

# Proto: A Compiler Construction Toolkit for DSELs

Eric Niebler  
Boost Consulting  
1608 E Republican St. 202  
Seattle, WA 98112  
eric@boost-consulting.com

## ABSTRACT

A Domain-Specific Embedded Language (DSEL) is a miniature language-within-a-language for solving problems in a particular domain. One technique for creating efficient and expressive DSELs in C++ is to use expression templates, but this technique is not for the faint of heart. Such libraries are difficult to write and maintain due to the esoteric nature of template meta-programming, and difficult to use because of the often impenetrable compiler error messages they generate. Existing tools help somewhat, but do not provide the support that language designers have come to expect: something like BNF for defining the language's grammar and associated semantic actions. This paper describes Proto, a C++ library that implements a compiler construction toolkit for embedded languages. The benefits of grammar-based DSELs are shown by contrasting them to other existing approaches to DSEL design. The nature of embedded languages with constrained grammars and their implications for a embedded compiler construction toolkit is briefly explored. Some examples are shown where library interfaces can be made more expressive through the use of grammar-based DSELs.

## 1. INTRODUCTION

Domain-specific embedded languages raise the level of abstraction, allowing programmers to express solutions in a way that naturally suits the domain in which they are working. Examples include POOMA [2] for parallel physics computation, and Boost.Spirit [3]<sup>1</sup> for parser generation. These two libraries use *expression templates* [18] to define an embedded language within C++. The technique involves overloading operators to build a tree that represents an expression, which is evaluated later. Spirit has a hand-crafted solution, whereas POOMA is built with the help of PETE [1], the Portable Expression Template Engine.

<sup>1</sup>This paper refers to the currently available version of Spirit, not Spirit-2 which is currently under development.

There are good reasons to offload the work of building and processing expression templates to a separate library as POOMA does, not the least of which is isolating implementation complexity. However, as shown below, PETE is lacking in some respects that make it ill-suited to the task of defining embedded languages. This paper presents Proto, a compiler construction toolkit for expression template-based DSELs. Proto provides tools for the following:

1. *Expression Construction*: Defining leaves and building expression trees.
2. *Expression Evaluation*: Immediately evaluating an expression.
3. *Expression Introspection*: Discovering the structure of an expression tree by matching it against a grammar. The grammar is defined declaratively using templates in a rough approximation of BNF.
4. *Expression Transformation*: Transforming expression trees into other types of objects by decorating grammars with *transforms*. These can be thought of as the analogue of semantic actions. A transformation might be as simple as counting the number of leaves in an expression tree or as complicated as building an NFA for matching a regular expression.
5. *Expression Extension*: Bestowing expressions with additional behaviors and members. For example, expressions in a linear algebra DSEL such as `(vector1 + vector2)` [3] would need an additional `operator[]` member that evaluates expressions at an index without creating temporary vectors.

The rest of this paper is organized as follows. The second section explores the implications of embedded languages on compiler construction toolkits. The third section shows a simple example to demonstrate the basic concepts in Proto. The fourth section describes the existing state of the art in expression template-based libraries. The fifth section explores Proto in depth and contrasts its approach with the others. The sixth section presents some real-world library design problems and shows how Proto might be used to solve them.

## 2. EMBEDDED LANGUAGES

In 1998, Todd Veldhuizen gave a talk entitled “The Library That Thinks It Is A Compiler” [17] in which he describes how his Blitz++ library [16] makes use of expression templates to define a domain-specific embedded language for numerical computing. In this light, it seems logical to consider the possibility of a library that thinks it is a compiler construction toolkit. But what exactly does it mean to host both one language and its compiler construction toolkit wholly within another language, and what benefits does it bring?

The benefit of toolkits such as yacc [12] is that they allow language designers to work at a higher level of abstraction by writing grammars and semantic actions directly instead of hand-coding parsers, which can be tedious and error-prone. But hand-coded parsers are currently the state of the art in expression template-based DSELS such as Blitz++, POOMA, The Boost Lambda Library [9] and Spirit, none of which rigorously codifies the grammar of its embedded language.

C++ does not directly support embedded languages; rather, by pressing the template and operator overloading mechanisms into service, expressions can be made to build trees, which can then be evaluated by library code in a domain-specific way. Unlike other compiler construction toolkits that consume sequences of tokens, Proto consumes trees that already conform to the grammar for valid C++ expressions. A DSEL grammar constrains expressions further by defining the subset of expressions that are valid within a domain. The implication is that an embedded compiler construction toolkit need only be powerful enough to recognize and manipulate expression trees that conform to their host grammar, not token sequences.

Consider a simple grammar to recognize infix arithmetic expressions. In EBNF, it looks like the following:

```
<group> ::= "(" <expr> ")"
<fact>  ::= <integer> | <group>
<term>  ::= <fact> (("*" <fact>) | ("/" <fact>))*
<expr>  ::= <term> ( "+" <term> ) | ( "-" <term> )*
```

This grammar uses alternation, sequencing, repetition and recursion; and encodes the associativity and precedence of the arithmetic operators. Since Proto only sees expressions after they have been assembled into trees according to C++’s rules, parent-child relationships take the place of sequencing and repetition in Proto grammars. Proto uses an `or_<>` template to express alternation and parameterization on incomplete types to express recursion. Here, for instance, is the equivalent grammar implemented with Proto.

```
struct Expr;
struct Int  : terminal<int> {};
struct Plus : plus<Expr, Expr> {};
struct Minus : minus<Expr, Expr> {};
struct Mult  : multiplies<Expr, Expr> {};
struct Div   : divides<Expr, Expr> {};
struct Expr  : or_<Int,Plus,Minus,Mult,Div> {};
```

The line `struct Plus : plus<Expr, Expr> {};` means that `Plus` recognizes binary `+` expressions where both operands conform to the `Expr` grammar. No care is needed to encode operator associativity or precedence into Proto grammars; the rules for valid C++ expressions can simply be assumed. The following sections show what such a grammar can be used for.

