is getting executed. The actual runtime arguments to the lambda are kept in a separate look-aside. The decision about whether to read a value out of the lambda itself or from the look-aside is based on the type of the value. For instance, there is nothing special about the type int, so an integer like 42 in a lambda expression stands for itself. But the type arg1\_tag is treated specially. Phoenix recognizes that terminals of type arg1\_tag stand for data in the look-aside, and it reads the value from there instead.

So arg1 really is just a placeholder—just a stand-in for data found elsewhere. The only important thing about arg1 is its type; hence, it is a terminal of a special type—the arg1\_tag struct—that is empty. This is a very common idiom in EDSL design.

**UPDATE: Brace Initialization Rationale**

I got a few questions about the odd brace-initialization syntax of the arg1 object. What actually is arg1 anyway? Nowhere do I say what proto::terminal<arg1\_tag>::type is a typedef for. Certainly, it’s a terminal node in an expression tree—that’s the abstraction—and its exact type shouldn’t really matter. Sadly in this case, it’s a [leaky abstraction](http://www.joelonsoftware.com/articles/LeakyAbstractions.html); some of the implementation details of the arg1 object are bleeding through. In particular, the fact that the type of arg1 is [POD](http://en.wikipedia.org/wiki/Plain_old_data_structure)—a C-style struct without constructors, base classes, access specifiers or virtual functions—is exposed by the way arg1 needs to be initialized.

By default, all Proto expressions are POD and require brace-initialization syntax. Why? It’s so that objects like arg1 can be statically initialized. When building an EDSL, you often want to define global constants (e.g. arg1) at namespace scope. If these things had constructors that needed to be run before main, then it’s possible to invent scenarios where some globals get used before they are initialized. That’s not good. Instead, by making arg1 POD and using the brace-initializer syntax, C++ guarantees that these objects require no runtime initialization at all; arg1 just is, and you can stop worrying about bizarre bugs caused by order-of-initialization problems.

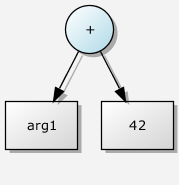
[](http://cpp-next.com/wp-content/uploads/2010/09/04-fig1.dot_.notugly.png)

Figure 1: Expression tree for arg1 + 42

Now for the hard part: making this tree actually do something.

### Introducing Proto Transforms

If you’re at least passingly familiar with compiler construction toolkits like ANTLR, yacc, or Spirit, you know that to get stuff done when parsing, you embed semantic actions in your language’s grammar. The semantic actions get executed when certain parts of the input match parts of your grammar. Proto is no different. (If you’re not familiar with this parsing stuff, sit tight; you will be shortly.) In Proto, semantic actions are called transforms, which are a special kind of function object. They accept expression trees and optionally some additional state and perform some computation.

For EDSLs that look and behave a lot like C++, Proto provides the very handy proto::\_default transform. It gives all the operators their “default” C++ meaning. It’s the “what-would-C++-do?” transform. Consider the following code:

// Use the default expression evaluator.

proto::\_default<> eval;

// Create an expression tree representing the

// expression "1 + 2" and evaluate it.

int i = eval( proto::lit(1) + 2 );

assert( i == 3 );

The first line declares a function object called eval that knows how to evaluate trees that represent the C++ expressions that created them. When we invoke it with proto::lit(1) + 2 we get 3. Easy enough. (Proto::lit is a function that turns its argument into a Proto terminal so that expressions involving it build expression trees.)

If we try to pass to eval the expression “arg1 + 42“, it would choke; proto::\_default doesn’t know what to do with arg1. We’ll teach it below.

**UPDATE**

Transforms are an important topic, and this introduction leaves much unexplained. So let me try to fill in the details. Part of the difficulty in learning transforms is that they constitute their own little programming language with a peculiar syntax. Until you learn this language, transforms will make little sense. So instead of just presenting this syntax in all its awfulness, I’ll try to explain the logic of transforms first using pseudo-C++ that lacks types. Once you see the logic of the transform behind the syntax, it should be easier to see how the actual C++ transform works.

To that end, let’s see what the \_default transform looks like in type-less pseudo-C++:

// Define a \_default function that recursively evaluates

// an expression tree according to the rules of C++

auto \_default( expr )

{

// Dispatch to the correct case based on the

// node type of the expression

switch ( tag\_type( expr ) )

{

case terminal:

return value( expr );

case plus:

return \_default( left( expr ) ) + \_default( right( expr ) );

case minus:

return \_default( left( expr ) ) - \_default( right( expr ) );

// ... handle all the operators similarly

}

}

In the above, tag\_type returns an expression node’s type, left returns a binary node’s left child, and right returns the right child, and value returns the value stored in a terminal node.

