
Boost.Proto

Eric Niebler

Copyright © 2008 Eric Niebler

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Preface	3
Users' Guide	4
Getting Started	5
Installing Proto	5
Naming Conventions	5
Hello World	6
Hello Calculator	7
Fronts Ends: Defining Terminals and Non-Terminals of Your DSEL	11
Making Terminals	12
Proto's Operator Overloads	12
Making Lazy Functions	13
Adding Members by Extending Expressions	17
Domains	17
The <code>extends<></code> Expression Wrapper	18
Expression Generators	22
Controlling Operator Overloads	23
Adapting Existing Types to Proto	24
Generating Repetitive Code with the Preprocessor	25
Intermediate Form: Understanding and Introspecting Expressions	27
Accessing Parts of an Expression	30
Deep-copying Expressions	33
Debugging Expressions	33
Operator Tags and Metafunctions	34
Expressions as Fusion Sequences	36
Expression Introspection: Defining a Grammar	38
Finding Patterns in Expressions	38
Fuzzy and Exact Matches of Terminals	40
<code>if_<></code> , <code>and_<></code> , and <code>not_<></code>	42
Improving Compile Times With <code>switch_<></code>	42
Matching Vararg Expressions	46
Defining DSEL Grammars	47
Back Ends: Making Expression Templates Do Useful Work	49
Expression Evaluation: Imparting Behaviors with a Context	49
Evaluating an Expression with <code>proto::eval()</code>	50
Defining an Evaluation Context	51
Proto's Built-In Contexts	54
Expression Transformation: Semantic Actions	58
"Activating" Your Grammars	58
Handling Alternation and Recursion	60
Callable Transforms	61
Object Transforms	61
Example: Calculator Arity	62
Transforms With State Accumulation	66
Passing Auxiliary Data to Transforms	68

Proto's Built-In Transforms	72
Building Custom Primitive Transforms	78
Making Your Transform Callable	80
Examples	82
Hello World: Building an Expression Template and Evaluating It	82
Calc1: Defining an Evaluation Context	83
Calc2: Adding Members Using <code>proto::extends<></code>	85
Calc3: Defining a Simple Transform	87
Lazy Vector: Controlling Operator Overloads	89
RGB: Type Manipulations with Proto Transforms	92
TArray: A Simple Linear Algebra Library	94
Vec3: Computing With Transforms and Contexts	98
Vector: Adapting a Non-Proto Terminal Type	101
Mixed: Adapting Several Non-Proto Terminal Types	105
Map Assign: An Intermediate Transform	112
Future Group: A More Advanced Transform	114
Lambda: A Simple Lambda Library with Proto	117
Background and Resources	121
Glossary	121
Reference	123
Concepts	123
Classes	123
Functions	129
Header <code><boost/proto/args.hpp></code>	129
Header <code><boost/proto/core.hpp></code>	132
Header <code><boost/proto/debug.hpp></code>	132
Header <code><boost/proto/deep_copy.hpp></code>	138
Header <code><boost/proto/domain.hpp></code>	142
Header <code><boost/proto/eval.hpp></code>	150
Header <code><boost/proto/expr.hpp></code>	154
Header <code><boost/proto/extends.hpp></code>	161
Header <code><boost/proto/fusion.hpp></code>	174
Header <code><boost/proto/generate.hpp></code>	185
Header <code><boost/proto/literal.hpp></code>	197
Header <code><boost/proto/make_expr.hpp></code>	200
Header <code><boost/proto/matches.hpp></code>	210
Header <code><boost/proto/operators.hpp></code>	229
Header <code><boost/proto/proto.hpp></code>	236
Header <code><boost/proto/proto_fwd.hpp></code>	236
Header <code><boost/proto/proto_typeof.hpp></code>	243
Header <code><boost/proto/repeat.hpp></code>	244
Header <code><boost/proto/tags.hpp></code>	261
Header <code><boost/proto/traits.hpp></code>	307
Header <code><boost/proto/transform/arg.hpp></code>	441
Header <code><boost/proto/transform/call.hpp></code>	460
Header <code><boost/proto/transform/default.hpp></code>	464
Header <code><boost/proto/transform/fold.hpp></code>	468
Header <code><boost/proto/transform/fold_tree.hpp></code>	471
Header <code><boost/proto/transform/impl.hpp></code>	475
Header <code><boost/proto/transform/lazy.hpp></code>	481
Header <code><boost/proto/transform/make.hpp></code>	483
Header <code><boost/proto/transform/pass_through.hpp></code>	490
Header <code><boost/proto/transform/when.hpp></code>	493
Header <code><boost/proto/context/callable.hpp></code>	499
Header <code><boost/proto/context/default.hpp></code>	503
Header <code><boost/proto/context/null.hpp></code>	508
Appendices	520
Appendix A: Release Notes	520

Appendix B: History	523
Appendix C: Rationale	524
Static Initialization	524
Why Not Reuse MPL, Fusion, et cetera?	524
Appendix D: Implementation Notes	525
Quick-n-Dirty Type Categorization	525
Detecting the Arity of Function Objects	525
Appendix E: Acknowledgements	527

Preface

“There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy.”

-- William Shakespeare

Description

Proto is a framework for building Domain Specific Embedded Languages in C++. It provides tools for constructing, type-checking, transforming and executing *expression templates*¹. More specifically, Proto provides:

- An expression tree data structure.
- A mechanism for giving expressions additional behaviors and members.
- Operator overloads for building the tree from an expression.
- Utilities for defining the grammar to which an expression must conform.
- An extensible mechanism for immediately executing an expression template.
- An extensible set of tree transformations to apply to expression trees.

Motivation

Expression Templates are an advanced technique that C++ library developers use to define embedded mini-languages that target specific problem domains. The technique has been used to create efficient and easy-to-use libraries for linear algebra as well as to define C++ parser generators with a readable syntax. But developing such a library involves writing an inordinate amount of unreadable and unmaintainable template mumbo-jumbo. Boost.Proto eases the development of [domain-specific embedded languages \(DSELs\)](#). Use Proto to define the primitives of your mini-language and let Proto handle the operator overloading and the construction of the expression parse tree. Immediately evaluate the expression tree by passing it a function object. Or transform the expression tree by defining the grammar of your mini-language, decorated with an assortment of tree transforms provided by Proto or defined by you. Then use the grammar to give your users short and readable syntax errors for invalid expressions! No more mumbo-jumbo -- an expression template library developed with Proto is declarative and readable.

In short, Proto is a DSEL for defining DSELs.

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *italics*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in code font and its name is followed by `()` to indicate that it is a free function.

¹ See [Expression Templates](#)

- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by `<>` to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.



Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of Proto
#include <boost/proto/proto.hpp>

// Create some namespace aliases
namespace mpl = boost::mpl;
namespace fusion = boost::fusion;
namespace proto = boost::proto;

// Allow unqualified use of Proto's wildcard pattern
using proto::_;
```

Users' Guide

Compilers, Compiler Construction Toolkits, and Proto

Most compilers have front ends and back ends. The front end parses the text of an input program into some intermediate form like an abstract syntax tree, and the back end takes the intermediate form and generates an executable from it.

A library built with Proto is essentially a compiler for a domain-specific embedded language (DSEL). It also has a front end, an intermediate form, and a back end. The front end is comprised of the symbols (a.k.a., terminals), members, operators and functions that make up the user-visible aspects of the DSEL. The back end is made of evaluation contexts and transforms that give meaning and behavior to the expression templates generated by the front end. In between is the intermediate form: the expression template itself, which is an abstract syntax tree in a very real sense.

To build a library with Proto, you will first decide what your interface will be; that is, you'll design a programming language for your domain and build the front end with tools provided by Proto. Then you'll design the back end by writing evaluation contexts and/or transforms that accept expression templates and do interesting things with them.

This users' guide is organized as follows. After a [Getting Started guide](#), we'll cover the tools Proto provides for defining and manipulating the three major parts of a compiler:

Front Ends

How to define the aspects of your DSEL with which your users will interact directly.

Intermediate Form

What Proto expression templates look like, how to discover their structure and access their constituents.

Back Ends

How to define evaluation contexts and transforms that make expression templates do interesting things.

After that, you may be interested in seeing some [Examples](#) to get a better idea of how the pieces all fit together.

Getting Started

Installing Proto

Getting Proto

You can get Proto by downloading Boost (Proto is in version 1.37 and later), or by accessing Boost's SVN repository on SourceForge.net. Just go to <http://svn.boost.org/trac/boost/wiki/BoostSubversion> and follow the instructions there for anonymous SVN access.

Building with Proto

Proto is a header-only template library, which means you don't need to alter your build scripts or link to any separate lib file to use it. All you need to do is `#include <boost/proto/proto.hpp>`. Or, you might decide to just include the core of Proto (`#include <boost/proto/core.hpp>`) and whichever contexts and transforms you happen to use.

Requirements

Proto depends on Boost. You must use either Boost version 1.34.1 or higher, or the version in SVN trunk.

Supported Compilers

Currently, Boost.Proto is known to work on the following compilers:

- Visual C++ 7.1 and higher
- GNU C++ 3.4 and higher
- Intel on Linux 8.1 and higher
- Intel on Windows 9.1 and higher



Note

Please send any questions, comments and bug reports to eric <at> boostpro <dot> com.

Naming Conventions

Proto is a large library and probably quite unlike any library you've used before. Proto uses some consistent naming conventions to make it easier to navigate, and they're described below.

Functions

All of Proto's functions are defined in the `boost::proto` namespace. For example, there is a function called `value()` defined in `boost::proto` that accepts a terminal expression and returns the terminal's value.

Metafunctions

Proto defines *metafunctions* that correspond to each of Proto's free functions. The metafunctions are used to compute the functions' return types. All of Proto's metafunctions live in the `boost::proto::result_of` namespace and have the same name as the functions to which they correspond. For instance, there is a class template `boost::proto::result_of::value<>` that you can use to compute the return type of the `boost::proto::value()` function.

Function Objects

Proto defines *function object* equivalents of all of its free functions. (A function object is an instance of a class type that defines an `operator()` member function.) All of Proto's function object types are defined in the `boost::proto::functional` namespace

and have the same name as their corresponding free functions. For example, `boost::proto::functional::value` is a class that defines a function object that does the same thing as the `boost::proto::value()` free function.

Primitive Transforms

Proto also defines *primitive transforms* -- class types that can be used to compose larger transforms for manipulating expression trees. Many of Proto's free functions have corresponding primitive transforms. These live in the `boost::proto` namespace and their names have a leading underscore. For instance, the transform corresponding to the `value()` function is called `boost::proto::_value`.

The following table summarizes the discussion above:

Table 1. Proto Naming Conventions

Entity	Example
Free Function	<code>boost::proto::value()</code>
Metafunction	<code>boost::proto::result_of::value<></code>
Function Object	<code>boost::proto::functional::value</code>
Transform	<code>boost::proto::_value</code>

Hello World

Below is a very simple program that uses Proto to build an expression template and then execute it.

```
#include <iostream>
#include <boost/proto/proto.hpp>
#include <boost/typeof/std/ostream.hpp>
using namespace boost;

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

template< typename Expr >
void evaluate( Expr const & expr )
{
    proto::default_context ctx;
    proto::eval(expr, ctx);
}

int main()
{
    evaluate( cout_ << "hello" << ',' << " world" );
    return 0;
}
```

This program outputs the following:

```
hello, world
```

This program builds an object representing the output operation and passes it to an `evaluate()` function, which then executes it.

The basic idea of expression templates is to overload all the operators so that, rather than evaluating the expression immediately, they build a tree-like representation of the expression so that it can be evaluated later. For each operator in an expression, at least one operand must be Protofied in order for Proto's operator overloads to be found. In the expression ...


```
cout_ << "hello" << ', ' << " world"
```

... the Protofied sub-expression is `cout_`, which is the Proto-ification of `std::cout`. The presence of `cout_` "infects" the expression, and brings Proto's tree-building operator overloads into consideration. Any literals in the expression are then Protofied by wrapping them in a Proto terminal before they are combined into larger Proto expressions.

Once Proto's operator overloads have built the expression tree, the expression can be lazily evaluated later by walking the tree. That is what `proto::eval()` does. It is a general tree-walking expression evaluator, whose behavior is customizable via a *context* parameter. The use of `proto::default_context` assigns the standard meanings to the operators in the expression. (By using a different context, you could give the operators in your expressions different semantics. By default, Proto makes no assumptions about what operators actually *mean*.)

Proto Design Philosophy

Before we continue, let's use the above example to illustrate an important design principle of Proto's. The expression template created in the *hello world* example is totally general and abstract. It is not tied in any way to any particular domain or application, nor does it have any particular meaning or behavior on its own, until it is evaluated in a *context*. Expression templates are really just heterogeneous trees, which might mean something in one domain, and something else entirely in a different one.

As we'll see later, there is a way to create Proto expression trees that are *not* purely abstract, and that have meaning and behaviors independent of any context. There is also a way to control which operators are overloaded for your particular domain. But that is not the default behavior. We'll see later why the default is often a good thing.

Hello Calculator

"Hello, world" is nice, but it doesn't get you very far. Let's use Proto to build a DSEL (domain-specific embedded language) for a lazily-evaluated calculator. We'll see how to define the terminals in your mini-language, how to compose them into larger expressions, and how to define an evaluation context so that your expressions can do useful work. When we're done, we'll have a mini-language that will allow us to declare a lazily-evaluated arithmetic expression, such as $(_2 - _1) / _2 * 100$, where `_1` and `_2` are placeholders for values to be passed in when the expression is evaluated.

Defining Terminals

The first order of business is to define the placeholders `_1` and `_2`. For that, we'll use the `proto::terminal<>` metafunction.

```
// Define a placeholder type
template<int I>
struct placeholder
{
};

// Define the Protofied placeholder terminals
proto::terminal<placeholder<0> >::type const _1 = {{{}};
proto::terminal<placeholder<1> >::type const _2 = {{{}};
```

The initialization may look a little odd at first, but there is a good reason for doing things this way. The objects `_1` and `_2` above do not require run-time construction -- they are *statically initialized*, which means they are essentially initialized at compile time. See the [Static Initialization](#) section in the [Rationale](#) appendix for more information.

Constructing Expression Trees

Now that we have terminals, we can use Proto's operator overloads to combine these terminals into larger expressions. So, for instance, we can immediately say things like:

```
// This builds an expression template
(_2 - _1) / _2 * 100;
```

This creates an expression tree with a node for each operator. The type of the resulting object is large and complex, but we are not terribly interested in it right now.

So far, the object is just a tree representing the expression. It has no behavior. In particular, it is not yet a calculator. Below we'll see how to make it a calculator by defining an evaluation context.

Evaluating Expression Trees

No doubt you want your expression templates to actually *do* something. One approach is to define an *evaluation context*. The context is like a function object that associates behaviors with the node types in your expression tree. The following example should make it clear. It is explained below.

```
struct calculator_context
: proto::callable_context< calculator_context const >
{
    // Values to replace the placeholders
    std::vector<double> args;

    // Define the result type of the calculator.
    // (This makes the calculator_context "callable".)
    typedef double result_type;

    // Handle the placeholders:
    template<int I>
    double operator()(proto::tag::terminal, placeholder<I>) const
    {
        return this->args[I];
    }
};
```

In `calculator_context`, we specify how Proto should evaluate the placeholder terminals by defining the appropriate overloads of the function call operator. For any other nodes in the expression tree (e.g., arithmetic operations or non-placeholder terminals), Proto will evaluate the expression in the "default" way. For example, a binary plus node is evaluated by first evaluating the left and right operands and adding the results. Proto's default evaluator uses the [Boost.Typeof](#) library to compute return types.

Now that we have an evaluation context for our calculator, we can use it to evaluate our arithmetic expressions, as below:

```
calculator_context ctx;
ctx.args.push_back(45); // the value of _1 is 45
ctx.args.push_back(50); // the value of _2 is 50

// Create an arithmetic expression and immediately evaluate it
double d = proto::eval( (_2 - _1) / _2 * 100, ctx );

// This prints "10"
std::cout << d << std::endl;
```

Later, we'll see how to define more interesting evaluation contexts and expression transforms that give you total control over how your expressions are evaluated.

Customizing Expression Trees

Our calculator DSEL is already pretty useful, and for many DSEL scenarios, no more would be needed. But let's keep going. Imagine how much nicer it would be if all calculator expressions overloaded `operator()` so that they could be used as function objects. We can do that by creating a calculator *domain* and telling Proto that all expressions in the calculator domain have extra members. Here is how to define a calculator domain:


```
// Forward-declare an expression wrapper
template<typename Expr>
struct calculator;

// Define a calculator domain. Expression within
// the calculator domain will be wrapped in the
// calculator<> expression wrapper.
struct calculator_domain
: proto::domain< proto::generator<calculator> >
{
};
```

The calculator<> type will be an expression wrapper. It will behave just like the expression that it wraps, but it will have extra member functions that we will define. The calculator_domain is what informs Proto about our wrapper. It is used below in the definition of calculator<>. Read on for a description.

```
// Define a calculator expression wrapper. It behaves just like
// the expression it wraps, but with an extra operator() member
// function that evaluates the expression.
template<typename Expr>
struct calculator
: proto::extends<Expr, calculator<Expr>, calculator_domain>
{
    typedef
        proto::extends<Expr, calculator<Expr>, calculator_domain>
        base_type;

    calculator(Expr const &expr = Expr())
        : base_type(expr)
    {}

    typedef double result_type;

    // Overload operator() to invoke proto::eval() with
    // our calculator_context.
    double operator()(double a1 = 0, double a2 = 0) const
    {
        calculator_context ctx;
        ctx.args.push_back(a1);
        ctx.args.push_back(a2);

        return proto::eval(*this, ctx);
    }
};
```

The calculator<> struct is an expression *extension*. It uses proto::extends<> to effectively add additional members to an expression type. When composing larger expressions from smaller ones, Proto notes what domain the smaller expressions are in. The larger expression is in the same domain and is automatically wrapped in the domain's extension wrapper.

All that remains to be done is to put our placeholders in the calculator domain. We do that by wrapping them in our calculator<> wrapper, as below:

```
// Define the Protofied placeholder terminals, in the
// calculator domain.
calculator<proto::terminal<placeholder<0> >::type> const _1;
calculator<proto::terminal<placeholder<1> >::type> const _2;
```

Any larger expression that contain these placeholders will automatically be wrapped in the calculator<> wrapper and have our operator() overload. That means we can use them as function objects as follows.