## 3. A LAZY INPUT DSEL

A simple example of Proto will demonstrate one of the uses for a grammar-based DSEL. The code below defines a little DSEL for deferred read operations. It uses Proto to build expressions and introspect them at compile-time to differentiate between expressions that read from streams and those that read from serialization archives.<sup>2</sup>

```
#include <iostream>
#include <boost/utility/enable_if.hpp>
#include <boost/xpressive/proto/proto.hpp>
using namespace boost;
using namespace proto;

// Hypothetical serialization archive
struct archive { /* ... */ };

struct Stream :
    or_<shift_right<terminal<std::istream&>, _>,
        shift_right<Stream, _> >
{};

struct Archive :
    or_<shift_right<terminal<archive&>, _>,
        shift_right<Archive, _> >
{};

// Read from a stream:
template<typename Expr>
typename enable_if<matches<Expr, Stream> >::type
read(Expr const &expr)
    {std::cout << "Stream!\n";}

// Read from an archive:
template<typename Expr>
typename enable_if<matches<Expr, Archive> >::type
read(Expr const &expr)
    {std::cout << "Archive!\n";}

archive ar;
terminal<std::istream &>::type cin_ = {std::cin};
terminal<archive &>::type ar_ = {ar};

int main()
{
    int i, j, k;
    // Prints "Stream!"
    read( cin_ >> i >> j >> k );
    // Prints "Archive!"
    read( ar_ >> i >> j >> k );
}
```

<sup>2</sup>All the code in this paper uses the version of Boost in the HEAD branch of its CVS repository on SourceForge.net, dated 2007-07-20.

The line `terminal<std::istream &>::type cin_ = {std::cin};` declares an object `cin_` that behaves as a leaf node in an expression tree. Uses of this object such as “`cin_ >> i >> j >> k`” build expression trees using operator overloads found in Proto’s namespace via argument-dependent lookup.

The `Stream` structure definition defines the grammar of input stream expressions:

```
struct Stream :
    or_<shift_right<terminal<std::istream&>, _>,
        shift_right<Stream, _> >
{};
```

This says that an input stream expression is either a right-shift expression where the left operand is a `std::istream`, or a right-shift expression where the left operand matches the `Stream` grammar. In either case, the right operand is allowed to be anything, as specified by Proto’s `_` wildcard. If we let `<expression>` be the grammar for C++ expressions and `istream&` stand in for a terminal of that type, then the EBNF grammar corresponding to `Stream` is as follows:

```
<Stream> ::= istream& ">>" <expression>
          | <Stream> ">>" <expression>
```

In the `Stream` struct definition, the `shift_right<>` template corresponds to C++’s `>>` operator (which depending on the particular DSEL in question may or may not correspond semantically to a right-shift operation). Notice that the definition of `Stream` is recursive—at the point in the definition that `Stream` refers back to itself, it is incomplete. It will be complete by the time `matches<>` uses the grammar.

The `Stream` grammar and the similarly defined `Archive` grammar are used with Proto’s `matches<>` meta-function, which compares an expression type to a grammar definition and yields a compile-time Boolean representing whether the expression type matches the grammar or not. In this case, the Boolean is used with Boost’s `enable_if<>` [11] to select among overloads of the `read()` function.

The use of grammars and `matches<>` results in a declarative style that is understandable at a high-level and largely obviates the need to write the complicated recursive template specializations that are typical of template meta-programs. As shown below, DSEL grammars bring other benefits as well.

## 4. CURRENT IMPLEMENTATION TECHNIQUES

POOMA and Spirit use expression templates for different reasons, and that affects their implementation choices. POOMA uses expression templates primarily for the performance advantage. By delaying evaluation of complicated expressions until the full expression is available, expression templates make the job of an optimizing compiler easier. Also, expression templates

allow POOMA to do domain-specific optimizations such as loop unrolling and elimination of temporary objects for holding intermediate results. However, the operators themselves retain their usual meaning: `+` means addition, `*` means multiplication, etc. For a library such as POOMA, having flexible mechanisms for *evaluating* an expression is very important. PETE, the expression template library upon which POOMA is built, provides such functionality.

### 4.1 PETE: the Portable Expression Template Engine

PETE provides tools for building expression templates and the `forEach()` algorithm for evaluating them. The behavior of `forEach()` can be customized by partially specializing PETE’s class templates. For instance, a typical invocation of PETE’s `forEach()` algorithm looks like this:

```
template<class T>
void evaluate(const Expression<T> &expr)
{
    forEach(expr, MyLeafTag(), MyCombinerTag());
}
```

The `MyLeafTag` type specifies how leaf nodes should be evaluated, and the `MyCombinerTag` type specifies how non-terminal nodes should be evaluated. The class templates `LeafFunctor<>`, `Combine1<>`, `Combine2<>`, and `Combine3<>` must be partially specified on these user-defined tag types to customize the behavior of the `forEach()` algorithm.

This framework is conceptually simple and makes expression templates easy to work with, but it has some limitations.

1. The `forEach()` algorithm always evaluates expressions bottom-up, offering no way to express alternate control flow. That means that `forEach()` cannot perform many common transformations; e.g., folding part of an expression tree into a list in either postfix or prefix order. It also means that it cannot short-circuit expression evaluation; e.g. to only evaluate one branch of a ternary conditional expression.
2. There is no easy way to express constraints on valid expressions or to detect and reject invalid expressions.
3. There is no support for sub-domains in which an operator means something different than in the super-domain. This is a problem for a parser generator DSEL like Spirit in which one part of an expression represents an EBNF rule (where `|` means alternation) and another represents an action (where `|` means bitwise-or).
4. There is no way to extend PETE expressions to give them additional behaviors. Consider that in a linear algebra library it would be desirable for the expression `(vector1 + vector2)[4]` to be evaluated as if it were written `vector1[4] + vector2[4]`. That implies that the expression `(vector1 + vector2)` should have an `operator[]` to do the transformation, but PETE gives

no way to define such extra members on expression objects.