Actually, let me expand on that a bit. As I’ve mentioned, all transforms accept an additional state parameter. And the \_default transform can be further parameterized on the operation to perform on each child node. That operation is \_default by default. So in pseudo-code, \_default looks more like this:

// Define a \_default function that recursively evaluates

// an expression tree according to the rules of C++

auto \_default( expr, state = null, eval = \_default )

{

// Dispatch to the correct case based on the

// node type of the expression

switch ( tag\_type( expr ) )

{

case terminal:

return value( expr );

case plus:

return eval( left( expr ), state, eval ) + eval( right( expr ), state, eval );

case minus:

return eval( left( expr ), state, eval ) - eval( right( expr ), state, eval );

// ... handle all the operators similarly

}

}

Back in the world of real C++, eval is a template parameter to \_default, but the end result is the same. Hopefully, that should make the \_default transform a little less mysterious.

### Passing State To Transforms

As I mentioned above, when you evaluate transforms, you can optionally pass additional state. Simply pass the state as a second parameter:

// We can pass an extra parameter to eval. It

// gets ignored here.

int i = eval( proto::lit(1) + 2, 42 );

assert( i == 3 );

The proto::\_default transform doesn’t use the state parameter, so it gets passed around but not used. However, there is a proto::\_state transform that returns the state parameter:

// Use the \_state parameter to fetch the state parameter

proto::\_state get\_state;

int j = get\_state( proto::lit(1) + 2, 42 );

assert( j == 42 );

This may not seem all that useful, but its purpose will be clear soon.

**UPDATE**

In our typeless pseudo-C++, the \_state transform is really very simple:

// Pseudo-C++ equivalent of the \_state transform.

auto \_state( expr, state = null )

{

return state;

}

That should make it pretty clear that \_state does nothing more fancy that returning its second argument.

In an earlier update, I referred to a separate “look-aside” for the arguments passed to a lambda. Although it’s not clear yet, the look-aside will be passed in as the state argument when evaluating the lambda expression. The proto::\_state transform is how the evaluation routine will get access to the lambda’s arguments.

### Grammars + Transforms ⇒ EDSL Bliss

In the last article, I used a grammar to detect invalid expressions and report useful error messages. I also dropped a hint that they were far more useful than that. Now we’ll take a giant step forward and use grammars together with transforms to define our custom lambda expression evaluator. It is described below.

**UPDATE**

Before you read any further and damage your eyes from the ugly syntax, it would be helpful to know in advance what the logic of our transform looks like in our typeless pseudo-C++:

// Pseudo-C++ that demonstrates how to evaluate lambda expressions

auto Lambda( expr, state = null )

{

if ( decltype( expr ) matches? terminal<arg1\_tag> )

return \_state( expr, state )

else

return \_default( expr, state, Lambda )

}

In the above pseudo-C++, I’ve introduced a new operator called matches? that oddly operates on types, not values. (OK, so my pseudo-C++ isn’t really typeless. It’s more like Haskell where types are inferred.) You can use matches? to test at compile time whether an expression node matches a grammar. Let me reemphasize the point that matches? is a compile time test; the runtime value of expr can in no way influence the result of matches?. The grammar terminal<arg1\_tag> matches any terminal whose value type is arg1\_tag. If it does, call the \_state transform, which will simply return the state parameter. Otherwise, do the “default” thing which will evaluate all child nodes with Lambda and combine the results in the C++ way: addition for plus nodes, subtraction for minus nodes, etc. The recursive application of Lambda is what will cause arg1 to be replaced with the state parameter wherever it appears in the expression tree.

OK, it’s now safe to proceed and read the real C++ transform.

// A grammar with embedded transforms for evaluating lambda expressions

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 > >

>

{};

The new bits here are proto::when and proto::otherwise. Proto::when takes two template parameters: a grammar and a transform to execute when an expression matches that grammar. Proto::otherwise just takes a transform, which it executes unconditionally.

Instantiations of Proto::when can be used as both a grammar and as a function object. As a grammar, it matches the same things its first template parameter does. As a function object, it has the same behavior as its second.

Lambda is both a grammar and a function object for evaluating valid lambda expressions. As a grammar, it reads as follows: An expression is a Lambda either if it is an arg1\_tag terminal (like arg1), or if it is anything else. (Admittedly, that’s not a very interesting grammar.) As a function object, Lambda works as follows: for arg1\_tag terminals, return the current state. For everything else, do the “default” thing by recursively evaluating all child nodes according to the Lambda grammar (specified by the Lambda template parameter to \_default), and then combining the results using the operator that created the parent node.