```
double result = ((_2 - _1) / _2 * 100)(45.0, 50.0);
assert(result == (50.0 - 45.0) / 50.0 * 100);
```

Since calculator expressions are now valid function objects, we can use them with standard algorithms, as shown below:

```
double a1[4] = { 56, 84, 37, 69 };
double a2[4] = { 65, 120, 60, 70 };
double a3[4] = { 0 };

// Use std::transform() and a calculator expression
// to calculate percentages given two input sequences:
std::transform(a1, a1+4, a2, a3, (_2 - _1) / _2 * 100);
```

Now, let's use the calculator example to explore some other useful features of Proto.

Detecting Invalid Expressions

You may have noticed that you didn't have to define an overloaded `operator-()` or `operator/()` -- Proto defined them for you. In fact, Proto overloads *all* the operators for you, even though they may not mean anything in your domain-specific language. That means it may be possible to create expressions that are invalid in your domain. You can detect invalid expressions with Proto by defining the *grammar* of your domain-specific language.

For simplicity, assume that our calculator DSEL should only allow addition, subtraction, multiplication and division. Any expression involving any other operator is invalid. Using Proto, we can state this requirement by defining the grammar of the calculator DSEL. It looks as follows:

```
// Define the grammar of calculator expressions
struct calculator_grammar
: proto::or_<
    proto::plus< calculator_grammar, calculator_grammar >
    , proto::minus< calculator_grammar, calculator_grammar >
    , proto::multiplies< calculator_grammar, calculator_grammar >
    , proto::divides< calculator_grammar, calculator_grammar >
    , proto::terminal< proto::_ >
>
{};
```

You can read the above grammar as follows: an expression tree conforms to the calculator grammar if it is a binary plus, minus, multiplies or divides node, where both child nodes also conform to the calculator grammar; or if it is a terminal. In a Proto grammar, `proto::_` is a wildcard that matches any type, so `proto::terminal< proto::_ >` matches any terminal, whether it is a placeholder or a literal.



Note

This grammar is actually a little looser than we would like. Only placeholders and literals that are convertible to doubles are valid terminals. Later on we'll see how to express things like that in Proto grammars.

Once you have defined the grammar of your DSEL, you can use the `proto::matches<>` metafunction to check whether a given expression type conforms to the grammar. For instance, we might add the following to our `calculator::operator()` overload:


```

template<typename Expr>
struct calculator
: proto::extends< /* ... as before ... */ >
{
    /* ... */
    double operator()(double a1 = 0, double a2 = 0) const
    {
        // Check here that the expression we are about to
        // evaluate actually conforms to the calculator grammar.
        BOOST_MPL_ASSERT((proto::matches<Expr, calculator_grammar>));
        /* ... */
    }
};

```

The addition of the `BOOST_MPL_ASSERT()` line enforces at compile time that we only evaluate expressions that conform to the calculator DSEL's grammar. With Proto grammars, `proto::matches<>` and `BOOST_MPL_ASSERT()` it is very easy to give the users of your DSEL short and readable compile-time errors when they accidentally misuse your DSEL.



Note

`BOOST_MPL_ASSERT()` is part of the Boost Metaprogramming Library. To use it, just `#include <boost/mpl/assert.hpp>`.

Controlling Operator Overloads

Grammars and `proto::matches<>` make it possible to detect when a user has created an invalid expression and issue a compile-time error. But what if you want to prevent users from creating invalid expressions in the first place? By using grammars and domains together, you can disable any of Proto's operator overloads that would create an invalid expression. It is as simple as specifying the DSEL's grammar when you define the domain, as shown below:

```

// Define a calculator domain. Expression within
// the calculator domain will be wrapped in the
// calculator<> expression wrapper.
// NEW: Any operator overloads that would create an
//       expression that does not conform to the
//       calculator grammar is automatically disabled.
struct calculator_domain
: proto::domain< proto::generator<calculator>, calculator_grammar >
{
};

```

The only thing we changed is we added `calculator_grammar` as the second template parameter to the `proto::domain<>` template when defining `calculator_domain`. With this simple addition, we disable any of Proto's operator overloads that would create an invalid calculator expression.

... And Much More

Hopefully, this gives you an idea of what sorts of things Proto can do for you. But this only scratches the surface. The rest of this users' guide will describe all these features and others in more detail.

Happy metaprogramming!

Fronts Ends: Defining Terminals and Non-Terminals of Your DSEL

Here is the fun part: designing your own mini-programming language. In this section we'll talk about the nuts and bolts of designing a DSEL interface using Proto. We'll cover the definition of terminals and lazy functions that the users of your DSEL will get to program with. We'll also talk about Proto's expression template-building operator overloads, and about ways to add additional members to expressions within your domain.

Making Terminals

As we saw with the Calculator example from the Introduction, the simplest way to get a DSEL up and running is simply to define some terminals, as follows.

```
// Define a literal integer Proto expression.
proto::terminal<int>::type i = {0};

// This creates an expression template.
i + 1;
```

With some terminals and Proto's operator overloads, you can immediately start creating expression templates.

Defining terminals -- with aggregate initialization -- can be a little awkward at times. Proto provides an easier-to-use wrapper for literals that can be used to construct Prototyped terminal expressions. It's called `proto::literal<>`.

```
// Define a literal integer Proto expression.
proto::literal<int> i = 0;

// Proto literals are really just Proto terminal expressions.
// For example, this builds a Proto expression template:
i + 1;
```

There is also a `proto::lit()` function for constructing a `proto::literal<>` in-place. The above expression can simply be written as:

```
// proto::lit(0) creates an integer terminal expression
proto::lit(0) + 1;
```

Proto's Operator Overloads

Once we have some Proto terminals, expressions involving those terminals build expression trees for us. Proto defines overloads for each of C++'s overloadable operators in the `boost::proto` namespace. As long as one operand is a Proto expression, the result of the operation is a tree node representing that operation.



Note

Proto's operator overloads live in the `boost::proto` namespace and are found via ADL (argument-dependent lookup). That is why expressions must be "tainted" with Proto-ness for Proto to be able to build trees out of expressions.

As a result of Proto's operator overloads, we can say:

```
-_1;           // OK, build a unary-negate tree node
-_1 + 42;      // OK, build a binary-plus tree node
```

For the most part, this Just Works and you don't need to think about it, but a few operators are special and it can be helpful to know how Proto handles them.

Assignment, Subscript, and Function Call Operators

Proto also overloads `operator=`, `operator[]`, and `operator()`, but these operators are member functions of the expression template rather than free functions in Proto's namespace. The following are valid Proto expressions:


```

_1 = 5;      // OK, builds a binary assign tree node
_1[6];      // OK, builds a binary subscript tree node
_1();       // OK, builds a unary function tree node
_1(7);      // OK, builds a binary function tree node
_1(8,9);    // OK, builds a ternary function tree node
// ... etc.

```

For the first two lines, assignment and subscript, it should be fairly unsurprising that the resulting expression node should be binary. After all, there are two operands in each expression. It may be surprising at first that what appears to be a function call with no arguments, `_1()`, actually creates an expression node with one child. The child is `_1` itself. Likewise, the expression `_1(7)` has two children: `_1` and `7`.

Because these operators can only be defined as member functions, the following expressions are invalid:

```

int i;
i = _1;      // ERROR: cannot assign _1 to an int

int *p;
p[_1];      // ERROR: cannot use _1 as an index

std::sin(_1); // ERROR: cannot call std::sin() with _1

```

Also, C++ has special rules for overloads of `operator->` that make it useless for building expression templates, so Proto does not overload it.

The Address-Of Operator

Proto overloads the address-of operator for expression types, so that the following code creates a new unary address-of tree node:

```

&_1;      // OK, creates a unary address-of tree node

```

It does *not* return the address of the `_1` object. However, there is special code in Proto such that a unary address-of node is implicitly convertible to a pointer to its child. In other words, the following code works and does what you might expect, but not in the obvious way:

```

typedef
    proto::terminal< placeholder<0> >::type
    _1_type;

_1_type const _1 = {{{}}};
_1_type const * p = &_1; // OK, &_1 implicitly converted

```

Making Lazy Functions

If we limited ourselves to nothing but terminals and operator overloads, our domain-specific embedded languages wouldn't be very expressive. Imagine that we wanted to extend our calculator DSEL with a full suite of math functions like `sin()` and `pow()` that we could invoke lazily as follows.

```

// A calculator expression that takes one argument
// and takes the sine of it.
sin(_1);

```

We would like the above to create an expression template representing a function invocation. When that expression is evaluated, it should cause the function to be invoked. (At least, that's the meaning of function invocation we'd like the calculator DSEL to have.) You can define `sin` quite simply as follows.


```
// "sin" is a Proto terminal containing a function pointer
proto::terminal< double (*)(double) >::type const sin = {&std::sin};
```

In the above, we define `sin` as a Proto terminal containing a pointer to the `std::sin()` function. Now we can use `sin` as a lazy function. The `default_context` that we saw in the Introduction knows how to evaluate lazy functions. Consider the following:

```
double pi = 3.1415926535;
proto::default_context ctx;
// Create a lazy "sin" invocation and immediately evaluate it
std::cout << proto::eval( sin(pi/2), ctx ) << std::endl;
```

The above code prints out:

```
1
```

It is important to note that there is nothing special about terminals that contain function pointers. *Any* Proto expression has an overloaded function call operator. Consider:

```
// This compiles!
proto::lit(1)(2)(3,4)(5,6,7,8);
```

That may look strange at first. It creates an integer terminal with `proto::lit()`, and then invokes it like a function again and again. What does it mean? To be sure, the `default_context` wouldn't know what to do with it. The `default_context` only knows how to evaluate expressions that are sufficiently C++-like. In the case of function call expressions, the left hand side must evaluate to something that can be invoked: a pointer to a function, a reference to a function, or a TR1-style function object. That doesn't stop you from defining your own evaluation context that gives that expression a meaning. But more on that later.

Making Lazy Functions, Continued

Now, what if we wanted to add a `pow()` function to our calculator DSEL that users could invoke as follows?

```
// A calculator expression that takes one argument
// and raises it to the 2nd power
pow< 2 >(_1);
```

The simple technique described above of making `pow` a terminal containing a function pointer doesn't work here. If `pow` is an object, then the expression `pow< 2 >(_1)` is not valid C++. `pow` needs to be a real function template. But it must be an unusual function; it must return an expression template.

Before we can write the `pow()` function, we need a function object that wraps an invocation of `std::pow()`.

```
// Define a pow_fun function object
template<int Exp>
struct pow_fun
{
    typedef double result_type;
    double operator()(double d) const
    {
        return std::pow(d, Exp);
    }
};
```

Now, let's try to define a function template that returns an expression template. We'll use the `proto::function<>` metafunction to calculate the type of a Proto expression that represents a function call. It is analogous to `proto::terminal<>`. (We'll see a couple of different ways to solve this problem, and each will demonstrate another utility for defining Proto front-ends.)


```
// Define a lazy pow() function for the calculator DSEL.
// Can be used as: pow< 2 >(_1)
template<int Exp, typename Arg>
typename proto::function<
    typename proto::terminal<pow_fun<Exp> >::type
    , Arg const &
>::type const
pow(Arg const &arg)
{
    typedef
        typename proto::function<
            typename proto::terminal<pow_fun<Exp> >::type
            , Arg const &
        >::type
        result_type;

    result_type result = {{{}}, arg};
    return result;
}
```

In the code above, notice how the `proto::function<>` and `proto::terminal<>` metafunctions are used to calculate the return type: `pow()` returns an expression template representing a function call where the first child is the function to call and the second is the argument to the function. (Unfortunately, the same type calculation is repeated in the body of the function so that we can initialize a local variable of the correct type. We'll see in a moment how to avoid that.)



Note

As with `proto::function<>`, there are metafunctions corresponding to all of the overloadable C++ operators for calculating expression types.

With the above definition of the `pow()` function, we can create calculator expressions like the one below and evaluate them using the `calculator_context` we implemented in the Introduction.

```
// Initialize a calculator context
calculator_context ctx;
ctx.args.push_back(3); // let _1 be 3

// Create a calculator expression that takes one argument,
// adds one to it, and raises it to the 2nd power; and then
// immediately evaluate it using the calculator_context.
assert( 16 == proto::eval( pow<2>(_1 + 1), ctx ) );
```

Protofying Lazy Function Arguments

Above, we defined a `pow()` function template that returns an expression template representing a lazy function invocation. But if we tried to call it as below, we'll run into a problem.

```
// ERROR: pow() as defined above doesn't work when
// called with a non-Proto argument.
pow< 2 >( 4 );
```

Proto expressions can only have other Proto expressions as children. But if we look at `pow()`'s function signature, we can see that if we pass it a non-Proto object, it will try to make it a child.


```
template<int Exp, typename Arg>
typename proto::function<
    typename proto::terminal<pow_fun<Exp> >::type
    , Arg const & // <== ERROR! This may not be a Proto type!
>::type const
pow(Arg const &arg)
```

What we want is a way to make `Arg` into a Proto terminal if it is not a Proto expression already, and leave it alone if it is. For that, we can use `proto::as_child()`. The following implementation of the `pow()` function handles all argument types, expression templates or otherwise.

```
// Define a lazy pow() function for the calculator DSEL. Use
// proto::as_child() to Protofy the argument, but only if it
// is not a Proto expression type to begin with!
template<int Exp, typename Arg>
typename proto::function<
    typename proto::terminal<pow_fun<Exp> >::type
    , typename proto::result_of::as_child<Arg const>::type
>::type const
pow(Arg const &arg)
{
    typedef
        typename proto::function<
            typename proto::terminal<pow_fun<Exp> >::type
            , typename proto::result_of::as_child<Arg const>::type
        >::type
        result_type;

    result_type result = {{{}}, proto::as_child(arg)};
    return result;
}
```

Notice how we use the `proto::result_of::as_child<>` metafunction to calculate the return type, and the `proto::as_child()` function to actually normalize the argument.

Lazy Functions Made Simple With `make_expr()`

The versions of the `pow()` function we've seen above are rather verbose. In the return type calculation, you have to be very explicit about wrapping non-Proto types. Worse, you have to restate the return type calculation in the body of `pow()` itself. Proto provides a helper for building expression templates directly that handles these mundane details for you. It's called `proto::make_expr()`. We can redefine `pow()` with it as below.


```
// Define a lazy pow() function for the calculator DSEL.
// Can be used as: pow< 2 >(_1)
template<int Exp, typename Arg>
typename proto::result_of::make_expr<
    proto::tag::function // Tag type
    , pow_fun<Exp>        // First child (by value)
    , Arg const &         // Second child (by reference)
>::type const
pow(Arg const &arg)
{
    return proto::make_expr<proto::tag::function>(
        pow_fun<Exp>() // First child (by value)
        , boost::ref(arg) // Second child (by reference)
    );
}
```

There are some things to notice about the above code. We use `proto::result_of::make_expr<>` to calculate the return type. The first template parameter is the tag type for the expression node we're building -- in this case, `proto::tag::function`, which is the tag type Proto uses for function call expressions.

Subsequent template parameters to `proto::result_of::make_expr<>` represent children nodes. If a child type is not already a Proto expression, it is made into a terminal with `proto::as_child()`. A type such as `pow_fun<Exp>` results in terminal that is held by value, whereas a type like `Arg const &` (note the reference) indicates that the result should be held by reference.

In the function body is the runtime invocation of `proto::make_expr()`. It closely mirrors the return type calculation. `proto::make_expr()` requires you to specify the node's tag type as a template parameter. The arguments to the function become the node's children. When a child should be stored by value, nothing special needs to be done. When a child should be stored by reference, you must use the `boost::ref()` function to wrap the argument. Without this extra information, the `proto::make_expr()` function couldn't know whether to store a child by value or by reference.

Adding Members by Extending Expressions

In this section, we'll see how to associate Proto expressions with a *domain*, how to add members to expressions within a domain, and how to control which operators are overloaded in a domain.

Domains

In the [Hello Calculator](#) section, we looked into making calculator expressions directly usable as lambda expressions in calls to STL algorithms, as below:

```
double data[] = {1., 2., 3., 4.};

// Use the calculator DSEL to square each element ... HOW?
std::transform( data, data + 4, data, _1 * _1 );
```

The difficulty, if you recall, was that by default Proto expressions don't have interesting behaviors of their own. They're just trees. In particular, the expression `_1 * _1` won't have an `operator()` that takes a double and returns a double like `std::transform()` expects -- unless we give it one. To make this work, we needed to define an expression wrapper type that defined the `operator()` member function, and we needed to associate the wrapper with the calculator *domain*.

In Proto, the term *domain* refers to a type that associates expressions in that domain to an expression *generator*. The generator is just a function object that accepts an expression and does something to it, like wrapping it in an expression wrapper.

You can also use a domain to associate expressions with a grammar. When you specify a domain's grammar, Proto ensures that all the expressions it generates in that domain conform to the domain's grammar. It does that by disabling any operator overloads that would create invalid expressions.

The `extends<>` Expression Wrapper

The first step to giving your calculator expressions extra behaviors is to define a calculator domain. All expressions within the calculator domain will be imbued with calculator-ness, as we'll see.

```
// A type to be used as a domain tag (to be defined below)
struct calculator_domain;
```

We use this domain type when extending the `proto::expr<>` type, which we do with the `proto::extends<>` class template. Here is our expression wrapper, which imbues an expression with calculator-ness. It is described below.

```
// The calculator<> expression wrapper makes expressions
// function objects.
template< typename Expr >
struct calculator
: proto::extends< Expr, calculator< Expr >, calculator_domain >
{
    typedef
        proto::extends< Expr, calculator< Expr >, calculator_domain >
        base_type;

    calculator( Expr const &expr = Expr() )
    : base_type( expr )
    {}

    // This is usually needed because by default, the compiler-
    // generated assignment operator hides extends<>::operator=
    BOOST_PROTO_EXTENDS_USING_ASSIGN(calculator)

    typedef double result_type;

    // Hide base_type::operator() by defining our own which
    // evaluates the calculator expression with a calculator context.
    result_type operator()( double d1 = 0.0, double d2 = 0.0 ) const
    {
        // As defined in the Hello Calculator section.
        calculator_context ctx;

        // ctx.args is a vector<double> that holds the values
        // with which we replace the placeholders (e.g., _1 and _2)
        // in the expression.
        ctx.args.push_back( d1 ); // _1 gets the value of d1
        ctx.args.push_back( d2 ); // _2 gets the value of d2

        return proto::eval(*this, ctx ); // evaluate the expression
    }
};
```

We want calculator expressions to be function objects, so we have to define an `operator()` that takes and returns doubles. The `calculator<>` wrapper above does that with the help of the `proto::extends<>` template. The first template to `proto::extends<>` parameter is the expression type we are extending. The second is the type of the wrapped expression. The third parameter is the domain that this wrapper is associated with. A wrapper type like `calculator<>` that inherits from `proto::extends<>` behaves just like the expression type it has extended, with any additional behaviors you choose to give it.



Note

Why not just inherit from `proto::expr<>`?

You might be thinking that this expression extension business is unnecessarily complicated. After all, isn't this why C++ supports inheritance? Why can't `calculator<Expr>` just inherit from `Expr` directly? The reason is because `Expr`, which presumably is an instantiation of `proto::expr<>`, has expression template-building operator overloads that will be incorrect for derived types. They will store `*this` by reference to `proto::expr<>`, effectively slicing off any derived parts. `proto::extends<>` gives your derived types operator overloads that don't slice off your additional members.

Although not strictly necessary in this case, we bring `extends<>::operator=` into scope with the `BOOST_PROTO_EXTENDS_USING_ASSIGN()` macro. This is really only necessary if you want expressions like `_1 = 3` to create a lazily evaluated assignment. `proto::extends<>` defines the appropriate `operator=` for you, but the compiler-generated `calculator<>::operator=` will hide it unless you make it available with the macro.

Note that in the implementation of `calculator<>::operator()`, we evaluate the expression with the `calculator_context` we defined earlier. As we saw before, the context is what gives the operators their meaning. In the case of the calculator, the context is also what defines the meaning of the placeholder terminals.

Now that we have defined the `calculator<>` expression wrapper, we need to wrap the placeholders to imbue them with calculator-ness:

```
calculator< proto::terminal< placeholder<0> >::type > const _1;
calculator< proto::terminal< placeholder<1> >::type > const _2;
```

Retaining POD-ness with `BOOST_PROTO_EXTENDS()`

To use `proto::extends<>`, your extension type must derive from `proto::extends<>`. Unfortunately, that means that your extension type is no longer POD and its instances cannot be *statically initialized*. (See the [Static Initialization](#) section in the [Rationale](#) appendix for why this matters.) In particular, as defined above, the global placeholder objects `_1` and `_2` will need to be initialized at runtime, which could lead to subtle order of initialization bugs.

There is another way to make an expression extension that doesn't sacrifice POD-ness: the `BOOST_PROTO_EXTENDS()` macro. You can use it much like you use `proto::extends<>`. We can use `BOOST_PROTO_EXTENDS()` to keep `calculator<>` a POD and our placeholders statically initialized.