Some of these issues can be addressed by extending PETE. For instance, a facility for building DSEL grammars can be layered on top of PETE, and PETE can provide other more flexible mechanisms for expression evaluation and transformation besides `forEach()`. PETE's operator overloads, which are generated with an external tool, can be generated instead with the C++ preprocessor, making PETE wholly embedded. PETE's expression type can be made to handle expressions with a configurable arity, rather than the current hard-coded limit of 3. Other shortcomings are more difficult to overcome. It is not clear how PETE expressions can be extended within a domain to provide additional domain-specific member functions without a significant redesign. Likewise, changing PETE's expression type so that it can be used to define globals that need no dynamic initialization would require considerable work, and is likely to break existing clients.

## 4.2 Boost.Spirit

Spirit, in contrast to POOMA, assigns novel meanings to the operators to better emulate the syntax of EBNF in C++. For instance, unary `+` means “one or more” and `|` means alternation within grammar productions. It is easier to think of Spirit as a language embedded within C++ with semantics all its own. For such embedded languages, it is often the case that the expressions are not in the optimal form for efficient execution. To stretch the “embedded language” metaphor, an optimal strategy may involve “compiling” expressions into some executable form rather than “interpreting” them directly. For a library such as Spirit, mechanisms for *transforming* an expression to some other object—one with the proper domain-specific behaviors—is important.

For the reasons outlined in the previous section, namely the limitations of the bottom-up `forEach()` algorithm, PETE would have been a poor choice for implementing Spirit, which requires the ability to express alternative execution flow. Spirit implements its own custom expression template solution. Although this approach works, it is not without its drawbacks.

1. Spirit's expression template machinery accounts for a large fraction of its implementation complexity, and since it is highly specialized for parser generation, it cannot be reused.
2. There is no easy way to verify that a given expression type is, in fact, a valid Spirit expression except to attempt to transform it into a parser. If that transformation fails, the user is often confronted with an intimidating template instantiation backtrace.
3. Spirit's DSEL is a walled garden; it does not interoperate well with other DSELs. When every DSEL uses its own expression template representation, conflicts arise. Operator overloads become ambiguous and common names such as `_1` and `alpha` clash and need explicit qualification.<sup>3</sup>

<sup>3</sup>It might seem ironic that Proto itself defines common sym-

Spirit is currently undergoing a comprehensive rewrite. The new version is being implemented on top of Proto.

## 4.3 Daixtrose

There is another expression template library besides PETE that is worth mentioning: Daixtrose [20]. Like PETE and Proto, Daixtrose provides facilities for building expression trees; however, it does not provide any support for expression evaluation or transformation, and only rudimentary support for introspection. By default, Daixtrose disallows expressions with leaves of different types, but it provides a traits-based mechanism to allow mixed expressions; e.g., multiplication of matrices and vectors. This is sufficient for many scientific computing applications, but not for general DSELs where the legality of an expression depends on its structure, not just on the types of the terminals.

Unlike PETE, Daixtrose provides support for expression extension by allowing users to specialize a template Daixtrose uses as a base class for expression types. In this way, users can add domain-specific member functions to their expressions. This is an important feature that is useful in many DSEL scenarios.

## 5. PROTO AND DSEL DESIGN

Proto aims to address all the problems described in the previous section and become a unifying *lingua franca* for expression template-based DSELs. For the POOMA scenario, Proto must have good support for expression evaluation. For Spirit, Proto must be good at expression transformation. And for maximum flexibility, Proto should allow expressions within a domain to have extra member functions for providing custom domain-specific behaviors. It must make expression template libraries—notorious for their implementation complexity—easier to implement and reason about, and easier to use by generating concise, helpful compile errors that shield the users from internal details.

### 5.1 Expression Construction

Building expression templates involves defining Proto terminals or functions that return Proto expressions and then letting argument-dependent lookup find the operator overloads defined in Proto's namespace.<sup>4</sup> For instance, an argument placeholder such as that used by `tr1::bind()` [7] might be defined as:

```
template<int I> struct S
{ static int const value = I; };

terminal<S<1> >::type const _1 = {};;
```

Aggregate initialization is necessary because Proto expression types do not define constructors. This is so that terminals such as `_1` above, which are typically const objects

bols that may require qualification, but any such qualification would occur in the DSEL's library code and would not interfere with uses of the DSEL.

<sup>4</sup>It is also possible to non-intrusively adapt non-Proto leaf types to Proto's expression template framework, but that is not described here.

defined at namespace scope, do not require dynamic initialization that could lead to subtle order of initialization bugs and poor application start-up time.

A terminal like `_1` is immediately usable to create expression templates. For instance, the expression “`_1 + 42`” would create an object of the following type:

```
// The type of: _1 + 42
expr<tag::plus,
  args2<
    ref_<expr<tag::terminal, args0<S<1> > > const>,
    expr<tag::terminal, args0<int const&> >
  >
>
```

Proto holds all nodes and leaves in an expression tree by reference (`ref_<>` is a simple reference wrapper), so constructing expression trees is efficient. The types of even simple expressions can be large and complex, but expressions can be evaluated and transformed without ever having to state these types in code.

## 5.2 Expression Evaluation

The simplest way to evaluate Proto expressions is with the `eval()` function, which is similar to PETE’s `forEach()` algorithm. The behavior of `eval()` is customized by passing it a *context* object in addition to the expression object. Proto provides several context objects for common expression evaluation scenarios. The most useful is `default_context`, which gives the operators in the expression their usual C++ meaning. For instance, here is a “hello world” program fragment implemented in Proto that builds an output expression and then evaluates it using `default_context`:

```
terminal<ostream &>::type cout_ = {cout};

template<class Expr>
void print(Expr const & expr) {
    eval(expr, default_context());
}

int main() {
    print(cout_ << "hello world");
}
```

The invocation of `eval()` above causes the string “hello world” to be written to `cout`. Proto needs to know the types of intermediate results when evaluating expressions with `default_context`. For that, Proto uses the `Boost.Typeof` library [19], but in the future it will be able to use the proposed `decltype` keyword [10].