**UPDATE : Pseudo-C++ vs. Real C++**

How does the real C++, which seems deeply magical, relate to the pseudo-C++, which seems simple and straightforward? In particular, in the pseudo-C++ definition of Lambda, we can clearly see arguments getting passed around, whereas in the real C++ definition of Lambda, it appears that no argument passing is going on at all. It is, but it’s implicit.

In the pseudo-C++ definition of Lambda, both the \_state and \_default transforms get passed the current expr and state parameters unmodified. Similarly, when evaluating the real Lambda, Proto knows what the current values of expr and state are at any given time and pass them to \_state and \_default. You don’t have to tell it to.

But what is proto::\_state? It’s a class that implements the [TR1 result\_of protocol](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1454.html). Glossing over some details, it looks more or less like this:

// An approximate implementation of the \_state transform, that

// simply returns its second parameter

struct \_state

{

template<class Sig> struct result;

template<class This, class Expr, class State>

struct result<This(Expr, State)>

{

typedef State type;

};

template<class Expr, class State>

State operator()(Expr const & expr, State const & state) const

{

return state;

}

};

This is seriously verbose because two “computations” are happening here: a compile-time one for calculating the return type via the nested result template, and a runtime one for simply returning the second argument. But now that we’ve grouped the compile-time and runtime calculations together and given them a name (\_state), we can reuse that name in our transforms to represent both the compile-time and a runtime calculation that corresponds to returning the current state.

And to get back to the original question, Proto “knows” that the second parameter to proto::when (and the first parameter to proto::otherwise) is a [TR1-style function object](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1454.html) that accepts the current expr and state parameters. So Proto passes them along without you having to tell it to. It’s implicit.

**UPDATE: Expressions, Grammars, and Transforms**

Why is it interesting to note that Lambda is both a grammar and a function object? When used as a function object, the first parameter is the expression to evaluate. You can think of a grammar with embedded transforms as an algorithm for processing expressions. The grammar is used to manage the control flow within the algorithm, using pattern matching against nodes to find the correct transform to apply. These algorithms have an important precondition: that the expression passed to the algorithm matches the grammar used to drive the algorithm. For instance:

// Assert at compile-time that a given expression matches

// the Lambda grammar before applying the Lambda algorithm:

static\_assert(

proto::matches< decltype(arg1 + 42), Lambda >::value,

"This expression had better type-check against the Lambda grammar!");

// OK, it's valid. Evaluate it.

Lambda()( arg1 + 42, 1 );

In this case, the Lambda grammar is degenerate; because of the use of proto::otherwise, which is a catch-all, the grammar will always match everything, but that’s not generally the case.

There is a larger point to make about the relationship between expressions, grammars, and transforms. In terms that might be more familiar, we can say:

**Expression**

Data: No interesting behavior.

**Grammar**

Schema: A description of valid data.

**Transform**

Function: Accepts a piece of data and returns a new value.

**Grammar+Transform**

Algorithm: A step-by-step procedure for applying functions to data that conforms to the schema.

Hopefully, this all hangs together a bit more coherently now.

We can use Lambda as follows:

// Evaluate the lambda expression with 1 as the state

int k = Lambda()( arg1 + 42, 1 );

assert( k == 43 );

At each stage in the evaluation of arg1 + 42 with Lambda, it does a pattern match: does the current node look like an arg1 terminal? If yes, return the state parameter. Otherwise, invoke Lambda recursively on the children (passing along the state) and combining the results according to the rules for C++ expressions. For instance,

Actually, it would expand to something more like:

Lambda()(arg1, 1)

+ Lambda()(proto::lit(42), 1)

Why proto::lit? As Proto builds expression trees, it grabs non-Proto objects like 42 and turns them into Proto terminals before glomming them onto the tree. That wrapped integer terminal is what will eventually get passed to Lambda, not the raw 42 integer. And what does \_default do with terminals? It simply unwraps them.

Lambda()(arg1+42, 1)

expands roughly (see sidebar) to:

Lambda()(arg1, 1) + Lambda()(42, 1)

which further expands to 1 + 42 and ultimately to 43.

### What’s To Come

We now have a placeholder with which we can build all kinds of crazy lambda expressions and a custom evaluator called Lambda that we can use to evaluate them. What we don’t have yet is the operator() that will turn our lambda expressions into proper function objects that we can pass to standard algorithms. For that, we’ll need to learn how to extend Proto expressions, and that’s a whole `nuther topic. In the next installment, we’ll pick up where we left off and finish this little lambda library.

Until then…