```
// The calculator<> expression wrapper makes expressions
// function objects.
template< typename Expr >
struct calculator
{
    // Use BOOST_PROTO_EXTENDS() instead of proto::extends<> to
    // make this type a Proto expression extension.
    BOOST_PROTO_EXTENDS(Expr, calculator<Expr>, calculator_domain)

    typedef double result_type;

    result_type operator()( double d1 = 0.0, double d2 = 0.0 ) const
    {
        /* ... as before ... */
    }
};
```

With the new `calculator<>` type, we can redefine our placeholders to be statically initialized:


```
calculator< proto::terminal< placeholder<0> >::type > const _1 = {{{{}}};  
calculator< proto::terminal< placeholder<1> >::type > const _2 = {{{{}}};
```

We need to make one additional small change to accommodate the POD-ness of our expression extension, which we'll describe below in the section on expression generators.

What does `BOOST_PROTO_EXTENDS()` do? It defines a data member of the expression type being extended; some nested typedefs that Proto requires; `operator=`, `operator[]` and `operator()` overloads for building expression templates; and a nested `result<>` template for calculating the return type of `operator()`. In this case, however, the `operator()` overloads and the `result<>` template are not needed because we are defining our own `operator()` in the `calculator<>` type. Proto provides additional macros for finer control over which member functions are defined. We could improve our `calculator<>` type as follows:

```
// The calculator<> expression wrapper makes expressions  
// function objects.  
template< typename Expr >  
struct calculator  
{  
    // Use BOOST_PROTO_BASIC_EXTENDS() instead of proto::extends<> to  
    // make this type a Proto expression extension:  
    BOOST_PROTO_BASIC_EXTENDS(Expr, calculator<Expr>, calculator_domain)  
  
    // Define operator[] to build expression templates:  
    BOOST_PROTO_EXTENDS_SUBSCRIPT()  
  
    // Define operator= to build expression templates:  
    BOOST_PROTO_EXTENDS_ASSIGN()  
  
    typedef double result_type;  
  
    result_type operator()( double d1 = 0.0, double d2 = 0.0 ) const  
    {  
        /* ... as before ... */  
    }  
};
```

Notice that we are now using `BOOST_PROTO_BASIC_EXTENDS()` instead of `BOOST_PROTO_EXTENDS()`. This just adds the data member and the nested typedefs but not any of the overloaded operators. Those are added separately with `BOOST_PROTO_EXTENDS_ASSIGN()` and `BOOST_PROTO_EXTENDS_SUBSCRIPT()`. We are leaving out the function call operator and the nested `result<>` template that could have been defined with Proto's `BOOST_PROTO_EXTENDS_FUNCTION()` macro.

In summary, here are the macros you can use to define expression extensions, and a brief description of each.

Table 2. Expression Extension Macros

Macro	Purpose
<pre>BOOST_PROTO_BASIC_EXTENDS(expression , extension , domain)</pre>	Defines a data member of type <i>expression</i> and some nested typedefs that Proto requires.
<code>BOOST_PROTO_EXTENDS_ASSIGN()</code>	Defines operator=. Only valid when preceded by <code>BOOST_PROTO_BASIC_EXTENDS()</code> .
<code>BOOST_PROTO_EXTENDS_SUBSCRIPT()</code>	Defines operator[]. Only valid when preceded by <code>BOOST_PROTO_BASIC_EXTENDS()</code> .
<code>BOOST_PROTO_EXTENDS_FUNCTION()</code>	Defines operator() and a nested <code>result<></code> template for return type calculation. Only valid when preceded by <code>BOOST_PROTO_BASIC_EXTENDS()</code> .
<pre>BOOST_PROTO_EXTENDS(expression , extension , domain)</pre>	<p>Equivalent to:</p> <pre>BOOST_PROTO_BASIC_EXTENDS(expression, extension, domain) BOOST_PROTO_EXTENDS_ASSIGN() BOOST_PROTO_EXTENDS_SUBSCRIPT() BOOST_PROTO_EXTENDS_FUNCTION()</pre>



Warning

Argument-Dependent Lookup and `BOOST_PROTO_EXTENDS()`

Proto's operator overloads are defined in the `boost::proto` namespace and are found by argument-dependent lookup (ADL). This usually just works because expressions are made up of types that live in the `boost::proto` namespace. However, sometimes when you use `BOOST_PROTO_EXTENDS()` that is not the case. Consider:

```
template<class T>
struct my_complex
{
    BOOST_PROTO_EXTENDS(
        typename proto::terminal<std::complex<T> >::type
        , my_complex<T>
        , proto::default_domain
    )
};

int main()
{
    my_complex<int> c0, c1;

    c0 + c1; // ERROR: operator+ not found
}
```

The problem has to do with how argument-dependent lookup works. The type `my_complex<int>` is not associated in any way with the `boost::proto` namespace, so the operators defined there are not considered. (Had we inherited from `proto::extends<>` instead of used `BOOST_PROTO_EXTENDS()`, we would have avoided the problem because inheriting from a type in `boost::proto` namespace is enough to get ADL to kick in.)

So what can we do? By adding an extra dummy template parameter that defaults to a type in the `boost::proto` namespace, we can trick ADL into finding the right operator overloads. The solution looks like this:

```
template<class T, class Dummy = proto::is_proto_expr>
struct my_complex
{
    BOOST_PROTO_EXTENDS(
        typename proto::terminal<std::complex<T> >::type
        , my_complex<T>
        , proto::default_domain
    )
};

int main()
{
    my_complex<int> c0, c1;

    c0 + c1; // OK, operator+ found now!
}
```

The type `proto::is_proto_expr` is nothing but an empty struct, but by making it a template parameter we make `boost::proto` an associated namespace of `my_complex<int>`. Now ADL can successfully find Proto's operator overloads.

Expression Generators

The last thing that remains to be done is to tell Proto that it needs to wrap all of our calculator expressions in our `calculator<>` wrapper. We have already wrapped the placeholders, but we want *all* expressions that involve the calculator placeholders to be calculators. We can do that by specifying an expression generator when we define our `calculator_domain`, as follows:


```
// Define the calculator_domain we forward-declared above.
// Specify that all expression in this domain should be wrapped
// in the calculator<> expression wrapper.
struct calculator_domain
: proto::domain< proto::generator< calculator > >
{};
```

The first template parameter to `proto::domain<>` is the generator. "Generator" is just a fancy name for a function object that accepts an expression and does something to it. `proto::generator<>` is a very simple one --- it wraps an expression in the wrapper you specify. `proto::domain<>` inherits from its generator parameter, so all domains are themselves function objects.

If we used `BOOST_PROTO_EXTENDS()` to keep our expression extension type POD, then we need to use `proto::pod_generator<>` instead of `proto::generator<>`, as follows:

```
// If calculator<> uses BOOST_PROTO_EXTENDS() instead of
// use proto::extends<>, use proto::pod_generator<> instead
// of proto::generator<>.
struct calculator_domain
: proto::domain< proto::pod_generator< calculator > >
{};
```

After Proto has calculated a new expression type, it checks the domains of the child expressions. They must match. Assuming they do, Proto creates the new expression and passes it to `Domain::operator()` for any additional processing. If we don't specify a generator, the new expression gets passed through unchanged. But since we've specified a generator above, `calculator_domain::operator()` returns `calculator<>` objects.

Now we can use calculator expressions as function objects to STL algorithms, as follows:

```
double data[] = {1., 2., 3., 4.};

// Use the calculator DSEL to square each element ... WORKS! :-)
std::transform( data, data + 4, data, _1 * _1 );
```

Controlling Operator Overloads

By default, Proto defines every possible operator overload for Protofied expressions. This makes it simple to bang together a DSEL. In some cases, however, the presence of Proto's promiscuous overloads can lead to confusion or worse. When that happens, you'll have to disable some of Proto's overloaded operators. That is done by defining the grammar for your domain and specifying it as the second parameter of the `proto::domain<>` template.

In the [Hello Calculator](#) section, we saw an example of a Proto grammar, which is repeated here:

```
// Define the grammar of calculator expressions
struct calculator_grammar
: proto::or_<
    proto::plus< calculator_grammar, calculator_grammar >
    , proto::minus< calculator_grammar, calculator_grammar >
    , proto::multiplies< calculator_grammar, calculator_grammar >
    , proto::divides< calculator_grammar, calculator_grammar >
    , proto::terminal< proto::_ >
>
{};
```

We'll have much more to say about grammars in subsequent sections, but for now, we'll just say that the `calculator_grammar` struct describes a subset of all expression types -- the subset that comprise valid calculator expressions. We would like to prohibit Proto from creating a calculator expression that does not conform to this grammar. We do that by changing the definition of the `calculator_domain` struct.


```
// Define the calculator_domain. Expressions in the calculator
// domain are wrapped in the calculator<> wrapper, and they must
// conform to the calculator_grammar:
struct calculator_domain
: proto::domain< proto::generator< calculator >, calculator_grammar >
{};
```

The only new addition is `calculator_grammar` as the second template parameter to the `proto::domain<>` template. That has the effect of disabling any of Proto's operator overloads that would create an invalid calculator expression.

Another common use for this feature would be to disable Proto's unary `operator&` overload. It may be surprising for users of your DSEL that they cannot take the address of their expressions! You can very easily disable Proto's unary `operator&` overload for your domain with a very simple grammar, as below:

```
// For expressions in my_domain, disable Proto's
// unary address-of operator.
struct my_domain
: proto::domain<
    proto::generator< my_wrapper >
    // A simple grammar that matches any expression that
    // is not a unary address-of expression.
    , proto::not_< proto::address_of< _ > >
>
{};
```

The type `proto::not_< proto::address_of< _ > >` is a very simple grammar that matches all expressions except unary address-of expressions. In the section describing Proto's intermediate form, we'll have much more to say about grammars.

Adapting Existing Types to Proto

The preceding discussions of defining Proto front ends have all made a big assumption: that you have the luxury of defining everything from scratch. What happens if you have existing types, say a matrix type and a vector type, that you would like to treat as if they were Proto terminals? Proto usually trades only in its own expression types, but with `BOOST_PROTO_DEFINE_OPERATORS()`, it can accommodate your custom terminal types, too.

Let's say, for instance, that you have the following types and that you can't modify them to make them “native” Proto terminal types.

```
namespace math
{
    // A matrix type ...
    struct matrix { /*...*/ };

    // A vector type ...
    struct vector { /*...*/ };
}
```

You can non-intrusively make objects of these types Proto terminals by defining the proper operator overloads using `BOOST_PROTO_DEFINE_OPERATORS()`. The basic procedure is as follows:

1. Define a trait that returns true for your types and false for all others.
2. Reopen the namespace of your types and use `BOOST_PROTO_DEFINE_OPERATORS()` to define a set of operator overloads, passing the name of the trait as the first macro parameter, and the name of a Proto domain (e.g., `proto::default_domain`) as the second.

The following code demonstrates how it works.


```

namespace math
{
    template<typename T>
    struct is_terminal
        : mpl::false_
    {};

    // OK, "matrix" is a custom terminal type
    template<>
    struct is_terminal<matrix>
        : mpl::true_
    {};

    // OK, "vector" is a custom terminal type
    template<>
    struct is_terminal<vector>
        : mpl::true_
    {};

    // Define all the operator overloads to construct Proto
    // expression templates, treating "matrix" and "vector"
    // objects as if they were Proto terminals.
    BOOST_PROTO_DEFINE_OPERATORS(is_terminal, proto::default_domain)
}

```

The invocation of the `BOOST_PROTO_DEFINE_OPERATORS()` macro defines a complete set of operator overloads that treat `matrix` and `vector` objects as if they were Proto terminals. And since the operators are defined in the same namespace as the `matrix` and `vector` types, the operators will be found by argument-dependent lookup. With the code above, we can now construct expression templates with matrices and vectors, as shown below.

```

math::matrix m1;
math::vector v1;
proto::literal<int> i(0);

m1 * 1; // custom terminal and literals are OK
m1 * i; // custom terminal and Proto expressions are OK
m1 * v1; // two custom terminals are OK, too.

```

Generating Repetitive Code with the Preprocessor

Sometimes as a DSEL designer, to make the lives of your users easy, you have to make your own life hard. Giving your users natural and flexible syntax often involves writing large numbers of repetitive function overloads. It can be enough to give you repetitive stress injury! Before you hurt yourself, check out the macros Proto provides for automating many repetitive code-generation chores.

Imagine that we are writing a lambda DSEL, and we would like to enable syntax for constructing temporary objects of any type using the following syntax:

```

// A lambda expression that takes two arguments and
// uses them to construct a temporary std::complex<>
construct< std::complex<int> >( _1, _2 )

```

For the sake of the discussion, imagine that we already have a function object template `construct_impl<>` that accepts arguments and constructs new objects from them. We would want the above lambda expression to be equivalent to the following:


```
// The above lambda expression should be roughly equivalent
// to the following:
proto::make_expr<proto::tag::function>(
    construct_impl<std::complex<int>>>() // The function to invoke lazily
    , boost::ref(_1)                     // The first argument to the function
    , boost::ref(_2)                     // The second argument to the function
);
```

We can define our `construct()` function template as follows:

```
template<typename T, typename A0, typename A1>
typename proto::result_of::make_expr<
    proto::tag::function
    , construct_impl<T>
    , A0 const &
    , A1 const &
>::type const
construct(A0 const &a0, A1 const &a1)
{
    return proto::make_expr<proto::tag::function>(
        construct_impl<T>()
        , boost::ref(a0)
        , boost::ref(a1)
    );
}
```

This works for two arguments, but we would like it to work for any number of arguments, up to (`BOOST_PROTO_MAX_ARITY` - 1). (Why "- 1"? Because one child is taken up by the `construct_impl<T>()` terminal leaving room for only (`BOOST_PROTO_MAX_ARITY` - 1) other children.)

For cases like this, Proto provides the `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_REPEAT_FROM_TO()` macros. To use it, we turn the function definition above into a macro as follows:

```
#define M0(N, typename_A, A_const_ref, A_const_ref_a, ref_a) \
template<typename T, typename_A(N)> \
typename proto::result_of::make_expr< \
    proto::tag::function \
    , construct_impl<T> \
    , A_const_ref(N) \
>::type const \
construct(A_const_ref_a(N)) \
{ \
    return proto::make_expr<proto::tag::function>( \
        construct_impl<T>() \
        , ref_a(N) \
    ); \
}
```

Notice that we turned the function into a macro that takes 5 arguments. The first is the current iteration number. The rest are the names of other macros that generate different sequences. For instance, Proto passes as the second parameter the name of a macro that will expand to `typename A0`, `typename A1`,

Now that we have turned our function into a macro, we can pass the macro to `BOOST_PROTO_REPEAT_FROM_TO()`. Proto will invoke it iteratively, generating all the function overloads for us.


```
// Generate overloads of construct() that accept from
// 1 to BOOST_PROTO_MAX_ARITY-1 arguments:
BOOST_PROTO_REPEAT_FROM_TO(1, BOOST_PROTO_MAX_ARITY, M0)
#undef M0
```

Non-Default Sequences

As mentioned above, Proto passes as the last 4 arguments to your macro the names of other macros that generate various sequences. The macros `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_REPEAT_FROM_TO()` select defaults for these parameters. If the defaults do not meet your needs, you can use `BOOST_PROTO_REPEAT_EX()` and `BOOST_PROTO_REPEAT_FROM_TO_EX()` and pass different macros that generate different sequences. Proto defines a number of such macros for use as parameters to `BOOST_PROTO_REPEAT_EX()` and `BOOST_PROTO_REPEAT_FROM_TO_EX()`. Check the reference section for `boost/proto/repeat.hpp` for all the details.

Also, check out `BOOST_PROTO_LOCAL_ITERATE()`. It works similarly to `BOOST_PROTO_REPEAT()` and friends, but it can be easier to use when you want to change one macro argument and accept defaults for the others.

Intermediate Form: Understanding and Introspecting Expressions

By now, you know a bit about how to build a front-end for your DSEL "compiler" -- you can define terminals and functions that generate expression templates. But we haven't said anything about the expression templates themselves. What do they look like? What can you do with them? In this section we'll see.

The `expr<>` Type

All Proto expressions are an instantiation of a template called `proto::expr<>` (or a wrapper around such an instantiation). When we define a terminal as below, we are really initializing an instance of the `proto::expr<>` template.