All context types are of the following form:

```
struct Context {
    template<class Expr>
    struct eval {
```

```
        typedef ... result_type;
        result_type operator()(
            Expr &expr, Context &ctx) const {
            return ...;
        }
    };
};
```

The `eval()` function instantiates the `Context`’s nested `eval<>` function object on the type of the expression node and passes the expression and context objects to it for further processing. Since the `eval<>` function object receives the unprocessed expression, it is free to evaluate it any way it sees fit, which makes short-circuited expressions possible. Typically, the `eval<>` struct is partially specialized to do different things with different operators. For instance, the `default_context` is implemented like this:

```
struct default_context {
    template<class E, class T=typename E::proto_tag>
    struct eval;

    template<typename Expr>
    struct eval<Expr, tag::plus> {
        typedef ... result_type;
        result_type operator()(Expr &expr,
            default_context const &ctx) const
        {return eval(left(expr), ctx)
            + eval(right(expr), ctx);}
    };
    ...
};
```

Proto’s `eval()` function shares some shortcomings with PETE’s `forEach()` algorithm. Neither supports evaluation across sub-domains with differing semantics. And although `eval()` supports alternate flow control (e.g., short-circuit evaluation), it still is not useful for performing complicated transformations such as folding trees into lists because it does not allow for accumulation of state. It is provided mainly for its simple interface. The section below on expression transformation shows a richer, more powerful interface that can be used for expression evaluation and more.

## 5.3 Expression Introspection

A shortcoming of expression template-based DSELs and template meta-programs in general is the impenetrable compiler errors they often cause. In the case of DSELs, the problem typically arises when users inadvertently create invalid expressions and pass them to the library for evaluation. The resulting failure typically occurs deep within the the library’s implementation, resulting in an instantiation backtrace sometimes measured in megabytes. Often, the user has no way of knowing what went wrong.

A better approach would be to validate expressions at API boundaries and fail early before attempting a deeply nested template instantiation. The result is shorter, more comprehensible errors. This is the purpose of Proto’s `matches<>` meta-function and its associated grammar-building utilities.

To use Spirit as an example, Spirit has a `rule` class that stores a rule of EBNF, which can be made up of parsers in sequence or in alternate. A typical rule definition might look like this:

```
// r recognizes a character followed by an 'a'
// or the string "hello"
rule r = anychar_p >> 'a' | "hello";
```

In this simplified example, assume that the only allowable operators in the Spirit DSEL are `>>` and `|`. An invalid expression such as `"anychar_p << 'a' | "hello"` (syntax error: `<<` should be `>>`) might result in a large and difficult to understand compiler error. However, if `rule` validates expressions against a grammar, it can issue a short, readable diagnostic. Appendix A contains a program that demonstrates the technique. The definition of the grammar is reproduced here:

```
struct SpiritGrammar
: or_<
    terminal<anychar_p_tag>,
    terminal<char>,
    terminal<char const*>,
    shift_right<SpiritGrammar,SpiritGrammar>,
    bitwise_or<SpiritGrammar,SpiritGrammar>
>
{};
```

The type `anychar_p_tag` is an empty type used in the definition of the `anychar_p` primitive, a syntactic element of Spirit's DSEL. This grammar reads as follows: A valid Spirit expression is

- an `anychar_p` primitive, *or*
- a character literal, *or*
- a string literal, *or*
- a right-shift expression where both operands are valid Spirit expressions, *or*
- a bitwise-or expression where both operands are valid Spirit expressions.

This is similar to defining the grammar of a language in BNF, except that no care is needed to get the precedence or associativity right.

Once the grammar of the DSEL is defined in code, `rule` can use the `matches<>` meta-function to see if a given expression type matches the grammar. The code in Appendix A uses the resulting compile-time Boolean to issue a readable diagnostic with a static assertion. It also uses the Boolean as a compile-time switch to avoid executing meta-code that is certain to fail, generating additional distracting errors. It does this by dispatching to an empty function.

In addition to grammar constructs such as `or_<>`, `shift_right<>`, and `terminal<>`, Proto provides the following:

- `and_<>`: Matches if all its child grammars match.
- `if_<>`: Tests a node type against a compile-time predicate and matches if the predicate evaluates to true.
- `not_<>`: Matches if a node type does *not* match a grammar.
- `switch_<>`: Like `or_<>` but performs a fast  $O(1)$  look-up based on a node's tag type, for improved compile times.

## 5.4 Expression Transformation

By attaching *transforms* to the rules in a grammar, the grammar can be used to transform expression trees into other types of objects. Proto provides a basic set of transforms and a way to extend the set. Below is a program that takes an expression involving the addition of floats and promotes the floats to doubles using a custom transform.

```
#include <boost/xpressive/proto/proto.hpp>
using namespace boost::proto;

template<typename Grammar>
struct to_double : Grammar {
    template<class E,class S,class V>
    struct apply : terminal<double> {};

    template<class E,class S,class V>
    static terminal<double>::type
    call(E const& e, S const& s, V& v) {
        terminal<double>::type d = {arg(e)};
        return d;
    }
};

struct Promote :
    or_< to_double< terminal<float> >,
        plus<Promote, Promote> >
{};

int main() {
    int x = 0;
    typedef terminal<double>::type Double;
    plus< Double, Double >::type p =
        Promote::call( lit(1.0f) + 3.14f, x, x );
}
```

The `Promote` structure defines a grammar consisting of float terminals and plus non-terminals. In addition, the float terminal rule is decorated with the `to_double<>` transform, which accepts float terminals and promotes them to doubles. In the above, `arg()` returns the first child of a node (or, in this case, the value of a terminal) and `lit()` turns a literal into a Proto terminal. `Promote::call()` executes the transform. The first parameter is the expression to transform, the second is a *state* parameter, and the third is a *visitor*. (Neither the state nor the visitor is used by this transform.)

The `to_double<>` transform is parameterized on a grammar – `terminal<float>`. The transform inherits from the

grammar, meaning that the transform is itself a grammar that matches float terminals. In addition to being a grammar, it is also a transform because it has a nested `apply<>` template for computing its return type and a static `call()` member function for performing the transformation. The `call()` member accepts the node to be transformed as well as the current state and visitor parameters (not used).