```
// Define a placeholder type
template<int I>
struct placeholder
{};

// Define the Protofied placeholder terminal
proto::terminal< placeholder<0> >::type const _1 = {{{}}};
```

The actual type of `_1` looks like this:

```
proto::expr< proto::tag::terminal, proto::term< placeholder<0> >, 0 >
```

The `proto::expr<>` template is the most important type in Proto. Although you will rarely need to deal with it directly, it's always there behind the scenes holding your expression trees together. In fact, `proto::expr<>` is the expression tree -- branches, leaves and all.

The `proto::expr<>` template makes up the nodes in expression trees. The first template parameter is the node type; in this case, `proto::tag::terminal`. That means that `_1` is a leaf-node in the expression tree. The second template parameter is a list of child types, or in the case of terminals, the terminal's value type. Terminals will always have only one type in the type list. The last parameter is the arity of the expression. Terminals have arity 0, unary expressions have arity 1, etc.

The `proto::expr<>` struct is defined as follows:


```
template< typename Tag, typename Args, long Arity = Args::arity >
struct expr;

template< typename Tag, typename Args >
struct expr< Tag, Args, 1 >
{
    typedef typename Args::child0 proto_child0;
    proto_child0 child0;
    // ...
};
```

The `proto::expr<>` struct does not define a constructor, or anything else that would prevent static initialization. All `proto::expr<>` objects are initialized using *aggregate initialization*, with curly braces. In our example, `_1` is initialized with the initializer `{ {} }`. The outer braces are the initializer for the `proto::expr<>` struct, and the inner braces are for the member `_1.child0` which is of type `placeholder<0>`. Note that we use braces to initialize `_1.child0` because `placeholder<0>` is also an aggregate.

Building Expression Trees

The `_1` node is an instantiation of `proto::expr<>`, and expressions containing `_1` are also instantiations of `proto::expr<>`. To use Proto effectively, you won't have to bother yourself with the actual types that Proto generates. These are details, but you're likely to encounter these types in compiler error messages, so it's helpful to be familiar with them. The types look like this:


```

// The type of the expression -_1
typedef
    proto::expr<
        proto::tag::negate
        , proto::list1<
            proto::expr<
                proto::tag::terminal
                , proto::term< placeholder<0> >
                , 0
            > const &
        >
        , 1
    >
    negate_placeholder_type;

negate_placeholder_type x = -_1;

// The type of the expression _1 + 42
typedef
    proto::expr<
        proto::tag::plus
        , proto::list2<
            proto::expr<
                proto::tag::terminal
                , proto::term< placeholder<0> >
                , 0
            > const &
        , proto::expr<
            proto::tag::terminal
            , proto::term< int const & >
            , 0
        >
        , 2
    >
    placeholder_plus_int_type;

placeholder_plus_int_type y = _1 + 42;

```

There are a few things to note about these types:

- Terminals have arity zero, unary expressions have arity one and binary expressions have arity two.
- When one Proto expression is made a child node of another Proto expression, it is held by reference, *even if it is a temporary object*. This last point becomes important later.
- Non-Proto expressions, such as the integer literal, are turned into Proto expressions by wrapping them in new `expr<>` terminal objects. These new wrappers are not themselves held by reference, but the object wrapped *is*. Notice that the type of the Protofied 42 literal is `int const &` -- held by reference.

The types make it clear: everything in a Proto expression tree is held by reference. That means that building an expression tree is exceptionally cheap. It involves no copying at all.



Note

An astute reader will notice that the object `y` defined above will be left holding a dangling reference to a temporary `int`. In the sorts of high-performance applications Proto addresses, it is typical to build and evaluate an expression tree before any temporary objects go out of scope, so this dangling reference situation often doesn't arise, but it is certainly something to be aware of. Proto provides utilities for deep-copying expression trees so they can be passed around as value types without concern for dangling references.

Accessing Parts of an Expression

After assembling an expression into a tree, you'll naturally want to be able to do the reverse, and access a node's children. You may even want to be able to iterate over the children with algorithms from the Boost.Fusion library. This section shows how.

Getting Expression Tags and Arities

Every node in an expression tree has both a *tag* type that describes the node, and an *arity* corresponding to the number of child nodes it has. You can use the `proto::tag_of<>` and `proto::arity_of<>` metafunctions to fetch them. Consider the following:

```
template<typename Expr>
void check_plus_node(Expr const &)
{
    // Assert that the tag type is proto::tag::plus
    BOOST_STATIC_ASSERT((
        boost::is_same<
            typename proto::tag_of<Expr>::type
            , proto::tag::plus
        >::value
    ));

    // Assert that the arity is 2
    BOOST_STATIC_ASSERT( proto::arity_of<Expr>::value == 2 );
}

// Create a binary plus node and use check_plus_node()
// to verify its tag type and arity:
check_plus_node( proto::lit(1) + 2 );
```

For a given type `Expr`, you could access the tag and arity directly as `Expr::proto_tag` and `Expr::proto_arity`, where `Expr::proto_arity` is an MPL Integral Constant.

Getting Terminal Values

There is no simpler expression than a terminal, and no more basic operation than extracting its value. As we've already seen, that is what `proto::value()` is for.

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

// Get the value of the cout_ terminal:
std::ostream & sout = proto::value( cout_ );

// Assert that we got back what we put in:
assert( &sout == &std::cout );
```

To compute the return type of the `proto::value()` function, you can use `proto::result_of::value<>`. When the parameter to `proto::result_of::value<>` is a non-reference type, the result type of the metafunction is the type of the value as suitable for storage by value; that is, top-level reference and qualifiers are stripped from it. But when instantiated with a reference type, the result type has a reference *added* to it, yielding a type suitable for storage by reference. If you want to know the actual type of the terminal's value including whether it is stored by value or reference, you can use `fusion::result_of::value_at<Expr, 0>::type`.

The following table summarizes the above paragraph.

Table 3. Accessing Value Types

Metafunction Invocation	When the Value Type Is ...	The Result Is ...
<code>proto::result_of::value<Expr>::type</code>	T	<pre>typename boost::remove_const< typename boost::remove_referen- ce<T>::type >::type ^a</pre>
<code>proto::result_of::value<Expr &>::type</code>	T	<pre>typename boost::add_reference<T>::type</pre>
<code>proto::result_of::value<Expr const &>::type</code>	T	<pre>typename boost::add_reference< typename boost::add_const<T>::type >::type</pre>
<code>fusion::result_of::value_at<Expr, 0>::type</code>	T	T

^aIf T is a reference-to-function type, then the result type is simply T.

Getting Child Expressions

Each non-terminal node in an expression tree corresponds to an operator in an expression, and the children correspond to the operands, or arguments of the operator. To access them, you can use the `proto::child_c()` function template, as demonstrated below:

```
proto::terminal<int>::type i = {42};

// Get the 0-th operand of an addition operation:
proto::terminal<int>::type &ri = proto::child_c<0>( i + 2 );

// Assert that we got back what we put in:
assert( &i == &ri );
```

You can use the `proto::result_of::child_c<>` metafunction to get the type of the Nth child of an expression node. Usually you don't care to know whether a child is stored by value or by reference, so when you ask for the type of the Nth child of an expression Expr (where Expr is not a reference type), you get the child's type after references and cv-qualifiers have been stripped from it.


```

template<typename Expr>
void test_result_of_child_c(Expr const &expr)
{
    typedef typename proto::result_of::child_c<Expr, 0>::type type;

    // Since Expr is not a reference type,
    // result_of::child_c<Expr, 0>::type is a
    // non-cv qualified, non-reference type:
    BOOST_MPL_ASSERT((
        boost::is_same< type, proto::terminal<int>::type >
    ));
}

// ...
proto::terminal<int>::type i = {42};
test_result_of_child_c( i + 2 );

```

However, if you ask for the type of the Nth child of `Expr` & or `Expr const` & (note the reference), the result type will be a reference, regardless of whether the child is actually stored by reference or not. If you need to know exactly how the child is stored in the node, whether by reference or by value, you can use `fusion::result_of::value_at<Expr, N>::type`. The following table summarizes the behavior of the `proto::result_of::child_c<>` metafunction.

Table 4. Accessing Child Types

Metafunction Invocation	When the Child Is ...	The Result Is ...
<code>proto::result_of::child_c<Expr, N>::type</code>	T	<pre> typename boost::remove_const< typename boost::remove_referer ence<T>::type >::type </pre>
<code>proto::result_of::child_c<Expr &, N>::type</code>	T	<pre> typename boost::add_reference<T>::type </pre>
<code>proto::result_of::child_c<Expr const &, N>::type</code>	T	<pre> typename boost::add_reference< typename boost::add_const<T>::type >::type </pre>
<code>fusion::result_of::value_at<Expr, N>::type</code>	T	T

Common Shortcuts

Most operators in C++ are unary or binary, so accessing the only operand, or the left and right operands, are very common operations. For this reason, Proto provides the `proto::child()`, `proto::left()`, and `proto::right()` functions. `proto::child()` and `proto::left()` are synonymous with `proto::child_c<0>()`, and `proto::right()` is synonymous with `proto::child_c<1>()`.

There are also `proto::result_of::child<>`, `proto::result_of::left<>`, and `proto::result_of::right<>` metafunctions that merely forward to their `proto::result_of::child_c<>` counterparts.

Deep-copying Expressions

When you build an expression template with Proto, all the intermediate child nodes are held *by reference*. This avoids needless copies, which is crucial if you want your DSEL to perform well at runtime. Naturally, there is a danger if the temporary objects go out of scope before you try to evaluate your expression template. This is especially a problem in C++0x with the new `decltype` and `auto` keywords. Consider:

```
// OOPS: "ex" is left holding dangling references
auto ex = proto::lit(1) + 2;
```

The problem can happen in today's C++ also if you use `BOOST_TYPEOF()` or `BOOST_AUTO()`, or if you try to pass an expression template outside the scope of its constituents.

In these cases, you want to deep-copy your expression template so that all intermediate nodes and the terminals are held *by value*. That way, you can safely assign the expression template to a local variable or return it from a function without worrying about dangling references. You can do this with `proto::deep_copy()` as follows:

```
// OK, "ex" has no dangling references
auto ex = proto::deep_copy( proto::lit(1) + 2 );
```

If you are using `Boost.Typeof`, it would look like this:

```
// OK, use BOOST_AUTO() and proto::deep_copy() to
// store an expression template in a local variable
BOOST_AUTO( ex, proto::deep_copy( proto::lit(1) + 2 ) );
```

For the above code to work, you must include the `boost/proto/proto_typeof.hpp` header, which also defines the `BOOST_PROTO_AUTO()` macro which automatically deep-copies its argument. With `BOOST_PROTO_AUTO()`, the above code can be written as:

```
// OK, BOOST_PROTO_AUTO() automatically deep-copies
// its argument:
BOOST_PROTO_AUTO( ex, proto::lit(1) + 2 );
```

When deep-copying an expression tree, all intermediate nodes and all terminals are stored by value. The only exception is terminals that are function references, which are left alone.



Note

`proto::deep_copy()` makes no exception for arrays, which it stores by value. That can potentially cause a large amount of data to be copied.

Debugging Expressions

Proto provides a utility for pretty-printing expression trees that comes in very handy when you're trying to debug your DSEL. It's called `proto::display_expr()`, and you pass it the expression to print and optionally, an `std::ostream` to which to send the output. Consider:

```
// Use display_expr() to pretty-print an expression tree
proto::display_expr(
    proto::lit("hello") + 42
);
```

The above code writes this to `std::cout`:


```
plus(  
    terminal(hello)  
    , terminal(42)  
)
```

In order to call `proto::display_expr()`, all the terminals in the expression must be Streamable (that is, they can be written to a `std::ostream`). In addition, the tag types must all be Streamable as well. Here is an example that includes a custom terminal type and a custom tag:

```
// A custom tag type that is Streamable  
struct MyTag  
{  
    friend std::ostream &operator<<(std::ostream &s, MyTag)  
    {  
        return s << "MyTag";  
    }  
};  
  
// Some other Streamable type  
struct MyTerminal  
{  
    friend std::ostream &operator<<(std::ostream &s, MyTerminal)  
    {  
        return s << "MyTerminal";  
    }  
};  
  
int main()  
{  
    // Display an expression tree that contains a custom  
    // tag and a user-defined type in a terminal  
    proto::display_expr(  
        proto::make_expr<MyTag>(MyTerminal()) + 42  
    );  
}
```

The above code prints the following:

```
plus(  
    MyTag(  
        terminal(MyTerminal)  
    )  
    , terminal(42)  
)
```

Operator Tags and Metafunctions

The following table lists the overloadable C++ operators, the Proto tag types for each, and the name of the metafunctions for generating the corresponding Proto expression types. And as we'll see later, the metafunctions are also usable as grammars for matching such nodes, as well as pass-through transforms.

Table 5. Operators, Tags and Metafunctions

Operator	Proto Tag	Proto Metafunction
unary +	proto::tag::unary_plus	proto::unary_plus<>
unary -	proto::tag::negate	proto::negate<>
unary *	proto::tag::dereference	proto::dereference<>
unary ~	proto::tag::complement	proto::complement<>
unary &	proto::tag::address_of	proto::address_of<>
unary !	proto::tag::logical_not	proto::logical_not<>
unary prefix ++	proto::tag::pre_inc	proto::pre_inc<>
unary prefix --	proto::tag::pre_dec	proto::pre_dec<>
unary postfix ++	proto::tag::post_inc	proto::post_inc<>
unary postfix --	proto::tag::post_dec	proto::post_dec<>
binary <<	proto::tag::shift_left	proto::shift_left<>
binary >>	proto::tag::shift_right	proto::shift_right<>
binary *	proto::tag::multiplies	proto::multiplies<>
binary /	proto::tag::divides	proto::divides<>
binary %	proto::tag::modulus	proto::modulus<>
binary +	proto::tag::plus	proto::plus<>
binary -	proto::tag::minus	proto::minus<>
binary <	proto::tag::less	proto::less<>
binary >	proto::tag::greater	proto::greater<>
binary <=	proto::tag::less_equal	proto::less_equal<>
binary >=	proto::tag::greater_equal	proto::greater_equal<>
binary ==	proto::tag::equal_to	proto::equal_to<>
binary !=	proto::tag::not_equal_to	proto::not_equal_to<>
binary	proto::tag::logical_or	proto::logical_or<>
binary &&	proto::tag::logical_and	proto::logical_and<>
binary &	proto::tag::bitwise_and	proto::bitwise_and<>
binary	proto::tag::bitwise_or	proto::bitwise_or<>

Operator	Proto Tag	Proto Metafunction
binary ^	proto::tag::bitwise_xor	proto::bitwise_xor<>
binary ,	proto::tag::comma	proto::comma<>
binary ->*	proto::tag::mem_ptr	proto::mem_ptr<>
binary =	proto::tag::assign	proto::assign<>
binary <=<	proto::tag::shift_left_assign	proto::shift_left_assign<>
binary >>=	proto::tag::shift_right_assign	proto::shift_right_assign<>
binary *=	proto::tag::multiplies_assign	proto::multiplies_assign<>
binary /=	proto::tag::divides_assign	proto::divides_assign<>
binary %=	proto::tag::modulus_assign	proto::modulus_assign<>
binary +=	proto::tag::plus_assign	proto::plus_assign<>
binary -=	proto::tag::minus_assign	proto::minus_assign<>
binary &=	proto::tag::bitwise_and_assign	proto::bitwise_and_assign<>
binary =	proto::tag::bitwise_or_assign	proto::bitwise_or_assign<>
binary ^=	proto::tag::bitwise_xor_assign	proto::bitwise_xor_assign<>
binary subscript	proto::tag::subscript	proto::subscript<>
ternary ?:	proto::tag::if_else_	proto::if_else_<>
n-ary function call	proto::tag::function	proto::function<>

Expressions as Fusion Sequences

Boost.Fusion is a library of iterators, algorithms, containers and adaptors for manipulating heterogeneous sequences. In essence, a Proto expression is just a heterogeneous sequence of its child expressions, and so Proto expressions are valid Fusion random-access sequences. That means you can apply Fusion algorithms to them, transform them, apply Fusion filters and views to them, and access their elements using `fusion::at()`. The things Fusion can do to heterogeneous sequences are beyond the scope of this users' guide, but below is a simple example. It takes a lazy function invocation like `fun(1, 2, 3, 4)` and uses Fusion to print the function arguments in order.


```

struct display
{
    template<typename T>
    void operator()(T const &t) const
    {
        std::cout << t << std::endl;
    }
};

struct fun_t {};
proto::terminal<fun_t>::type const fun = {{{}};

// ...
fusion::for_each(
    fusion::transform(
        // pop_front() removes the "fun" child
        fusion::pop_front(fun(1,2,3,4))
        // Extract the ints from the terminal nodes
        , proto::functional::value()
    )
    , display()
);

```

Recall from the Introduction that types in the `proto::functional` namespace define function objects that correspond to Proto's free functions. So `proto::functional::value()` creates a function object that is equivalent to the `proto::value()` function. The above invocation of `fusion::for_each()` displays the following:

```

1
2
3
4

```

Terminals are also valid Fusion sequences. They contain exactly one element: their value.

Flattening Proto Expression Tress

Imagine a slight variation of the above example where, instead of iterating over the arguments of a lazy function invocation, we would like to iterate over the terminals in an addition expression:

```

proto::terminal<int>::type const _1 = {1};

// ERROR: this doesn't work! Why?
fusion::for_each(
    fusion::transform(
        _1 + 2 + 3 + 4
        , proto::functional::value()
    )
    , display()
);

```

The reason this doesn't work is because the expression `_1 + 2 + 3 + 4` does not describe a flat sequence of terminals --- it describes a binary tree. We can treat it as a flat sequence of terminals, however, using Proto's `proto::flatten()` function. `proto::flatten()` returns a view which makes a tree appear as a flat Fusion sequence. If the top-most node has a tag type `T`, then the elements of the flattened sequence are the child nodes that do *not* have tag type `T`. This process is evaluated recursively. So the above can correctly be written as:


```

proto::terminal<int>::type const _1 = {1};

// OK, iterate over a flattened view
fusion::for_each(
    fusion::transform(
        proto::flatten(_1 + 2 + 3 + 4)
        , proto::functional::value()
    )
    , display()
);

```

The above invocation of `fusion::for_each()` displays the following:

```

1
2
3
4

```

Expression Introspection: Defining a Grammar

Expression trees can have a very rich and complicated structure. Often, you need to know some things about an expression's structure before you can process it. This section describes the tools Proto provides for peering inside an expression tree and discovering its structure. And as you'll see in later sections, all the really interesting things you can do with Proto begin right here.

Finding Patterns in Expressions

Imagine your DSEL is a miniature I/O facility, with `istream` operations that execute lazily. You might want expressions representing input operations to be processed by one function, and output operations to be processed by a different function. How would you do that?

The answer is to write patterns (a.k.a, *grammars*) that match the structure of input and output expressions. Proto provides utilities for defining the grammars, and the `proto::matches<>` template for checking whether a given expression type matches the grammar.

First, let's define some terminals we can use in our lazy I/O expressions:

```

proto::terminal< std::istream & >::type cin_ = { std::cin };
proto::terminal< std::ostream & >::type cout_ = { std::cout };

```

Now, we can use `cout_` instead of `std::cout`, and get I/O expression trees that we can execute later. To define grammars that match input and output expressions of the form `cin_ >> i` and `cout_ << 1` we do this:

```

struct Input
: proto::shift_right< proto::terminal< std::istream & >, proto::_ >
{};

struct Output
: proto::shift_left< proto::terminal< std::ostream & >, proto::_ >
{};

```

We've seen the template `proto::terminal<>` before, but here we're using it without accessing the nested `::type`. When used like this, it is a very simple grammar, as are `proto::shift_right<>` and `proto::shift_left<>`. The newcomer here is `_` in the `proto` namespace. It is a wildcard that matches anything. The `Input` struct is a grammar that matches any right-shift expression that has a `std::istream` terminal as its left operand.

We can use these grammars together with the `proto::matches<>` template to query at compile time whether a given I/O expression type is an input or output operation. Consider the following:


```

template< typename Expr >
void input_output( Expr const & expr )
{
    if( proto::matches< Expr, Input >::value )
    {
        std::cout << "Input!\n";
    }

    if( proto::matches< Expr, Output >::value )
    {
        std::cout << "Output!\n";
    }
}

int main()
{
    int i = 0;
    input_output( cout_ << 1 );
    input_output( cin_ >> i );

    return 0;
}

```

This program prints the following:

```

Output!
Input!

```

If we wanted to break the `input_output()` function into two functions, one that handles input expressions and one for output expressions, we can use `boost::enable_if<>`, as follows:

```

template< typename Expr >
typename boost::enable_if< proto::matches< Expr, Input >::type
input_output( Expr const & expr )
{
    std::cout << "Input!\n";
}

template< typename Expr >
typename boost::enable_if< proto::matches< Expr, Output >::type
input_output( Expr const & expr )
{
    std::cout << "Output!\n";
}

```

This works as the previous version did. However, the following does not compile at all:

```

input_output( cout_ << 1 << 2 ); // oops!

```

What's wrong? The problem is that this expression does not match our grammar. The expression groups as if it were written like `(cout_ << 1) << 2`. It will not match the Output grammar, which expects the left operand to be a terminal, not another left-shift operation. We need to fix the grammar.

We notice that in order to verify an expression as input or output, we'll need to recurse down to the bottom-left-most leaf and check that it is a `std::istream` or `std::ostream`. When we get to the terminal, we must stop recursing. We can express this in our grammar using `proto::or_<>`. Here are the correct Input and Output grammars:


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

struct Output
: proto::or_<
    proto::shift_left< proto::terminal< std::ostream & >, proto::_ >
    , proto::shift_left< Output, proto::_ >
>
{};
```

This may look a little odd at first. We seem to be defining the `Input` and `Output` types in terms of themselves. This is perfectly OK, actually. At the point in the grammar that the `Input` and `Output` types are being used, they are *incomplete*, but by the time we actually evaluate the grammar with `proto::matches<>`, the types will be complete. These are recursive grammars, and rightly so because they must match a recursive data structure!

Matching an expression such as `cout_ << 1 << 2` against the `Output` grammar proceeds as follows:

1. The first alternate of the `proto::or_<>` is tried first. It will fail, because the expression `cout_ << 1 << 2` does not match the grammar `proto::shift_left< proto::terminal< std::ostream & >, proto::_ >`.
2. Then the second alternate is tried next. We match the expression against `proto::shift_left< Output, proto::_ >`. The expression is a left-shift, so we next try to match the operands.
3. The right operand `2` matches `proto::_` trivially.
4. To see if the left operand `cout_ << 1` matches `Output`, we must recursively evaluate the `Output` grammar. This time we succeed, because `cout_ << 1` will match the first alternate of the `proto::or_<>`.

We're done -- the grammar matches successfully.

Fuzzy and Exact Matches of Terminals

The terminals in an expression tree could be const or non-const references, or they might not be references at all. When writing grammars, you usually don't have to worry about it because `proto::matches<>` gives you a little wiggle room when matching terminals. A grammar such as `proto::terminal<int>` will match a terminal of type `int`, `int &`, or `int const &`.

You can explicitly specify that you want to match a reference type. If you do, the type must match exactly. For instance, a grammar such as `proto::terminal<int &>` will only match an `int &`. It will not match an `int` or an `int const &`.

The table below shows how Proto matches terminals. The simple rule is: if you want to match only reference types, you must specify the reference in your grammar. Otherwise, leave it off and Proto will ignore const and references.

Table 6. proto::matches<> and Reference / CV-Qualification of Terminals

Terminal	Grammar	Matches?
T	T	yes
T &	T	yes
T const &	T	yes
T	T &	no
T &	T &	yes
T const &	T &	no
T	T const &	no
T &	T const &	no
T const &	T const &	yes

This begs the question: What if you want to match an `int`, but not an `int &` or an `int const &`? For forcing exact matches, Proto provides the `proto::exact<>` template. For instance, `proto::terminal< proto::exact<int> >` would only match an `int` held by value.

Proto gives you extra wiggle room when matching array types. Array types match themselves or the pointer types they decay to. This is especially useful with character arrays. The type returned by `proto::as_expr("hello")` is `proto::terminal<char const[6]>::type`. That's a terminal containing a 6-element character array. Naturally, you can match this terminal with the grammar `proto::terminal<char const[6]>`, but the grammar `proto::terminal<char const *>` will match it as well, as the following code fragment illustrates.

```
struct CharString
: proto::terminal< char const * >
{
};

typedef proto::terminal< char const[6] >::type char_array;

BOOST_MPL_ASSERT(( proto::matches< char_array, CharString > ));
```

What if we only wanted `CharString` to match terminals of exactly the type `char const *`? You can use `proto::exact<>` here to turn off the fuzzy matching of terminals, as follows:

```
struct CharString
: proto::terminal< proto::exact< char const * > >
{
};

typedef proto::terminal<char const[6]>::type char_array;
typedef proto::terminal<char const *>::type char_string;

BOOST_MPL_ASSERT(( proto::matches< char_string, CharString > ));
BOOST_MPL_ASSERT_NOT(( proto::matches< char_array, CharString > ));
```

Now, `CharString` does not match array types, only character string pointers.

The inverse problem is a little trickier: what if you wanted to match all character arrays, but not character pointers? As mentioned above, the expression `as_expr("hello")` has the type `proto::terminal< char const[6] >::type`. If you wanted to

match character arrays of arbitrary size, you could use `proto::N`, which is an array-size wildcard. The following grammar would match any string literal: `proto::terminal< char const[proto::N] >`.

Sometimes you need even more wiggle room when matching terminals. For example, maybe you're building a calculator DSEL and you want to allow any terminals that are convertible to double. For that, Proto provides the `proto::convertible_to<>` template. You can use it as: `proto::terminal< proto::convertible_to< double > >`.

There is one more way you can perform a fuzzy match on terminals. Consider the problem of trying to match a `std::complex<>` terminal. You can easily match a `std::complex<float>` or a `std::complex<double>`, but how would you match any instantiation of `std::complex<>`? You can use `proto::_` here to solve this problem. Here is the grammar to match any `std::complex<>` instantiation:

```
struct StdComplex
: proto::terminal< std::complex< proto::_ > >
{};
```

When given a grammar like this, Proto will deconstruct the grammar and the terminal it is being matched against and see if it can match all the constituents.

`if_<>`, `and_<>`, and `not_<>`

We've already seen how to use expression generators like `proto::terminal<>` and `proto::shift_right<>` as grammars. We've also seen `proto::or_<>`, which we can use to express a set of alternate grammars. There are a few others of interest; in particular, `proto::if_<>`, `proto::and_<>` and `proto::not_<>`.

The `proto::not_<>` template is the simplest. It takes a grammar as a template parameter and logically negates it; `not_<Grammar>` will match any expression that `Grammar` does *not* match.

The `proto::if_<>` template is used together with a Proto transform that is evaluated against expression types to find matches. (Proto transforms will be described later.)

The `proto::and_<>` template is like `proto::or_<>`, except that each argument of the `proto::and_<>` must match in order for the `proto::and_<>` to match. As an example, consider the definition of `CharString` above that uses `proto::exact<>`. It could have been written without `proto::exact<>` as follows:

```
struct CharString
: proto::and_<
    proto::terminal< proto::_ >
    , proto::if_< boost::is_same< proto::_value, char const * >() >
>
{};
```

This says that a `CharString` must be a terminal, *and* its value type must be the same as `char const *`. Notice the template argument of `proto::if_<>`: `boost::is_same< proto::_value, char const * >()`. This is Proto transform that compares the value type of a terminal to `char const *`.

The `proto::if_<>` template has a couple of variants. In addition to `if_<Condition>` you can also say `if_<Condition, ThenGrammar>` and `if_<Condition, ThenGrammar, ElseGrammar>`. These let you select one sub-grammar or another based on the `Condition`.

Improving Compile Times With `switch_<>`

When your Proto grammar gets large, you'll start to run into some scalability problems with `proto::or_<>`, the construct you use to specify alternate sub-grammars. First, due to limitations in C++, `proto::or_<>` can only accept up to a certain number of sub-grammars, controlled by the `BOOST_PROTO_MAX_LOGICAL_ARITY` macro. This macro defaults to eight, and you can set it higher, but doing so will aggravate another scalability problem: long compile times. With `proto::or_<>`, alternate sub-grammars are tried in order -- like a series of cascading `if`'s -- leading to lots of unnecessary template instantiations. What you would prefer instead is something like `switch` that avoids the expense of cascading `if`'s. That's the purpose of `proto::switch_<>`; although less convenient

than `proto::or_<>`, it improves compile times for larger grammars and does not have an arbitrary fixed limit on the number of sub-grammars.

Let's illustrate how to use `proto::switch_<>` by first writing a big grammar with `proto::or_<>` and then translating it to an equivalent grammar using `proto::switch_<>`:

```
// Here is a big, inefficient grammar
struct ABigGrammar
{
    proto::or_<
        proto::terminal<int>
        , proto::terminal<double>
        , proto::unary_plus<ABigGrammar>
        , proto::negate<ABigGrammar>
        , proto::complement<ABigGrammar>
        , proto::plus<ABigGrammar, ABigGrammar>
        , proto::minus<ABigGrammar, ABigGrammar>
        , proto::or_<
            proto::multiplies<ABigGrammar, ABigGrammar>
            , proto::divides<ABigGrammar, ABigGrammar>
            , proto::modulus<ABigGrammar, ABigGrammar>
        >
    >
};
```

The above might be the grammar to a more elaborate calculator DSEL. Notice that since there are more than eight sub-grammars, we had to chain the sub-grammars with a nested `proto::or_<>` -- not very nice.

The idea behind `proto::switch_<>` is to dispatch based on an expression's tag type to a sub-grammar that handles expressions of that type. To use `proto::switch_<>`, you define a struct with a nested `case_<>` template, specialized on tag types. The above grammar can be expressed using `proto::switch_<>` as follows. It is described below.


```

// Redefine ABigGrammar more efficiently using proto::switch_<>
struct ABigGrammar;

struct ABigGrammarCases
{
    // The primary template matches nothing:
    template<typename Tag>
    struct case_
    {
        : proto::not_<_>
    };
};

// Terminal expressions are handled here
template<>
struct ABigGrammarCases::case_<proto::tag::terminal>
{
    : proto::or_<
        proto::terminal<int>
        , proto::terminal<double>
    >
};

// Non-terminals are handled similarly
template<>
struct ABigGrammarCases::case_<proto::tag::unary_plus>
{
    : proto::unary_plus<ABigGrammar>
};

template<>
struct ABigGrammarCases::case_<proto::tag::negate>
{
    : proto::negate<ABigGrammar>
};

template<>
struct ABigGrammarCases::case_<proto::tag::complement>
{
    : proto::complement<ABigGrammar>
};

template<>
struct ABigGrammarCases::case_<proto::tag::plus>
{
    : proto::plus<ABigGrammar, ABigGrammar>
};

template<>
struct ABigGrammarCases::case_<proto::tag::minus>
{
    : proto::minus<ABigGrammar, ABigGrammar>
};

template<>
struct ABigGrammarCases::case_<proto::tag::multiplies>
{
    : proto::multiplies<ABigGrammar, ABigGrammar>
};

template<>
struct ABigGrammarCases::case_<proto::tag::divides>
{
    : proto::divides<ABigGrammar, ABigGrammar>
};

template<>
struct ABigGrammarCases::case_<proto::tag::modulus>
{
    : proto::modulus<ABigGrammar, ABigGrammar>
};

```



```
// Define ABigGrammar in terms of ABigGrammarCases
// using proto::switch_<>
struct ABigGrammar
: proto::switch_<ABigGrammarCases>
{};
```

Matching an expression type `E` against `proto::switch_<C>` is equivalent to matching it against `C::case_<E::proto_tag>`. By dispatching on the expression's tag type, we can jump to the sub-grammar that handles expressions of that type, skipping over all the other sub-grammars that couldn't possibly match. If there is no specialization of `case_<>` for a particular tag type, we select the primary template. In this case, the primary template inherits from `proto::not_<>` which matches no expressions.

Notice the specialization that handles terminals:

```
// Terminal expressions are handled here
template<>
struct ABigGrammarCases::case_<proto::tag::terminal>
: proto::or_<
    proto::terminal<int>
    , proto::terminal<double>
>
{};
```

The `proto::tag::terminal` type by itself isn't enough to select an appropriate sub-grammar, so we use `proto::or_<>` to list the alternate sub-grammars that match terminals.



Note

You might be tempted to define your `case_<>` specializations *in situ* as follows:

```
struct ABigGrammarCases
{
    template<typename Tag>
    struct case_ : proto::not_<_> {};

    // ERROR: not legal C++
    template<>
    struct case_<proto::tag::terminal>
        /* ... */
};
```

Unfortunately, for arcane reasons, it is not legal to define an explicit nested specialization *in situ* like this. It is, however, perfectly legal to define *partial* specializations *in situ*, so you can add a extra dummy template parameter that has a default, as follows:

```
struct ABigGrammarCases
{
    // Note extra "Dummy" template parameter here:
    template<typename Tag, int Dummy = 0>
    struct case_ : proto::not_<_> {};

    // OK: "Dummy" makes this a partial specialization
    // instead of an explicit specialization.
    template<int Dummy>
    struct case_<proto::tag::terminal, Dummy>
        /* ... */
};
```

You might find this cleaner than defining explicit `case_<>` specializations outside of their enclosing struct.

Matching Vararg Expressions

Not all of C++'s overloadable operators are unary or binary. There is the oddball `operator()` -- the function call operator -- which can have any number of arguments. Likewise, with Proto you may define your own "operators" that could also take more than two arguments. As a result, there may be nodes in your Proto expression tree that have an arbitrary number of children (up to `BOOST_PROTO_MAX_ARITY`, which is configurable). How do you write a grammar to match such a node?

For such cases, Proto provides the `proto::vararg<>` class template. Its template argument is a grammar, and the `proto::vararg<>` will match the grammar zero or more times. Consider a Proto lazy function called `fun()` that can take zero or more characters as arguments, as follows:

```
struct fun_tag {};
struct FunTag : proto::terminal< fun_tag > {};
FunTag::type const fun = {{}};

// example usage:
fun();
fun('a');
fun('a', 'b');
...
```

Below is the grammar that matches all the allowable invocations of `fun()`:


```
struct FunCall
: proto::function< FunTag, proto::vararg< proto::terminal< char > > >
{};
```

The FunCall grammar uses `proto::vararg<>` to match zero or more character literals as arguments of the `fun()` function.

As another example, can you guess what the following grammar matches?

```
struct Foo
: proto::or_<
    proto::terminal< proto::_ >
    , proto::nary_expr< proto::_ , proto::vararg< Foo > >
>
{};
```

Here's a hint: the first template parameter to `proto::nary_expr<>` represents the node type, and any additional template parameters represent child nodes. The answer is that this is a degenerate grammar that matches every possible expression tree, from root to leaves.

Defining DSEL Grammars

In this section we'll see how to use Proto to define a grammar for your DSEL and use it to validate expression templates, giving short, readable compile-time errors for invalid expressions.



Tip

You might think that this is a backwards way of doing things. “If Proto let me select which operators to overload, my users wouldn't be able to create invalid expressions in the first place, and I wouldn't need a grammar at all!” That may be true, but there are reasons for preferring to do things this way.

First, it lets you develop your DSEL rapidly -- all the operators are there for you already! -- and worry about invalid syntax later.

Second, it might be the case that some operators are only allowed in certain contexts within your DSEL. This is easy to express with a grammar, and hard to do with straight operator overloading.

Third, using a DSEL grammar to flag invalid expressions can often yield better errors than manually selecting the overloaded operators.

Fourth, the grammar can be used for more than just validation. You can use your grammar to define *tree transformations* that convert expression templates into other more useful objects.

If none of the above convinces you, you actually *can* use Proto to control which operators are overloaded within your domain. And to do it, you need to define a grammar!

In a previous section, we used Proto to define a DSEL for a lazily evaluated calculator that allowed any combination of placeholders, floating-point literals, addition, subtraction, multiplication, division and grouping. If we were to write the grammar for this DSEL in EBNF, it might look like this:

```
group      ::= '(' expression ')'
factor     ::= double | '_1' | '_2' | group
term       ::= factor (('*' factor) | ('/' factor))*
expression ::= term (('+' term) | ('-' term))*
```

This captures the syntax, associativity and precedence rules of a calculator. Writing the grammar for our calculator DSEL using Proto is *even simpler*. Since we are using C++ as the host language, we are bound to the associativity and precedence rules for the

C++ operators. Our grammar can assume them. Also, in C++ grouping is already handled for us with the use of parenthesis, so we don't have to code that into our grammar.

Let's begin our grammar for forward-declaring it:

```
struct CalculatorGrammar;
```

It's an incomplete type at this point, but we'll still be able to use it to define the rules of our grammar. Let's define grammar rules for the terminals:

```
struct Double
: proto::terminal< proto::convertible_to< double > >
{};

struct Placeholder1
: proto::terminal< placeholder<0> >
{};

struct Placeholder2
: proto::terminal< placeholder<1> >
{};

struct Terminal
: proto::or_< Double, Placeholder1, Placeholder2 >
{};
```

Now let's define the rules for addition, subtraction, multiplication and division. Here, we can ignore issues of associativity and precedence -- the C++ compiler will enforce that for us. We only must enforce that the arguments to the operators must themselves conform to the CalculatorGrammar that we forward-declared above.