As can be inferred from the `Promote::call()` invocation, the `Promote` struct inherits a `call()` member function from `or_<>`, which, in addition to being a grammar construct, is also a transform. The `or_<>` transform invokes the transform of whichever alternate child grammar matches the expression. `plus<>` also has a transform; the transform `plus<Promote, Promote>` accepts plus expressions, applies the `Promote` transform to both the left and right children, and reassembles the transformed children into a new plus node. Just as grammars are recursive by parameterization on incomplete types, so too are transforms.

This transform is rather weak. A more useful transform might be to promote all floats to doubles in *any* expression tree, regardless of the type of the non-terminals or the number of their children. The following transform, which uses `nary_expr<>` and `vararg<>`, does just that.

```
struct Promote :
    or_< to_double< terminal<float> >,
        nary_expr<_, vararg<Promote> > >
{};
```

`nary_expr<_, vararg<Promote> >` says to match a node with any tag type (designated by the `_` wildcard), and zero or more `Promote` child nodes. In addition, the custom `to_double<>` transform can be replaced with Proto's `construct<>` transform, as follows.

```
struct Promote :
    or_<
        transform::construct<
            terminal<float>,
            terminal<double>::type(transform::arg<_>)>,
            nary_expr<_, vararg<Promote> > >
{};
```

Note that `terminal<double>::type(transform::arg<_>)` is a function type, but the `construct<>` transform interprets it as the type of object to construct and the arguments from which to construct it. Here, `_` is used as a placeholder that will receive the float terminal, and `transform::arg<>` is a transform that will extract the float from the terminal.

The preceding discussion has glossed over an interesting point: the many roles filled by the `plus<>` template. As `plus<Promote, Promote>`, `plus<>` behaves both as a grammar rule as well as a transform. But `main()` uses `plus<>` as a meta-function for calculating the type of a Proto expression. More generally, `plus< terminal<X>::type, terminal<X>::type >::type` is the type of an expression

and `plus< terminal<X>, terminal<X> >` is a grammar that matches that expression type and a transform that can manipulate it.

All transforms accept state and visitor parameters in addition to the node to be transformed. They can be anything, but some Proto transforms use the state parameter to accumulate a result. A good example is the `fold<>` transform, which transforms the first child node and uses the result as the state parameter when transforming the second child node, and so on. In this way, an  $N$ -ary node can be transformed into a heterogeneous list with  $N$  elements, for instance. There is also a `fold_tree<>` transform that, in addition to iterating over the child nodes, recurses into them to fold entire trees. It recurses into any child nodes that have a certain tag type, making it simple to turn trees like `(a | b | c)` into a list such as `cons<A, cons<B, cons<C, nil> > >`.

It is not too hard to see that anything that can be done with `eval()`, Proto's expression evaluation utility, can also be done with appropriately defined transforms. In fact, associating transforms with grammar rules rather than node types and giving them the ability to accumulate state make transforms more powerful than `eval()`.

One feature area not covered by PETE is the ability to evaluate an expression in one domain that contains sub-expressions from a different domain. An example would be an expression in an EBNF domain that contains a semantic action sub-expression in a lambda domain. Imagine a future version of the Boost Lambda Library, a lambda DSEL, that is built on top of Proto and publishes its grammar, decorated with transforms that evaluate the lambda expression. It might look like this:

```
struct LambdaGrammar
    : or_<...
        do_bitwise_or<
            bitwise_or<LambdaGrammar, LambdaGrammar>
        >, ...
    >
{};
```

where `do_bitwise_or<>` is a custom transform that performs a bitwise-or on the left and right child nodes. Then, imagine a future version of Spirit that also uses Proto grammars and transforms.<sup>5</sup> It might look like this:

```
struct SpiritGrammar
    : or_<...
        do_alternate<
            bitwise_or<SpiritGrammar, SpiritGrammar>
        >,
        do_action<
            subscript<SpiritGrammar, LambdaGrammar>
        >, ...
    >
{};
```

<sup>5</sup>These hypothetical future versions of the Lambda Library and Spirit are actually both under active development.

where `do_alternate<>` and `do_action<>` are also custom transforms that evaluate parser alternatives and execute semantic actions, respectively. This allows Spirit to use lambda expressions from the Lambda Library as semantic actions by putting them in square brackets `[]` after the rule with which they are associated. The key here is that both `LambdaGrammar` and `SpiritGrammar` define what `operator|` means within their domains, and expressions from these two domains can be freely mixed in the same expression—resulting in one big expression tree evaluable with a single pass—without any conflict or confusion about what `operator|` means.

## 5.5 Expression Extension

A library such as the Boost Lambda Library that simulates lambda expressions using expression templates presents unique challenges for a general DSEL framework like Proto. Lambda expressions should be function objects that are immediately executable by application of their `operator()` member function. For example, the expression `_1 + _2` should be a binary function object that adds its first argument to its second, but the following code does not compile.

```
terminal<mpl::int_<0> >::type _1 = {};
terminal<mpl::int_<1> >::type _2 = {};

int main()
{
    int in1[] = {1,2,3}, in2[] = {1,3,5}, out[3];
    // ERROR: "_1 + _2" doesn't have an
    // int operator(int, int) member.
    std::transform(in1, in1+3, in2, out, _1 + _2);
}
```

Proto handles *expression extension* with the `extends<>` template that lets you wrap Proto expressions and give them additional behaviors and members. The first step is to define a context for use with `eval()` that evaluates the lambda expression. `lambda_ctx<>`, defined below, uses Proto's `callable_context<>`. Any node types not handled explicitly by `lambda_ctx<>`, `callable_context<>` will dispatch to a default expression evaluator.

```
template<class Tuple> struct lambda_ctx :
    callable_context<lambda_ctx<Tuple> >
{
    Tuple args_;
    lambda_ctx(Tuple const& args) : args_(args) {}

    template<class Sig> struct result;
    template<class T, int I>
    struct result<T(tag::terminal,const int_<I>&)>
        : tuple_element<I, Tuple> {};

    template<int I>
    typename tuple_element<I, Tuple>::type
    operator()(tag::terminal,const int_<I>&) const
        {return get<I>(args_);}
};
```

The `lambda_ctx<>` is parameterized on a tuple containing the values to be substituted for the `_1` and `_2` placeholders. It handles the placeholder terminals explicitly. `callable_context<>` handles all other terminals and non-terminals automatically by dispatching to a `default_context`. When used with `lambda_ctx<>`, `eval()` will evaluate lambda expressions, as shown below.

```
tuple<int,int> args(3,5);
lambda_ctx<tuple<int,int> > ctx(args);
int j = eval(_1 + _2, ctx);
assert(j == 8);
```

The next step is to implement an expression wrapper that makes lambda expressions into function objects. The code is described below.

```
template<class E> struct lambda_expr;

struct lambda_domain :
    domain<generator<lambda_expr> > {};

template<class E> struct lambda_expr :
    extends<E,lambda_expr<E>,lambda_domain>
{
    lambda_expr(const E& e = E()) :
        extends<E,lambda_expr<E>,lambda_domain>(e)
    {}

    template<class A0, class A1>
    typename result_of::eval<E const,
        lambda_ctx<tuple<A0,A1> > >::type
    operator()(const A0& a0, const A1& a1) const
    {
        tuple<A0,A1> args(a0,a1);
        lambda_ctx<tuple<A0,A1> > ctx(args);
        return eval(*this, ctx);
    }
    // other operator() overloads, up to N args ...
};
```

The way `extends<>` works is to associate an expression with a *domain*, which in turn associates special domain-specific behaviors with the expression. The code above instructs Proto to wrap all expressions within the `lambda_domain` in a `lambda_expr<>` wrapper. `lambda_expr<E>` is an extension of the expression type `E`—it behaves just like an `E` except that it has an `operator()` overload that causes the expression to be evaluated with a `lambda_ctx<>`.

All that remains now is to put `_1` and `_2` in the lambda domain by wrapping them in `lambda_expr<>`. Any larger expressions involving `_1` and `_2` will also be in the lambda domain and will likewise be wrapped in `lambda_expr<>`.

```
lambda_expr<terminal<int_<0> >::type> _1;
lambda_expr<terminal<int_<1> >::type> _2;

int main()
```

```

{
  int in1[] = {1,2,3}, in2[] = {1,3,5}, out[3];
  std::transform(in1, in1+3, in2, out, _1 + _2);
  std::printf("%d,%d,%d\n",out[0],out[1],out[2]);
}

```

The above program works and displays “2,5,8”. The complete source code for the mini-lambda example is in Appendix B.

Proto domains have one other feature—they can be used to selectively disable some of Proto’s operator overloads. If the second template parameter to Proto’s `domain<>` template is a grammar, it will be used with `matches<>` to disable the operator overloads that would result in expressions that do not conform to the grammar. For instance, the following definition of `lambda_domain` disables Proto’s unary `operator&` overload in the lambda domain:

```

// Disable operator& in the lambda domain
struct lambda_domain :
  domain<generator<lambda_expr>,
    not_<address_of<_> > > {};

```

## 6. PROTO DESIGN EXAMPLES

Not all uses of an expression template library like Proto need to be as rich and complicated as POOMA, Spirit or the Lambda Library. Many library interfaces can be made more expressive with the judicious use of miniature DSELs. Proto can be helpful here, too. This section shows a couple of examples.

### 6.1 TR1 `regex_token_iterator<>`

The `regex_token_iterator<>` type from TR1 has a constructor that takes as one of its parameters either an array or vector of integers. A typical constructor call looks like this:

```

int subs[] = {1,2,3};
sregex_token_iterator iter(b,e,rx,subs);

```

where `b` and `e` are iterators designating a range of characters to search, `rx` is a regular expression object, and `subs` specifies which of the regex sub-matches are to be included in the tokenization of the string. The details of what `regex_token_iterator<>` does are not important, but the interface is. It forces users to declare a named variable `subs` on a separate line, even though it will only be used once, when constructing `iter`.

A mini-DSEL for specifying sub-matches can save users from having to declare a separate variable. One possible interface is shown below:

```

sregex_token_iterator iter(b,e,rx,(subs=1,2,3));

```

This interface only requires a `subs` terminal and an evaluation context that builds a vector of sub-match indices from the expression template. The following code demonstrates how.

```

struct subs_tag {};
terminal<subs_tag>::type subs = {{}};

struct push_back_ctx :
  callable_context<push_back_ctx, null_context>
{
  std::vector<int> subs;
  typedef void result_type;
  void operator()(tag::terminal, int sub)
    {subs.push_back(sub);}
};

```

Now all that remains is to define the appropriate `regex_token_iterator<>` constructor that accepts an expression template and evaluates it with `eval()` and `push_back_ctx<>`. After `eval()` returns, the `subs` member of the `push_back_ctx<>` will contain the sub-match indices.

In the Lambda Library example, `callable_context<>` dispatched all unhandled expression nodes to the `default_context`. Instead, `push_back_ctx` instructs `callable_context<>` to dispatch to `null_context`, another context type provided by Proto. The `null_context` evaluates all child nodes, but doesn’t combine the results in any way. The `default_context` is inappropriate in this case because it would try to execute the assignment to the `subs` placeholder.

### 6.2 Future Groups

Several standardization proposals related to threading make use of a type called `future<>` as a handle to a deferred calculation result. [6] [5] [21] In these proposals, a `future<int>` can be converted into an `int` by waiting for the calculation to finish. Hartmut Kaiser and Thorsten Schütt implemented such a type [13] and allowed `future<>`’s to be combined with `operator||` and `operator&&` for either waiting for one or all of the deferred calculations to complete. Herb Sutter suggested a similar interface [15], and Howard Hinnant elaborated on this design [8], where he suggested the following interface using tuples:

```

future<A> a1, a2;
future<B> b1, b2;
future<C> c;
future<tuple<A,B> > ab;

```

```

A t0 = a1.get();
tuple<A,B,C> t1 = (a1&&b1&&c).get();
tuple<A,C> t2 = ((a1||a2)&&c).get();
tuple<A,B,C> t3 = ((a1&&b1||a2&&b2)&&c).get();
tuple<tuple<A,B>,C> t4 = ((ab||ab)&&c).get();

```

The idea is that the expression `(a1&&b1).get()` blocks until both results are available and returns them in a `tuple<A,B>`. Waiting for either of two operations to complete is written as `(a1||a2).get()`, where `a1` and `a2` have the same type.