```
struct Plus
: proto::plus< CalculatorGrammar, CalculatorGrammar >
{};

struct Minus
: proto::minus< CalculatorGrammar, CalculatorGrammar >
{};

struct Multiplies
: proto::multiplies< CalculatorGrammar, CalculatorGrammar >
{};

struct Divides
: proto::divides< CalculatorGrammar, CalculatorGrammar >
{};
```

Now that we've defined all the parts of the grammar, we can define CalculatorGrammar:


```
struct CalculatorGrammar
: proto::or_<
    Terminal
    , Plus
    , Minus
    , Multiplies
    , Divides
>
{};
```

That's it! Now we can use `CalculatorGrammar` to enforce that an expression template conforms to our grammar. We can use `proto::matches<>` and `BOOST_MPL_ASSERT()` to issue readable compile-time errors for invalid expressions, as below:

```
template< typename Expr >
void evaluate( Expr const & expr )
{
    BOOST_MPL_ASSERT(( proto::matches< Expr, CalculatorGrammar > ));
    // ...
}
```

Back Ends: Making Expression Templates Do Useful Work

Now that you've written the front end for your DSEL compiler, and you've learned a bit about the intermediate form it produces, it's time to think about what to *do* with the intermediate form. This is where you put your domain-specific algorithms and optimizations. Proto gives you two ways to evaluate and manipulate expression templates: contexts and transforms.

- A *context* is like a function object that you pass along with an expression to the `proto::eval()` function. It associates behaviors with node types. `proto::eval()` walks the expression and invokes your context at each node.
- A *transform* is a way to associate behaviors, not with node types in an expression, but with rules in a Proto grammar. In this way, they are like semantic actions in other compiler-construction toolkits.

Two ways to evaluate expressions! How to choose? Since contexts are largely procedural, they are a bit simpler to understand and debug so they are a good place to start. But although transforms are more advanced, they are also more powerful; since they are associated with rules in your grammar, you can select the proper transform based on the entire *structure* of a sub-expression rather than simply on the type of its top-most node.

Also, transforms have a concise and declarative syntax that can be confusing at first, but highly expressive and fungible once you become accustomed to it. And -- this is admittedly very subjective -- the author finds programming with Proto transforms to be an inordinate amount of *fun*! Your mileage may vary.

Expression Evaluation: Imparting Behaviors with a Context

Once you have constructed a Proto expression tree, either by using Proto's operator overloads or with `proto::make_expr()` and friends, you probably want to actually *do* something with it. The simplest option is to use `proto::eval()`, a generic expression evaluator. To use `proto::eval()`, you'll need to define a *context* that tells `proto::eval()` how each node should be evaluated. This section goes through the nuts and bolts of using `proto::eval()`, defining evaluation contexts, and using the contexts that Proto provides.



Note

`proto::eval()` is a less powerful but easier-to-use evaluation technique than Proto transforms, which are covered later. Although very powerful, transforms have a steep learning curve and can be more difficult to debug. `proto::eval()` is a rather weak tree traversal algorithm. Dan Marsden has been working on a more general and powerful tree traversal library. When it is ready, I anticipate that it will eliminate the need for `proto::eval()`.

Evaluating an Expression with `proto::eval()`

Synopsis:

```
namespace proto
{
    namespace result_of
    {
        // A metafunction for calculating the return
        // type of proto::eval() given certain Expr
        // and Context types.
        template<typename Expr, typename Context>
        struct eval
        {
            typedef
                typename Context::template eval<Expr>::result_type
                type;
        };
    }

    namespace functional
    {
        // A callable function object type for evaluating
        // a Proto expression with a certain context.
        struct eval : callable
        {
            template<typename Sig>
            struct result;

            template<typename Expr, typename Context>
            typename proto::result_of::eval<Expr, Context>::type
            operator()(Expr &expr, Context &context) const;

            template<typename Expr, typename Context>
            typename proto::result_of::eval<Expr, Context>::type
            operator()(Expr &expr, Context const &context) const;
        };

        template<typename Expr, typename Context>
        typename proto::result_of::eval<Expr, Context>::type
        eval(Expr &expr, Context &context);

        template<typename Expr, typename Context>
        typename proto::result_of::eval<Expr, Context>::type
        eval(Expr &expr, Context const &context);
    }
}
```

Given an expression and an evaluation context, using `proto::eval()` is quite simple. Simply pass the expression and the context to `proto::eval()` and it does the rest and returns the result. You can use the `eval<>` metafunction in the `proto::result_of` namespace to compute the return type of `proto::eval()`. The following demonstrates a use of `proto::eval()`:


```
template<typename Expr>
typename proto::result_of::eval<Expr const, MyContext>::type
MyEvaluate(Expr const &expr)
{
    // Some user-defined context type
    MyContext ctx;

    // Evaluate an expression with the context
    return proto::eval(expr, ctx);
}
```

What `proto::eval()` does is also very simple. It defers most of the work to the context itself. Here essentially is the implementation of `proto::eval()`:

```
// eval() dispatches to a nested "eval<>" function
// object within the Context:
template<typename Expr, typename Context>
typename Context::template eval<Expr>::result_type
eval(Expr &expr, Context &ctx)
{
    typename Context::template eval<Expr> eval_fun;
    return eval_fun(expr, ctx);
}
```

Really, `proto::eval()` is nothing more than a thin wrapper that dispatches to the appropriate handler within the context class. In the next section, we'll see how to implement a context class from scratch.

Defining an Evaluation Context

As we saw in the previous section, there is really not much to the `proto::eval()` function. Rather, all the interesting expression evaluation goes on within a context class. This section shows how to implement one from scratch.

All context classes have roughly the following form:


```
// A prototypical user-defined context.
struct MyContext
{
    // A nested eval<> class template
    template<
        typename Expr
        , typename Tag = typename proto::tag_of<Expr>::type
    >
    struct eval;

    // Handle terminal nodes here...
    template<typename Expr>
    struct eval<Expr, proto::tag::terminal>
    {
        // Must have a nested result_type typedef.
        typedef ... result_type;

        // Must have a function call operator that takes
        // an expression and the context.
        result_type operator()(Expr &expr, MyContext &ctx) const
        {
            return ...;
        }
    };

    // ... other specializations of struct eval<> ...
};
```

Context classes are nothing more than a collection of specializations of a nested `eval<>` class template. Each specialization handles a different expression type.

In the [Hello Calculator](#) section, we saw an example of a user-defined context class for evaluating calculator expressions. That context class was implemented with the help of Proto's `proto::callable_context<>`. If we were to implement it from scratch, it would look something like this:


```

// The calculator_context from the "Hello Calculator" section,
// implemented from scratch.
struct calculator_context
{
    // The values with which we'll replace the placeholders
    std::vector<double> args;

    template<
        typename Expr
        // defaulted template parameters, so we can
        // specialize on the expressions that need
        // special handling.
        , typename Tag = typename proto::tag_of<Expr>::type
        , typename Arg0 = typename proto::child_c<Expr, 0>::type
    >
    struct eval;

    // Handle placeholder terminals here...
    template<typename Expr, int I>
    struct eval<Expr, proto::tag::terminal, placeholder<I> >
    {
        typedef double result_type;

        result_type operator()(Expr &, MyContext &ctx) const
        {
            return ctx.args[I];
        }
    };

    // Handle other terminals here...
    template<typename Expr, typename Arg0>
    struct eval<Expr, proto::tag::terminal, Arg0>
    {
        typedef double result_type;

        result_type operator()(Expr &expr, MyContext &) const
        {
            return proto::child(expr);
        }
    };

    // Handle addition here...
    template<typename Expr, typename Arg0>
    struct eval<Expr, proto::tag::plus, Arg0>
    {
        typedef double result_type;

        result_type operator()(Expr &expr, MyContext &ctx) const
        {
            return proto::eval(proto::left(expr), ctx)
                + proto::eval(proto::right(expr), ctx);
        }
    };

    // ... other eval<> specializations for other node types ...
};

```

Now we can use `proto::eval()` with the context class above to evaluate calculator expressions as follows:


```
// Evaluate an expression with a calculator_context
calculator_context ctx;
ctx.args.push_back(5);
ctx.args.push_back(6);
double d = proto::eval(_1 + _2, ctx);
assert(11 == d);
```

Defining a context from scratch this way is tedious and verbose, but it gives you complete control over how the expression is evaluated. The context class in the [Hello Calculator](#) example was much simpler. In the next section we'll see the helper class Proto provides to ease the job of implementing context classes.

Proto's Built-In Contexts

Proto provides some ready-made context classes that you can use as-is, or that you can use to help while implementing your own contexts. They are:

<code>default_context</code>	An evaluation context that assigns the usual C++ meanings to all the operators. For example, addition nodes are handled by evaluating the left and right children and then adding the results. The <code>proto::default_context</code> uses <code>Boost.Typeof</code> to deduce the types of the expressions it evaluates.
<code>null_context</code>	A simple context that recursively evaluates children but does not combine the results in any way and returns void.
<code>callable_context<></code>	A helper that simplifies the job of writing context classes. Rather than writing template specializations, with <code>proto::callable_context<></code> you write a function object with an overloaded function call operator. Any expressions not handled by an overload are automatically dispatched to a default evaluation context that you can specify.

`default_context`

The `proto::default_context` is an evaluation context that assigns the usual C++ meanings to all the operators. For example, addition nodes are handled by evaluating the left and right children and then adding the results. The `proto::default_context` uses `Boost.Typeof` to deduce the types of the expressions it evaluates.

For example, consider the following "Hello World" example:

```
#include <iostream>
#include <boost/proto/proto.hpp>
#include <boost/proto/context.hpp>
#include <boost/typeof/std/ostream.hpp>
using namespace boost;

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

template< typename Expr >
void evaluate( Expr const & expr )
{
    // Evaluate the expression with default_context,
    // to give the operators their C++ meanings:
    proto::default_context ctx;
    proto::eval(expr, ctx);
}

int main()
{
    evaluate( cout_ << "hello" << ',' << " world" );
    return 0;
}
```

This program outputs the following:


```
hello, world
```

`proto::default_context` is trivially defined in terms of a `default_eval<>` template, as follows:

```
// Definition of default_context
struct default_context
{
    template<typename Expr>
    struct eval
    : default_eval<
        Expr
        , default_context const
        , typename tag_of<Expr>::type
    >
    {};
};
```

There are a bunch of `default_eval<>` specializations, each of which handles a different C++ operator. Here, for instance, is the specialization for binary addition:

```
// A default expression evaluator for binary addition
template<typename Expr, typename Context>
struct default_eval<Expr, Context, proto::tag::plus>
{
private:
    static Expr      & s_expr;
    static Context    & s_ctx;

public:
    typedef
        decltype(
            proto::eval(proto::child_c<0>(s_expr), s_ctx)
            + proto::eval(proto::child_c<1>(s_expr), s_ctx)
        )
        result_type;

    result_type operator ()(Expr &expr, Context &ctx) const
    {
        return proto::eval(proto::child_c<0>(expr), ctx)
            + proto::eval(proto::child_c<1>(expr), ctx);
    }
};
```

The above code uses `decltype` to calculate the return type of the function call operator. `decltype` is a new keyword in the next version of C++ that gets the type of any expression. Most compilers do not yet support `decltype` directly, so `default_eval<>` uses the `Boost.Typeof` library to emulate it. On some compilers, that may mean that `default_context` either doesn't work or that it requires you to register your types with the `Boost.Typeof` library. Check the documentation for `Boost.Typeof` to see.

`null_context`

The `proto::null_context<>` is a simple context that recursively evaluates children but does not combine the results in any way and returns void. It is useful in conjunction with `callable_context<>`, or when defining your own contexts which mutate an expression tree in-place rather than accumulate a result, as we'll see below.

`proto::null_context<>` is trivially implemented in terms of `null_eval<>` as follows:


```
// Definition of null_context
struct null_context
{
    template<typename Expr>
    struct eval
        : null_eval<Expr, null_context const, Expr::proto_arity::value>
    {};
};
```

And `null_eval<>` is also trivially implemented. Here, for instance is a binary `null_eval<>`:

```
// Binary null_eval<>
template<typename Expr, typename Context>
struct null_eval<Expr, Context, 2>
{
    typedef void result_type;

    void operator()(Expr &expr, Context &ctx) const
    {
        proto::eval(proto::child_c<0>(expr), ctx);
        proto::eval(proto::child_c<1>(expr), ctx);
    }
};
```

When would such classes be useful? Imagine you have an expression tree with integer terminals, and you would like to increment each integer in-place. You might define an evaluation context as follows:

```
struct increment_ints
{
    // By default, just evaluate all children by delegating
    // to the null_eval<>
    template<typename Expr, typename Arg = proto::result_of::child<Expr>::type>
    struct eval
        : null_eval<Expr, increment_ints const>
    {};

    // Increment integer terminals
    template<typename Expr>
    struct eval<Expr, int>
    {
        typedef void result_type;

        void operator()(Expr &expr, increment_ints const &) const
        {
            ++proto::child(expr);
        }
    };
};
```

In the next section on `proto::callable_context<>`, we'll see an even simpler way to achieve the same thing.

`callable_context<>`

The `proto::callable_context<>` is a helper that simplifies the job of writing context classes. Rather than writing template specializations, with `proto::callable_context<>` you write a function object with an overloaded function call operator. Any expressions not handled by an overload are automatically dispatched to a default evaluation context that you can specify.

Rather than an evaluation context in its own right, `proto::callable_context<>` is more properly thought of as a context adaptor. To use it, you must define your own context that inherits from `proto::callable_context<>`.

In the `null_context` section, we saw how to implement an evaluation context that increments all the integers within an expression tree. Here is how to do the same thing with the `proto::callable_context<>`:

```
// An evaluation context that increments all
// integer terminals in-place.
struct increment_ints
: callable_context<
    increment_ints const // derived context
    , null_context const // fall-back context
>
{
    typedef void result_type;

    // Handle int terminals here:
    void operator()(proto::tag::terminal, int &i) const
    {
        ++i;
    }
};
```

With such a context, we can do the following:

```
literal<int> i = 0, j = 10;
proto::eval( i - j * 3.14, increment_ints() );

std::cout << "i = " << i.get() << std::endl;
std::cout << "j = " << j.get() << std::endl;
```

This program outputs the following, which shows that the integers `i` and `j` have been incremented by 1:

```
i = 1
j = 11
```

In the `increment_ints` context, we didn't have to define any nested `eval<>` templates. That's because `proto::callable_context<>` implements them for us. `proto::callable_context<>` takes two template parameters: the derived context and a fall-back context. For each node in the expression tree being evaluated, `proto::callable_context<>` checks to see if there is an overloaded `operator()` in the derived context that accepts it. Given some expression `expr` of type `Expr`, and a context `ctx`, it attempts to call:

```
ctx(
    typename Expr::proto_tag()
    , proto::child_c<0>(expr)
    , proto::child_c<1>(expr)
    ...
);
```

Using function overloading and metaprogramming tricks, `proto::callable_context<>` can detect at compile-time whether such a function exists or not. If so, that function is called. If not, the current expression is passed to the fall-back evaluation context to be processed.

We saw another example of the `proto::callable_context<>` when we looked at the simple calculator expression evaluator. There, we wanted to customize the evaluation of placeholder terminals, and delegate the handling of all other nodes to the `proto::default_context`. We did that as follows:


```
// An evaluation context for calculator expressions that
// explicitly handles placeholder terminals, but defers the
// processing of all other nodes to the default_context.
struct calculator_context
: proto::callable_context< calculator_context const >
{
    std::vector<double> args;

    // Define the result type of the calculator.
    typedef double result_type;

    // Handle the placeholders:
    template<int I>
    double operator()(proto::tag::terminal, placeholder<I>) const
    {
        return this->args[I];
    }
};
```

In this case, we didn't specify a fall-back context. In that case, `proto::callable_context<>` uses the `proto::default_context`. With the above `calculator_context` and a couple of appropriately defined placeholder terminals, we can evaluate calculator expressions, as demonstrated below:

```
template<int I>
struct placeholder
{
};

terminal<placeholder<0> >::type const _1 = { { } };
terminal<placeholder<1> >::type const _2 = { { } };
// ...

calculator_context ctx;
ctx.args.push_back(4);
ctx.args.push_back(5);

double j = proto::eval( (_2 - _1) / _2 * 100, ctx );
std::cout << "j = " << j << std::endl;
```

The above code displays the following:

```
j = 20
```

Expression Transformation: Semantic Actions

If you have ever built a parser with the help of a tool like Antlr, yacc or Boost.Spirit, you might be familiar with *semantic actions*. In addition to allowing you to define the grammar of the language recognized by the parser, these tools let you embed code within your grammar that executes when parts of the grammar participate in a parse. Proto has the equivalent of semantic actions. They are called *transforms*. This section describes how to embed transforms within your Proto grammars, turning your grammars into function objects that can manipulate or evaluate expressions in powerful ways.

Proto transforms are an advanced topic. We'll take it slow, using examples to illustrate the key concepts, starting simple.

“Activating” Your Grammars

The Proto grammars we've seen so far are static. You can check at compile-time to see if an expression type matches a grammar, but that's it. Things get more interesting when you give them runtime behaviors. A grammar with embedded transforms is more than just a static grammar. It is a function object that accepts expressions that match the grammar and does *something* with them.

Below is a very simple grammar. It matches terminal expressions.


```
// A simple Proto grammar that matches all terminals
proto::terminal< _ >
```

Here is the same grammar with a transform that extracts the value from the terminal:

```
// A simple Proto grammar that matches all terminals
// *and* a function object that extracts the value from
// the terminal
proto::when<
    proto::terminal< _ >
    , proto::_value          // <-- Look, a transform!
>
```

You can read this as follows: when you match a terminal expression, extract the value. The type `proto::_value` is a so-called transform. Later we'll see what makes it a transform, but for now just think of it as a kind of function object. Note the use of `proto::when<>`: the first template parameter is the grammar to match and the second is the transform to execute. The result is both a grammar that matches terminal expressions and a function object that accepts terminal expressions and extracts their values.

As with ordinary grammars, we can define an empty struct that inherits from a grammar+transform to give us an easy way to refer back to the thing we're defining, as follows:

```
// A grammar and a function object, as before
struct Value
: proto::when<
    proto::terminal< _ >
    , proto::_value
>
{};

// "Value" is a grammar that matches terminal expressions
BOOST_MPL_ASSERT(( proto::matches< proto::terminal<int>::type, Value > ));

// "Value" also defines a function object that accepts terminals
// and extracts their value.
proto::terminal<int>::type answer = {42};
Value get_value;
int i = get_value( answer );
```

As already mentioned, `Value` is a grammar that matches terminal expressions and a function object that operates on terminal expressions. It would be an error to pass a non-terminal expression to the `Value` function object. This is a general property of grammars with transforms; when using them as function objects, expressions passed to them must match the grammar.

Proto grammars are valid TR1-style function objects. That means you can use `boost::result_of<>` to ask a grammar what its return type will be, given a particular expression type. For instance, we can access the `Value` grammar's return type as follows:


```
// We can use boost::result_of<> to get the return type
// of a Proto grammar.
typedef
    typename boost::result_of<Value(proto::terminal<int>::type)>::type
result_type;

// Check that we got the type we expected
BOOST_MPL_ASSERT(( boost::is_same<result_type, int> ));
```



Note

A grammar with embedded transforms is both a grammar and a function object. Calling these things "grammars with transforms" would get tedious. We could call them something like "active grammars", but as we'll see *every* grammar that you can define with Proto is "active"; that is, every grammar has some behavior when used as a function object. So we'll continue calling these things plain "grammars". The term "transform" is reserved for the thing that is used as the second parameter to the `proto::when<>` template.

Handling Alternation and Recursion

Most grammars are a little more complicated than the one in the preceding section. For the sake of illustration, let's define a rather nonsensical grammar that matches any expression and recurses to the leftmost terminal and returns its value. It will demonstrate how two key concepts of Proto grammars -- alternation and recursion -- interact with transforms. The grammar is described below.

```
// A grammar that matches any expression, and a function object
// that returns the value of the leftmost terminal.
struct LeftmostLeaf
    : proto::or_<
        // If the expression is a terminal, return its value
        proto::when<
            proto::terminal< _ >
            , proto::_value
        >
        // Otherwise, it is a non-terminal. Return the result
        // of invoking LeftmostLeaf on the 0th (leftmost) child.
        , proto::when<
            _
            , LeftmostLeaf( proto::_child0 )
        >
    >
{};

// A Proto terminal wrapping std::cout
proto::terminal< std::ostream & >::type cout_ = { std::cout };

// Create an expression and use LeftmostLeaf to extract the
// value of the leftmost terminal, which will be std::cout.
std::ostream & sout = LeftmostLeaf()( cout_ << "the answer: " << 42 << '\n' );
```

We've seen `proto::or_<>` before. Here it is serving two roles. First, it is a grammar that matches any of its alternate sub-grammars; in this case, either a terminal or a non-terminal. Second, it is also a function object that accepts an expression, finds the alternate sub-grammar that matches the expression, and applies its transform. And since `LeftmostLeaf` inherits from `proto::or_<>`, `LeftmostLeaf` is also both a grammar and a function object.



Note

The second alternate uses `proto::_` as its grammar. Recall that `proto::_` is the wildcard grammar that matches any expression. Since alternates in `proto::or_<>` are tried in order, and since the first alternate handles all terminals, the second alternate handles all (and only) non-terminals. Often enough, `proto::when< _, some-transform >` is the last alternate in a grammar, so for improved readability, you could use the equivalent `proto::otherwise< some-transform >`.

The next section describes this grammar further.

Callable Transforms

In the grammar defined in the preceding section, the transform associated with non-terminals is a little strange-looking:

```
proto::when<
  _
  , LeftmostLeaf( proto::_child0 )    // <-- a "callable" transform
>
```

It has the effect of accepting non-terminal expressions, taking the 0th (leftmost) child and recursively invoking the `LeftmostLeaf` function on it. But `LeftmostLeaf(proto::_child0)` is actually a *function type*. Literally, it is the type of a function that accepts an object of type `proto::_child0` and returns an object of type `LeftmostLeaf`. So how do we make sense of this transform? Clearly, there is no function that actually has this signature, nor would such a function be useful. The key is in understanding how `proto::when<>` *interprets* its second template parameter.

When the second template parameter to `proto::when<>` is a function type, `proto::when<>` interprets the function type as a transform. In this case, `LeftmostLeaf` is treated as the type of a function object to invoke, and `proto::_child0` is treated as a transform. First, `proto::_child0` is applied to the current expression (the non-terminal that matched this alternate sub-grammar), and the result (the 0th child) is passed as an argument to `LeftmostLeaf`.



Note

Transforms are a Domain-Specific Language

`LeftmostLeaf(proto::_child0)` looks like an invocation of the `LeftmostLeaf` function object, but it's not, but then it actually is! Why this confusing subterfuge? Function types give us a natural and concise syntax for composing more complicated transforms from simpler ones. The fact that the syntax is suggestive of a function invocation is on purpose. It is a domain-specific embedded language for defining expression transformations. If the subterfuge worked, it may have fooled you into thinking the transform is doing exactly what it actually does! And that's the point.

The type `LeftmostLeaf(proto::_child0)` is an example of a *callable transform*. It is a function type that represents a function object to call and its arguments. The types `proto::_child0` and `proto::_value` are *primitive transforms*. They are plain structs, not unlike function objects, from which callable transforms can be composed. There is one other type of transform, *object transforms*, that we'll encounter next.

Object Transforms

The very first transform we looked at simply extracted the value of terminals. Let's do the same thing, but this time we'll promote all ints to longs first. (Please forgive the contrived-ness of the examples so far; they get more interesting later.) Here's the grammar:


```
// A simple Proto grammar that matches all terminals,
// and a function object that extracts the value from
// the terminal, promoting ints to longs:
struct ValueWithPromote
{
    proto::or_<
        proto::when<
            proto::terminal< int >
            , long(proto::_value)      // <-- an "object" transform
        >
        , proto::when<
            proto::terminal< _ >
            , proto::_value
        >
    >
};
```

You can read the above grammar as follows: when you match an int terminal, extract the value from the terminal and use it to initialize a long; otherwise, when you match another kind of terminal, just extract the value. The type `long(proto::_value)` is a so-called *object* transform. It looks like the creation of a temporary long, but it's really a function type. Just as a callable transform is a function type that represents a function to call and its arguments, an object transform is a function type that represents an object to construct and the arguments to its constructor.



Note

Object Transforms vs. Callable Transforms

When using function types as Proto transforms, they can either represent an object to construct or a function to call. It is similar to "normal" C++ where the syntax `foo("arg")` can either be interpreted as an object to construct or a function to call, depending on whether `foo` is a type or a function. But consider two of the transforms we've seen so far:

```
LeftmostLeaf(proto::_child0) // <-- a callable transform
long(proto::_value)          // <-- an object transform
```

Proto can't know in general which is which, so it uses a trait, `proto::is_callable<>`, to differentiate. `is_callable< long >::value` is false so `long(proto::_value)` is an object to construct, but `is_callable< LeftmostLeaf >::value` is true so `LeftmostLeaf(proto::_child0)` is a function to call. Later on, we'll see how Proto recognizes a type as "callable".

Example: Calculator Arity

Now that we have the basics of Proto transforms down, let's consider a slightly more realistic example. We can use transforms to improve the type-safety of the [calculator DSEL](#). If you recall, it lets you write infix arithmetic expressions involving argument placeholders like `_1` and `_2` and pass them to STL algorithms as function objects, as follows:

```
double a1[4] = { 56, 84, 37, 69 };
double a2[4] = { 65, 120, 60, 70 };
double a3[4] = { 0 };

// Use std::transform() and a calculator expression
// to calculate percentages given two input sequences:
std::transform(a1, a1+4, a2, a3, (_2 - _1) / _2 * 100);
```

This works because we gave calculator expressions an `operator()` that evaluates the expression, replacing the placeholders with the arguments to `operator()`. The overloaded `calculator<>::operator()` looked like this:


```
// Overload operator() to invoke proto::eval() with
// our calculator_context.
template<typename Expr>
double
calculator<Expr>::operator()(double a1 = 0, double a2 = 0) const
{
    calculator_context ctx;
    ctx.args.push_back(a1);
    ctx.args.push_back(a2);

    return proto::eval(*this, ctx);
}
```

Although this works, it's not ideal because it doesn't warn users if they supply too many or too few arguments to a calculator expression. Consider the following mistakes:

```
(_1 * _1)(4, 2); // Oops, too many arguments!
(_2 * _2)(42);   // Oops, too few arguments!
```

The expression `_1 * _1` defines a unary calculator expression; it takes one argument and squares it. If we pass more than one argument, the extra arguments will be silently ignored, which might be surprising to users. The next expression, `_2 * _2` defines a binary calculator expression; it takes two arguments, ignores the first and squares the second. If we only pass one argument, the code silently fills in 0.0 for the second argument, which is also probably not what users expect. What can be done?

We can say that the *arity* of a calculator expression is the number of arguments it expects, and it is equal to the largest placeholder in the expression. So, the arity of `_1 * _1` is one, and the arity of `_2 * _2` is two. We can increase the type-safety of our calculator DSEL by making sure the arity of an expression equals the actual number of arguments supplied. Computing the arity of an expression is simple with the help of Proto transforms.

It's straightforward to describe in words how the arity of an expression should be calculated. Consider that calculator expressions can be made of `_1`, `_2`, literals, unary expressions and binary expressions. The following table shows the arities for each of these 5 constituents.

Table 7. Calculator Sub-Expression Arities

Sub-Expression	Arity
Placeholder 1	1
Placeholder 2	2
Literal	0
Unary Expression	<i>arity of the operand</i>
Binary Expression	<i>max arity of the two operands</i>

Using this information, we can write the grammar for calculator expressions and attach transforms for computing the arity of each constituent. The code below computes the expression arity as a compile-time integer, using integral wrappers and metafunctions from the Boost MPL Library. The grammar is described below.


```
struct CalcArity
: proto::or_<
    proto::when< proto::terminal< placeholder<0> >,
        mpl::int_<1>()
    >
    , proto::when< proto::terminal< placeholder<1> >,
        mpl::int_<2>()
    >
    , proto::when< proto::terminal<_>,
        mpl::int_<0>()
    >
    , proto::when< proto::unary_expr<_, CalcArity>,
        CalcArity(proto::_child)
    >
    , proto::when< proto::binary_expr<_, CalcArity, CalcArity>,
        mpl::max<CalcArity(proto::_left),
            CalcArity(proto::_right)>()
    >
>
{};
```

When we find a placeholder terminal or a literal, we use an *object transform* such as `mpl::int_<1>()` to create a (default-constructed) compile-time integer representing the arity of that terminal.

For unary expressions, we use `CalcArity(proto::_child)` which is a *callable transform* that computes the arity of the expression's child.

The transform for binary expressions has a few new tricks. Let's look more closely:

```
// Compute the left and right arities and
// take the larger of the two.
mpl::max<CalcArity(proto::_left),
    CalcArity(proto::_right)>()
```

This is an object transform; it default-constructs ... what exactly? The `mpl::max<>` template is an MPL metafunction that accepts two compile-time integers. It has a nested `::type` typedef (not shown) that is the maximum of the two. But here, we appear to be passing it two things that are *not* compile-time integers; they're Proto callable transforms. Proto is smart enough to recognize that fact. It first evaluates the two nested callable transforms, computing the arities of the left and right child expressions. Then it puts the resulting integers into `mpl::max<>` and evaluates the metafunction by asking for the nested `::type`. That is the type of the object that gets default-constructed and returned.

More generally, when evaluating object transforms, Proto looks at the object type and checks whether it is a template specialization, like `mpl::max<>`. If it is, Proto looks for nested transforms that it can evaluate. After any nested transforms have been evaluated and substituted back into the template, the new template specialization is the result type, unless that type has a nested `::type`, in which case that becomes the result.

Now that we can calculate the arity of a calculator expression, let's redefine the `calculator<>` expression wrapper we wrote in the Getting Started guide to use the `CalcArity` grammar and some macros from Boost.MPL to issue compile-time errors when users specify too many or too few arguments.


```

// The calculator expression wrapper, as defined in the Hello
// Calculator example in the Getting Started guide. It behaves
// just like the expression it wraps, but with extra operator()
// member functions that evaluate the expression.
// NEW: Use the CalcArity grammar to ensure that the correct
// number of arguments are supplied.
template<typename Expr>
struct calculator
{
    : proto::extends<Expr, calculator<Expr>, calculator_domain>
    {
        typedef
            proto::extends<Expr, calculator<Expr>, calculator_domain>
            base_type;

        calculator(Expr const &expr = Expr())
            : base_type(expr)
        {}

        typedef double result_type;

        // Use CalcArity to compute the arity of Expr:
        static int const arity = boost::result_of<CalcArity(Expr)>::type::value;

        double operator()() const
        {
            BOOST_MPL_ASSERT_RELATION(0, ==, arity);
            calculator_context ctx;
            return proto::eval(*this, ctx);
        }

        double operator()(double a1) const
        {
            BOOST_MPL_ASSERT_RELATION(1, ==, arity);
            calculator_context ctx;
            ctx.args.push_back(a1);
            return proto::eval(*this, ctx);
        }

        double operator()(double a1, double a2) const
        {
            BOOST_MPL_ASSERT_RELATION(2, ==, arity);
            calculator_context ctx;
            ctx.args.push_back(a1);
            ctx.args.push_back(a2);
            return proto::eval(*this, ctx);
        }
    };
};

```

Note the use of `boost::result_of<>` to access the return type of the `CalcArity` function object. Since we used compile-time integers in our transforms, the arity of the expression is encoded in the return type of the `CalcArity` function object. Proto grammars are valid TR1-style function objects, so you can use `boost::result_of<>` to figure out their return types.

With our compile-time assertions in place, when users provide too many or too few arguments to a calculator expression, as in:

```

(_2 * _2)(42); // Oops, too few arguments!

```

... they will get a compile-time error message on the line with the assertion that reads something like this²:

² This error message was generated with Microsoft Visual C++ 9.0. Different compilers will emit different messages with varying degrees of readability.


```
c:\boost\org\trunk\libs\proto\scratch\main.cpp(97) : error C2664: 'boost::mpl::assertion_failed' : cannot convert parameter 1 from 'boost::mpl::failed *****boost::mpl::assert_relation<x,y,___formal>::*****' to 'boost::mpl::assert<false>::type'
    with
    [
        x=1,
        y=2,
        ___formal=bool boost::mpl::operator==(boost::mpl::failed,boost::mpl::failed)
    ]
```

The point of this exercise was to show that we can write a fairly simple Proto grammar with embedded transforms that is declarative and readable and can compute interesting properties of arbitrarily complicated expressions. But transforms can do more than that. Boost.Xpressive uses transforms to turn expressions into finite state automata for matching regular expressions, and Boost.Spirit uses transforms to build recursive descent parser generators. Proto comes with a collection of built-in transforms that you can use to perform very sophisticated expression manipulations like these. In the next few sections we'll see some of them in action.

Transforms With State Accumulation

So far, we've only seen examples of grammars with transforms that accept one argument: the expression to transform. But consider for a moment how, in ordinary procedural code, you would turn a binary tree into a linked list. You would start with an empty list. Then, you would recursively convert the right branch to a list, and use the result as the initial state while converting the left branch to a list. That is, you would need a function that takes two parameters: the current node and the list so far. These sorts of *accumulation* problems are quite common when processing trees. The linked list is an example of an accumulation variable or *state*. Each iteration of the algorithm takes the current element and state, applies some binary function to the two and creates a new state. In the STL, this algorithm is called `std::accumulate()`. In many other languages, it is called *fold*. Let's see how to implement a fold algorithm with Proto transforms.

All Proto grammars can optionally accept a state parameter in addition to the expression to transform. If you want to fold a tree to a list, you'll need to make use of the state parameter to pass around the list you've built so far. As for the list, the Boost.Fusion library provides a `fusion::cons<>` type from which you can build heterogeneous lists. The type `fusion::nil` represents an empty list.

Below is a grammar that recognizes output expressions like `cout_ << 42 << '\n'` and puts the arguments into a Fusion list. It is explained below.


```
// Fold the terminals in output statements like
// "cout_ << 42 << '\n'" into a Fusion cons-list.
struct FoldToList
: proto::or_<
    // Don't add the ostream terminal to the list
    proto::when<
        proto::terminal< std::ostream & >
        , proto::_state
    >
    // Put all other terminals at the head of the
    // list that we're building in the "state" parameter
    , proto::when<
        proto::terminal<_>
        , fusion::cons<proto::_value, proto::_state>(
            proto::_value, proto::_state
        )
    >
    // For left-shift operations, first fold the right
    // child to a list using the current state. Use
    // the result as the state parameter when folding
    // the left child to a list.
    , proto::when<
        proto::shift_left<FoldToList, FoldToList>
        , FoldToList(
            proto::_left
            , FoldToList(proto::_right, proto::_state)
        )
    >
>
{};
```

Before reading on, see if you can apply what you know already about object, callable and primitive transforms to figure out how this grammar works.

When you use the `FoldToList` function, you'll need to pass two arguments: the expression to fold, and the initial state: an empty list. Those two arguments get passed around to each transform. We learned previously that `proto::_value` is a primitive transform that accepts a terminal expression and extracts its value. What we didn't know until now was that it also accepts the current state *and ignores it*. `proto::_state` is also a primitive transform. It accepts the current expression, which it ignores, and the current state, which it returns.

When we find a terminal, we stick it at the head of the cons list, using the current state as the tail of the list. (The first alternate causes the ostream to be skipped. We don't want `cout` in the list.) When we find a shift-left node, we apply the following transform:

```
// Fold the right child and use the result as
// state while folding the right.
FoldToList(
    proto::_left
    , FoldToList(proto::_right, proto::_state)
)
```

You can read this transform as follows: using the current state, fold the right child to a list. Use the new list as the state while folding the left child to a list.



Tip

If your compiler is Microsoft Visual C++, you'll find that the above transform does not compile. The compiler has bugs with its handling of nested function types. You can work around the bug by wrapping the inner transform in `proto::call<>` as follows:

```
FoldToList(
    proto::_left
    , proto::call<FoldToList(proto::_right, proto::_state)>
)
```

`proto::call<>` turns a callable transform into a primitive transform, but more on that later.

Now that we have defined the `FoldToList` function object, we can use it to turn output expressions into lists as follows:

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

// This is the type of the list we build below
typedef
    fusion::cons<
        int
        , fusion::cons<
            double
            , fusion::cons<
                char
                , fusion::nil
            >
        >
    >
    result_type;

// Fold an output expression into a Fusion list, using
// fusion::nil as the initial state of the transformation.
FoldToList to_list;
result_type args = to_list(cout_ << 1 << 3.14 << '\n', fusion::nil());

// Now "args" is the list: {1, 3.14, '\n'}
```

When writing transforms, "fold" is such a basic operation that Proto provides a number of built-in fold transforms. We'll get to them later. For now, rest assured that you won't always have to stretch your brain so far to do such basic things.

Passing Auxiliary Data to Transforms

In the last section, we saw that we can pass a second parameter to grammars with transforms: an accumulation variable or *state* that gets updated as your transform executes. There are times when your transforms will need to access auxiliary data that does *not* accumulate, so bundling it with the state parameter is impractical. Instead, you can pass auxiliary data as a third parameter, known as the *data* parameter. Below we show an example involving string processing where the data parameter is essential.



Note

All Proto grammars are function objects that take one, two or three arguments: the expression, the state, and the data. There are no additional arguments to know about, we promise. In Haskell, there is set of tree traversal technologies known collectively as "[Scrap Your Boilerplate](#)". In that framework, there are also three parameters: the term, the accumulator, and the context. These are Proto's expression, state and data parameters under different names.

Expression templates are often used as an optimization to eliminate temporary objects. Consider the problem of string concatenation: a series of concatenations would result in the needless creation of temporary strings. We can use Proto to make string concatenation

very efficient. To make the problem more interesting, we can apply a locale-sensitive transformation to each character during the concatenation. The locale information will be passed as the data parameter.

Consider the following expression template:

```
proto::lit("hello") + " " + "world";
```

We would like to concatenate this string into a statically allocated wide character buffer, widening each character in turn using the specified locale. The first step is to write a grammar that describes this expression, with transforms that calculate the total string length. Here it is:

```
// A grammar that matches string concatenation expressions, and
// a transform that calculates the total string length.
struct StringLength
: proto::or_<
    proto::when<
        // When you find a character array ...
        proto::terminal<char[proto::N]>
        // ... the length is the size of the array minus 1.
        , mpl::prior<mpl::sizeof_proto::_value> >()
    >
    , proto::when<
        // The length of a concatenated string is ...
        proto::plus<StringLength, StringLength>
        // ... the sum of the lengths of each sub-string.
        , proto::fold<
            _
            , mpl::size_t<0>()
            , mpl::plus<StringLength, proto::_state>()
        >
    >
>
{};
```

Notice the use of `proto::fold<>`. It is a primitive transform that takes a sequence, a state, and function, just like `std::accumulate()`. The three template parameters are transforms. The first yields the sequence of expressions over which to fold, the second yields the initial state of the fold, and the third is the function to apply at each iteration. The use of `proto::_` as the first parameter might have you confused. In addition to being Proto's wildcard, `proto::_` is also a primitive transform that returns the current expression, which (if it is a non-terminal) is a sequence of its child expressions.

Next, we need a function object that accepts a narrow string, a wide character buffer, and a `std::ctype<>` facet for doing the locale-specific stuff. It's fairly straightforward.


```
// A function object that writes a narrow string
// into a wide buffer.
struct WidenCopy : proto::callable
{
    typedef wchar_t *result_type;

    wchar_t *
    operator()(char const *str, wchar_t *buf, std::ctype<char> const &ct) const
    {
        for(; *str; ++str, ++buf)
            *buf = ct.widen(*str);
        return buf;
    }
};
```

Finally, we need some transforms that actually walk the concatenated string expression, widens the characters and writes them to a buffer. We will pass a `wchar_t*` as the state parameter and update it as we go. We'll also pass the `std::ctype<>` facet as the data parameter. It looks like this:

```
// Write concatenated strings into a buffer, widening
// them as we go.
struct StringCopy
: proto::or_<
    proto::when<
        proto::terminal<char[proto::N]>
        , WidenCopy(proto::_value, proto::_state, proto::_data)
    >
    , proto::when<
        proto::plus<StringCopy, StringCopy>
        , StringCopy(
            proto::_right
            , StringCopy(proto::_left, proto::_state, proto::_data)
            , proto::_data
        )
    >
>
{};
```

Let's look more closely at the transform associated with non-terminals:

```
StringCopy(
    proto::_right
    , StringCopy(proto::_left, proto::_state, proto::_data)
    , proto::_data
)
```

This bears a resemblance to the transform in the previous section that folded an expression tree into a list. First we recurse on the left child, writing its strings into the `wchar_t*` passed in as the state parameter. That returns the new value of the `wchar_t*`, which is passed as state while transforming the right child. Both invocations receive the same `std::ctype<>`, which is passed in as the data parameter.