Getting the return type of the `get()` member function correct involves some complicated type calculations, for which Proto is ideally suited. Appendix C shows one possible solution. It defines the grammar of the future groups expres-

sions (terminals, logical-and expressions, and logical-or expressions) and how they should be transformed. It makes extensive use to Boost.Fusion [4] to perform type sequence calculations. The code handles future group expressions of arbitrary complexity, and yet the solution is only about 65 lines of code.

## 7. CONCLUSIONS

Domain-specific embedded languages, both small and large, are a powerful tool for increasing the expressiveness of C++ libraries. Proto is a toolkit for building DSELs that are rigorously defined in terms of grammars. This brings many benefits, including better diagnostics for invalid expressions and powerful expression evaluation and transformation abilities. Having a shared framework for expression template-based DSELs means that multiple DSELs can be mixed in the same expression without conflicts.

Proto is used by Boost.Xpressive [14] which uses it to build a regular expressions DSEL. It is the foundation for the next versions of Boost.Spirit, which uses Proto to build DSELs for lexing, parsing, and output generation; and Boost.Lambda.

## 8. ACKNOWLEDGEMENTS

Joel de Guzman and Hartmut Kaiser provided useful feedback during the early development of Proto.

## 9. REFERENCES

- [1] L. A. N. L. Advanced Computing Laboratory. PETE. <http://acts.nersc.gov/pete/>.
- [2] L. A. N. L. Advanced Computing Laboratory. POOMA. <http://www.nongnu.org/freepooma/>.
- [3] J. de Guzman and H. Kaiser. The Spirit Parser Framework. <http://spirit.sourceforge.net>.
- [4] J. de Guzman, D. Marsden, and T. Schwinger. Boost.Fusion. [http://spirit.sourceforge.net/dl\\_more/fusion\\_v2/libs/fusion/doc/html/index.html](http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion/doc/html/index.html).
- [5] P. Dimov. Proposed Text for Parallel Task Execution. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2185.html>.
- [6] P. Dimov. Transporting Values and Exceptions between Threads. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2096.html>.
- [7] P. Dimov, D. Gregor, J. Järvi, and G. Powell. A Proposal to Add an Enhanced Binder to the Library Technical Report. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1438.htm>.
- [8] H. Hinnant. future groups. [c++std-lib@accu.org](mailto:c++std-lib@accu.org), July 2007. [c++std-lib-19057](mailto:c++std-lib-19057).
- [9] J. Järvi and G. Powell. The Boost Lambda Library. <http://boost.org/doc/html/lambda.html>.
- [10] J. Järvi, B. Stroustrup, and G. Dos Reis. Decltype (revision 7): proposed wording. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2343.pdf>.
- [11] J. Järvi, J. Willcock, and A. Lumsdaine. Boost.Enable\_if. [http://boost.org/libs/utility/enable\\_if.html](http://boost.org/libs/utility/enable_if.html).
- [12] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

- [13] H. Kaiser and T. Schütt. futures based on boost.threads, 2005. <http://boost-consulting.com/vault/index.php?action=downloadfile&filename=futures.zip&directory=Concurrent%20Programming>.
- [14] E. Niebler. Boost.Xpressive. <http://boost.org/doc/html/xpressive.html>.
- [15] H. Sutter. Thread API interface tweaks. [cpp-threads@decadentplace.org.uk](mailto:cpp-threads@decadentplace.org.uk), August 2006.
- [16] T. Veldhuizen. Blitz++: Object Oriented Scientific Computing. <http://www.oonumerics.org/blitz/>.
- [17] T. Veldhuizen. Blitz++: The Library That Thinks It Is A Compiler. In *SciTools*, 1998. <http://www.oonumerics.org/blitz/blitztalk.ps>.
- [18] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
- [19] A. Vertleyb and P. Holt. Boost.Typeof. <http://boost.org/doc/html/typeof.html>.
- [20] M. Werle. Daixtrose, Differentiable Expression Templates. <http://daixtrose.sourceforge.net/index.html>.
- [21] A. Williams. Thread Pools and Futures. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2276.html>.

## APPENDIX

### A. MINI-SPIRIT EXAMPLE

Below is an example of using Proto grammars and `matches<>` to issue short, readable compile-time errors for malformed DSEL expressions. It uses a subset of Spirit's DSEL.

```
#include <boost/mpl/if.hpp>
#include <boost/mpl/bool.hpp>
#include <boost/mpl/assert.hpp>
#include <boost/xpressive/proto/proto.hpp>
using namespace boost;
using namespace proto;

struct anychar_p_tag {};
terminal<anychar_p_tag>::type anychar_p = {{{}};

struct SpiritGrammar
: or_<
    terminal<anychar_p_tag>,
    terminal<char>,
    terminal<char const*>,
    shift_right<SpiritGrammar,SpiritGrammar>,
    bitwise_or<SpiritGrammar,SpiritGrammar>
>
{};

struct VALID_RULE : mpl::true_ {};
struct INVALID_RULE : mpl::false_ {};

struct rule
{
    template<class Expr>
    rule &operator=(Expr const& expr) {
        typedef typename mpl::if_<
            matches<Expr, SpiritGrammar>,
```

```

        VALID_RULE, INVALID_RULE
    >::type IS_RULE_VALID;
    BOOST_MPL_ASSERT((IS_RULE_VALID));
    assign(expr, IS_RULE_VALID());
    return *this;
}

private:
    template<class Expr>
    void assign(Expr const& expr, VALID_RULE) {
        // Transform expr into a parser
        // with complicated meta-code.
        ...
    }
    template<class Expr>
    void assign(Expr const&, INVALID_RULE) {
        // no-op
    }
};

int main()
{
    rule r;
    r = anychar_p >> 'a' | "hello"; // OK
    r = anychar_p << 'a' | "hello"; // FAILS
    return 0;
}

```

On Visual C++ 8, the invalid rule causes the following compiler error:

```

main.cpp(32) : error C2664: 'boost::mpl::assertio
n_failed' : cannot convert parameter 1 from 'boos
t::mpl::failed *****INVALID_RULE:* *****
*****' to 'boost::mpl::assert<false>::type'
No constructor could take the source type, or con
structor overload resolution was ambiguous

```

The error message produced by gcc 3.4 below is equally short and comparatively readable:

```

main.cpp:32: error: conversion from 'mpl::failed
*****INVALID_RULE:*****' to non-sc
alar type 'mpl::assert< false>' requested
main.cpp:32: error: enumerator value for 'mpl_ass
ertion_in_line_32' not integer constant

```

Both errors are accompanied by only a single frame of template instantiation backtrace. With the addition of `static_assert` in C++0x, the error can be made even more readable.

## B. MINI-LAMBDA EXAMPLE

The following code demonstrates how to extend Proto expressions using the `extends<>` template. It implements a simple lambda library where lambda expressions are callable function objects.

```

#include <cstdio>

```

```

#include <algorithm>
#include <boost/mpl/int.hpp>
#include <boost/fusion/tuple.hpp>
#include <boost/xpressive/proto/proto.hpp>
#include <boost/xpressive/proto/context.hpp>
using namespace boost::proto;
using namespace boost::fusion;
using boost::mpl::int_;

template<class Tuple> struct lambda_ctx
    : callable_context<lambda_ctx<Tuple> >
{
    Tuple args_;
    lambda_ctx(Tuple const& args) : args_(args) {}

    template<class Sig> struct result;
    template<class T, int I>
    struct result<T(tag::terminal,const int_<I>&)>
        : tuple_element<I, Tuple> {};

    template<int I>
    typename tuple_element<I, Tuple>::type
    operator()(tag::terminal,const int_<I>&) const
        {return get<I>(args_);}
};

template<class E> struct lambda_expr;

struct lambda_domain :
    domain<generator<lambda_expr> > {};

template<class E> struct lambda_expr :
    extends<E,lambda_expr<E>,lambda_domain>
{
    lambda_expr(const E& e = E()) :
        extends<E,lambda_expr<E>,lambda_domain>(e)
    {}

    template<class A0, class A1>
    typename boost::proto::result_of::eval<E const,
        lambda_ctx<tuple<A0,A1> > >::type
    operator()(const A0& a0, const A1& a1) const
    {
        tuple<A0,A1> args(a0,a1);
        lambda_ctx<tuple<A0,A1> > ctx(args);
        return eval(*this, ctx);
    }
    // other operator() overloads, up to N args ...
};

lambda_expr<terminal<int_<0> >::type> _1;
lambda_expr<terminal<int_<1> >::type> _2;

int main()
{
    int in1[] = {1,2,3}, in2[] = {1,3,5}, out[3];
    // Use the lambda expression "_1 + _2" with
    // std::transform() to add two sequences.
    std::transform(in1, in1+3, in2, out, _1 + _2);
    std::printf("%d,%d,%d\n",out[0],out[1],out[2]);
}

```

## C. FUTURE GROUPS

Below is an mock-up of the future group suggestion made by Hartmut Kaiser and Thorsten Schütt [13] and Herb Sutter [15], and elaborated further by Howard Hinnant [8].

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/include/as_vector.hpp>
#include <boost/fusion/include/joint_view.hpp>
#include <boost/fusion/include/single_view.hpp>
#include <boost/xpressive/proto/proto.hpp>
#include <boost/xpressive/proto/transform.hpp>
using namespace boost;
using namespace proto;
namespace tfx = transform;

template<class L, class R>
struct pick_left
{
    BOOST_MPL_ASSERT((is_same<L, R>));
    typedef L type;
};

struct FutureGroup :
    or_<
        tfx::construct<
            terminal<_>,
            fusion::single_view<tfx::arg<_> >
                (tfx::arg<_>)
        >,
        tfx::construct<
            logical_and<FutureGroup, FutureGroup>,
            fusion::joint_view<
                add_const<tfx::left<_> >,
                add_const<tfx::right<_> >
            >(tfx::left<_>, tfx::right<_>)
        >,
        tfx::construct<
            logical_or<FutureGroup, FutureGroup>,
            pick_left<tfx::left<_>, tfx::right<_> >
                (tfx::left<_>)
        >
    >
{};

template<class E> struct future_expr;

struct future_dom :
    domain<generator<future_expr>, FutureGroup>
{};

template<class E>
struct future_expr :
    extends<E, future_expr<E>, future_dom>
{
    explicit future_expr(E const &e) :
        extends<E, future_expr<E>, future_dom>(e)
    {}

    typename fusion::result_of::as_vector<
        typename FutureGroup::apply<E, int, int>::type
    >::type
    get() const {
        int i = 0;
```

```
        return fusion::as_vector(
            FutureGroup::call(*this, i, i));
    }
};
```

```
template<class T>
struct future :
    future_expr<typename terminal<T>::type>
{
    future(T const &t = T()) :
        future_expr<typename terminal<T>::type>(
            terminal<T>::type::make(t))
    {}

    T get() const {return arg(*this);}
};
```

```
// TEST CASES
struct A {};
struct B {};
struct C {};
```

```
int main()
{
    using fusion::tuple;

    future<A> a;
    future<B> b;
    future<C> c;
    future<tuple<A,B> > ab;

    A t0 = a.get();
    tuple<A,B,C> t1 = (a&&b&&c).get();
    tuple<A,C> t2 = ((a||a)&&c).get();
    tuple<A,B,C> t3 = ((a&&b||a&&b)&&c).get();
    tuple<tuple<A,B>,C> t4 = ((ab||ab)&&c).get();
}
```