With these pieces in our pocket, we can implement our concatenate-and-widen function as follows:


```

template<typename Expr>
void widen( Expr const &expr )
{
    // Make sure the expression conforms to our grammar
    BOOST_MPL_ASSERT(( proto::matches<Expr, StringLength> ));

    // Calculate the length of the string and allocate a buffer statically
    static std::size_t const length =
        boost::result_of<StringLength(Expr)>::type::value;
    wchar_t buffer[ length + 1 ] = {L'\0'};

    // Get the current ctype facet
    std::locale loc;
    std::ctype<char> const &ct(std::use_facet<std::ctype<char> >(loc));

    // Concatenate and widen the string expression
    StringCopy()(expr, &buffer[0], ct);

    // Write out the buffer.
    std::wcout << buffer << std::endl;
}

int main()
{
    widen( proto::lit("hello") + " " + "world" );
}

```

The above code displays:

```
hello world
```

This is a rather round-about way of demonstrating that you can pass extra data to a transform as a third parameter. There are no restrictions on what this parameter can be, and (unlike the state parameter) Proto will never mess with it.

Implicit Parameters to Primitive Transforms

Let's use the above example to illustrate some other niceties of Proto transforms. We've seen that grammars, when used as function objects, can accept up to 3 parameters, and that when using these grammars in callable transforms, you can also specify up to 3 parameters. Let's take another look at the transform associated with non-terminals above:

```

StringCopy(
    proto::_right
    , StringCopy(proto::_left, proto::_state, proto::_data)
    , proto::_data
)

```

Here we specify all three parameters to both invocations of the `StringCopy` grammar. But we don't have to specify all three. If we don't specify a third parameter, `proto::_data` is assumed. Likewise for the second parameter and `proto::_state`. So the above transform could have been written more simply as:

```

StringCopy(
    proto::_right
    , StringCopy(proto::_left)
)

```

The same is true for any primitive transform. The following are all equivalent:

Table 8. Implicit Parameters to Primitive Transforms

Equivalent Transforms
<code>proto::when<_, StringCopy></code>
<code>proto::when<_, StringCopy()></code>
<code>proto::when<_, StringCopy(_)></code>
<code>proto::when<_, StringCopy(_, proto::_state)></code>
<code>proto::when<_, StringCopy(_, proto::_state, proto::_data)></code>

**Note****Grammars Are Primitive Transforms Are Function Objects**

So far, we've said that all Proto grammars are function objects. But it's more accurate to say that Proto grammars are primitive transforms -- a special kind of function object that takes between 1 and 3 arguments, and that Proto knows to treat specially when used in a callable transform, as in the table above.

**Note****Not All Function Objects Are Primitive Transforms**

You might be tempted now to drop the `_state` and `_data` parameters to `WidenCopy(proto::_value, proto::_state, proto::_data)`. That would be an error. `WidenCopy` is just a plain function object, not a primitive transform, so you must specify all its arguments. We'll see later how to write your own primitive transforms.

Once you know that primitive transforms will always receive all three parameters -- expression, state, and data -- it makes things possible that wouldn't be otherwise. For instance, consider that for binary expressions, these two transforms are equivalent. Can you see why?

Table 9. Two Equivalent Transforms

Without <code>proto::fold<></code>	With <code>proto::fold<></code>
<pre>StringCopy(proto::_right , String↓ Copy(proto::_left, proto::_state, proto::_data) , proto::_data)</pre>	<pre>proto::fold<_, proto::_state, String↓ Copy></pre>

Proto's Built-In Transforms

Primitive transforms are the building blocks for more interesting composite transforms. Proto defines a bunch of generally useful primitive transforms. They are summarized below.

`proto::_value` Given a terminal expression, return the value of the terminal.

`proto::_child_c<>` Given a non-terminal expression, `proto::_child_c<N>` returns the *N*-th child.

<code>proto::_child</code>	A synonym for <code>proto::_child_c<0></code> .
<code>proto::_left</code>	A synonym for <code>proto::_child_c<0></code> .
<code>proto::_right</code>	A synonym for <code>proto::_child_c<1></code> .
<code>proto::_expr</code>	Returns the current expression unmodified.
<code>proto::_state</code>	Returns the current state unmodified.
<code>proto::_data</code>	Returns the current data unmodified.
<code>proto::call<></code>	For a given callable transform <i>CT</i> , <code>proto::call<CT></code> turns the callable transform into a primitive transform. This is useful for disambiguating callable transforms from object transforms, and also for working around compiler bugs with nested function types.
<code>proto::make<></code>	For a given object transform <i>OT</i> , <code>proto::make<OT></code> turns the object transform into a primitive transform. This is useful for disambiguating object transforms from callable transforms, and also for working around compiler bugs with nested function types.
<code>proto::_default<></code>	Given a grammar <i>G</i> , <code>proto::_default<G></code> evaluates the current node according to the standard C++ meaning of the operation the node represents. For instance, if the current node is a binary plus node, the two children will both be evaluated according to <i>G</i> and the results will be added and returned. The return type is deduced with the help of the Boost.Typeof library.
<code>proto::fold<></code>	Given three transforms <i>ET</i> , <i>ST</i> , and <i>FT</i> , <code>proto::fold<ET, ST, FT></code> first evaluates <i>ET</i> to obtain a Fusion sequence and <i>ST</i> to obtain an initial state for the fold, and then evaluates <i>FT</i> for each element in the sequence to generate the next state from the previous.
<code>proto::reverse_fold<></code>	Like <code>proto::fold<></code> , except the elements in the Fusion sequence are iterated in reverse order.
<code>proto::fold_tree<></code>	Like <code>proto::fold<ET, ST, FT></code> , except that the result of the <i>ET</i> transform is treated as an expression tree that is <i>flattened</i> to generate the sequence to be folded. Flattening an expression tree causes child nodes with the same tag type as the parent to be put into sequence. For instance, <code>a >> b >> c</code> would be flattened to the sequence <code>[a, b, c]</code> , and this is the sequence that would be folded.
<code>proto::reverse_fold_tree<></code>	Like <code>proto::fold_tree<></code> , except that the flattened sequence is iterated in reverse order.
<code>proto::lazy<></code>	A combination of <code>proto::make<></code> and <code>proto::call<></code> that is useful when the nature of the transform depends on the expression, state and/or data parameters. <code>proto::lazy<R(A0, A1...An)></code> first evaluates <code>proto::make<R(>></code> to compute a callable type <i>R2</i> . Then, it evaluates <code>proto::call<R2(A0, A1...An)></code> .

All Grammars Are Primitive Transforms

In addition to the above primitive transforms, all of Proto's grammar elements are also primitive transforms. Their behaviors are described below.

<code>proto::_</code>	Return the current expression unmodified.
<code>proto::or<></code>	For the specified set of alternate sub-grammars, find the one that matches the given expression and apply its associated transform.
<code>proto::and<></code>	For the given set of sub-grammars, apply all the associated transforms and return the result of the last.
<code>proto::not<></code>	Return the current expression unmodified.
<code>proto::if<></code>	Given three transforms, evaluate the first and treat the result as a compile-time Boolean value. If it is true, evaluate the second transform. Otherwise, evaluate the third.

<code>proto::switch_<></code>	As with <code>proto::or_<></code> , find the sub-grammar that matches the given expression and apply its associated transform.
<code>proto::terminal<></code>	Return the current terminal expression unmodified.
<code>proto::plus<></code> , <code>proto::nary_expr<></code> , et. al.	A Proto grammar that matches a non-terminal such as <code>proto::plus<G0, G1></code> , when used as a primitive transform, creates a new plus node where the left child is transformed according to <i>G0</i> and the right child with <i>G1</i> .

The Pass-Through Transform

Note the primitive transform associated with grammar elements such as `proto::plus<>` described above. They possess a so-called *pass-through* transform. The pass-through transform accepts an expression of a certain tag type (say, `proto::tag::plus`) and creates a new expression of the same tag type, where each child expression is transformed according to the corresponding child grammar of the pass-through transform. So for instance this grammar ...

```
proto::function< X, proto::vararg<Y> >
```

... matches function expressions where the first child matches the *X* grammar and the rest match the *Y* grammar. When used as a transform, the above grammar will create a new function expression where the first child is transformed according to *X* and the rest are transformed according to *Y*.

The following class templates in Proto can be used as grammars with pass-through transforms:

Table 10. Class Templates With Pass-Through Transforms

Templates with Pass-Through Transforms
<code>proto::unary_plus<></code>
<code>proto::negate<></code>
<code>proto::dereference<></code>
<code>proto::complement<></code>
<code>proto::address_of<></code>
<code>proto::logical_not<></code>
<code>proto::pre_inc<></code>
<code>proto::pre_dec<></code>
<code>proto::post_inc<></code>
<code>proto::post_dec<></code>
<code>proto::shift_left<></code>
<code>proto::shift_right<></code>
<code>proto::multiplies<></code>
<code>proto::divides<></code>
<code>proto::modulus<></code>
<code>proto::plus<></code>
<code>proto::minus<></code>
<code>proto::less<></code>
<code>proto::greater<></code>
<code>proto::less_equal<></code>
<code>proto::greater_equal<></code>
<code>proto::equal_to<></code>
<code>proto::not_equal_to<></code>
<code>proto::logical_or<></code>
<code>proto::logical_and<></code>
<code>proto::bitwise_and<></code>
<code>proto::bitwise_or<></code>

Templates with Pass-Through Transforms`proto::bitwise_xor<>``proto::comma<>``proto::mem_ptr<>``proto::assign<>``proto::shift_left_assign<>``proto::shift_right_assign<>``proto::multiplies_assign<>``proto::divides_assign<>``proto::modulus_assign<>``proto::plus_assign<>``proto::minus_assign<>``proto::bitwise_and_assign<>``proto::bitwise_or_assign<>``proto::bitwise_xor_assign<>``proto::subscript<>``proto::if_else_<>``proto::function<>``proto::unary_expr<>``proto::binary_expr<>``proto::nary_expr<>`**The Many Roles of Proto Operator Metafunctions**

We've seen templates such as `proto::terminal<>`, `proto::plus<>` and `proto::nary_expr<>` fill many roles. They are metafunction that generate expression types. They are grammars that match expression types. And they are primitive transforms. The following code samples show examples of each.

As Metafunctions ...


```
// proto::terminal<> and proto::plus<> are metafunctions
// that generate expression types:
typedef proto::terminal<int>::type int_;
typedef proto::plus<int_, int_>::type plus_;

int_ i = {42}, j = {24};
plus_ p = {i, j};
```

As Grammars ...

```
// proto::terminal<> and proto::plus<> are grammars that
// match expression types
struct Int : proto::terminal<int> {};
struct Plus : proto::plus<Int, Int> {};

BOOST_MPL_ASSERT(( proto::matches< int_, Int > ));
BOOST_MPL_ASSERT(( proto::matches< plus_, Plus > ));
```

As Primitive Transforms ...

```
// A transform that removes all unary_plus nodes in an expression
struct RemoveUnaryPlus
: proto::or_<
    proto::when<
        proto::unary_plus<RemoveUnaryPlus>
        , RemoveUnaryPlus(proto::_child)
    >
    // Use proto::terminal<> and proto::nary_expr<>
    // both as grammars and as primitive transforms.
    , proto::terminal<_>
    , proto::nary_expr<_, proto::vararg<RemoveUnaryPlus> >
>
{};

int main()
{
    proto::literal<int> i(0);

    proto::display_expr(
        +i - +(i - +i)
    );

    proto::display_expr(
        RemoveUnaryPlus()( +i - +(i - +i) )
    );
}
```

The above code displays the following, which shows that unary plus nodes have been stripped from the expression:


```
minus(
    unary_plus(
        terminal(0)
    )
    , unary_plus(
        minus(
            terminal(0)
            , unary_plus(
                terminal(0)
            )
        )
    )
)
minus(
    terminal(0)
    , minus(
        terminal(0)
        , terminal(0)
    )
)
```

Building Custom Primitive Transforms

In previous sections, we've seen how to compose larger transforms out of smaller transforms using function types. The smaller transforms from which larger transforms are composed are *primitive transforms*, and Proto provides a bunch of common ones such as `_child0` and `_value`. In this section we'll see how to author your own primitive transforms.



Note

There are a few reasons why you might want to write your own primitive transforms. For instance, your transform may be complicated, and composing it out of primitives becomes unwieldy. You might also need to work around compiler bugs on legacy compilers that make composing transforms using function types problematic. Finally, you might also decide to define your own primitive transforms to improve compile times. Since Proto can simply invoke a primitive transform directly without having to process arguments or differentiate callable transforms from object transforms, primitive transforms are more efficient.

Primitive transforms inherit from `proto::transform<>` and have a nested `impl<>` template that inherits from `proto::transform_impl<>`. For example, this is how Proto defines the `_child_c<N>` transform, which returns the *N*-th child of the current expression:


```

namespace boost { namespace proto
{
    // A primitive transform that returns N-th child
    // of the current expression.
    template<int N>
    struct _child_c : transform<_child_c<N> >
    {
        template<typename Expr, typename State, typename Data>
        struct impl : transform_impl<Expr, State, Data>
        {
            typedef
                typename result_of::child_c<Expr, N>::type
                result_type;

            result_type operator ()(
                typename impl::expr_param expr
                , typename impl::state_param state
                , typename impl::data_param data
            ) const
            {
                return proto::child_c<N>(expr);
            }
        };
    };

    // Note that _child_c<N> is callable, so that
    // it can be used in callable transforms, as:
    // _child_c<0>(_child_c<1>)
    template<int N>
    struct is_callable<_child_c<N> >
        : mpl::true_
    {};
}}

```

The `proto::transform<>` base class provides the `operator()` overloads and the nested `result<>` template that make your transform a valid function object. These are implemented in terms of the nested `impl<>` template you define.

The `proto::transform_impl<>` base class is a convenience. It provides some nested typedefs that are generally useful. They are specified in the table below:

Table 11. `proto::transform_impl<Expr, State, Data>` typedefs

typedef	Equivalent To
<code>expr</code>	<code>typename remove_reference<Expr>::type</code>
<code>state</code>	<code>typename remove_reference<State>::type</code>
<code>data</code>	<code>typename remove_reference<Data>::type</code>
<code>expr_param</code>	<code>typename add_reference<typename add_const<Expr>::type>::type</code>
<code>state_param</code>	<code>typename add_reference<typename add_const<State>::type>::type</code>
<code>data_param</code>	<code>typename add_reference<typename add_const<Data>::type>::type</code>

You'll notice that `_child_c::impl::operator()` takes arguments of types `expr_param`, `state_param`, and `data_param`. The typedefs make it easy to accept arguments by reference or const reference accordingly.

The only other interesting bit is the `is_callable<>` specialization, which will be described in the [next section](#).

Making Your Transform Callable

Transforms are typically of the form `proto::when< Something, R(A0,A1,...) >`. The question is whether `R` represents a function to call or an object to construct, and the answer determines how `proto::when<>` evaluates the transform. `proto::when<>` uses the `proto::is_callable<>` trait to disambiguate between the two. Proto does its best to guess whether a type is callable or not, but it doesn't always get it right. It's best to know the rules Proto uses, so that you know when you need to be more explicit.

For most types `R`, `proto::is_callable<R>` checks for inheritance from `proto::callable`. However, if the type `R` is a template specialization, Proto assumes that it is *not* callable *even if the template inherits from* `proto::callable`. We'll see why in a minute. Consider the following erroneous callable object:

```
// Proto can't tell this defines something callable!
template<typename T>
struct times2 : proto::callable
{
    typedef T result_type;

    T operator()(T i) const
    {
        return i * 2;
    }
};

// ERROR! This is not going to multiply the int by 2:
struct IntTimes2
: proto::when<
    proto::terminal<int>
    , times2<int>(proto::_value)
>
{};
```

The problem is that Proto doesn't know that `times2<int>` is callable, so rather than invoking the `times2<int>` function object, Proto will try to construct a `times2<int>` object and initialize it with an `int`. That will not compile.



Note

Why can't Proto tell that `times2<int>` is callable? After all, it inherits from `proto::callable`, and that is detectable, right? The problem is that merely asking whether some type `X<Y>` inherits from `callable` will cause the template `X<Y>` to be instantiated. That's a problem for a type like `std::vector<_value(_child1)>`. `std::vector<>` will not suffer to be instantiated with `_value(_child1)` as a template parameter. Since merely asking the question will sometimes result in a hard error, Proto can't ask; it has to assume that `X<Y>` represents an object to construct and not a function to call.

There are a couple of solutions to the `times2<int>` problem. One solution is to wrap the transform in `proto::call<>`. This forces Proto to treat `times2<int>` as callable:

```
// OK, calls times2<int>
struct IntTimes2
: proto::when<
    proto::terminal<int>
    , proto::call<times2<int>(proto::_value)>
>
{};
```

This can be a bit of a pain, because we need to wrap every use of `times2<int>`, which can be tedious and error prone, and makes our grammar cluttered and harder to read.

Another solution is to specialize `proto::is_callable<>` on our `times2<>` template:


```
namespace boost { namespace proto
{
    // Tell Proto that times2<> is callable
    template<typename T>
    struct is_callable<times2<T> >
        : mpl::true_
    {};
}}

// OK, times2<> is callable
struct IntTimes2
    : proto::when<
        proto::terminal<int>
        , times2<int>(proto::_value)
    >
{};
```

This is better, but still a pain because of the need to open Proto's namespace.

You could simply make sure that the callable type is not a template specialization. Consider the following:

```
// No longer a template specialization!
struct times2int : times2<int> {};

// OK, times2int is callable
struct IntTimes2
    : proto::when<
        proto::terminal<int>
        , times2int(proto::_value)
    >
{};
```

This works because now Proto can tell that `times2int` inherits (indirectly) from `proto::callable`. Any non-template types can be safely checked for inheritance because, as they are not templates, there is no worry about instantiation errors.

There is one last way to tell Proto that `times2<>` is callable. You could add an extra dummy template parameter that defaults to `proto::callable`:


```
// Proto will recognize this as callable
template<typename T, typename Callable = proto::callable>
struct times2 : proto::callable
{
    typedef T result_type;

    T operator()(T i) const
    {
        return i * 2;
    }
};

// OK, this works!
struct IntTimes2
: proto::when<
    proto::terminal<int>
    , times2<int>(proto::_value)
>
{};
```

Note that in addition to the extra template parameter, `times2<>` still inherits from `proto::callable`. That's not necessary in this example but it is good style because any types derived from `times2<>` (as `times2int` defined above) will still be considered callable.

Examples

A code example is worth a thousand words ...

Hello World: Building an Expression Template and Evaluating It

A trivial example which builds an expression template and evaluates it.


```
////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
#include <boost/typeof/std/ostream.hpp>
namespace proto = boost::proto;

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

template< typename Expr >
void evaluate( Expr const & expr )
{
    proto::default_context ctx;
    proto::eval(expr, ctx);
}

int main()
{
    evaluate( cout_ << "hello" << ', ' << " world" );
    return 0;
}
```

Calc1: Defining an Evaluation Context

A simple example that builds a miniature domain-specific embedded language for lazy arithmetic expressions, with TR1 bind-style argument placeholders.


```

// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This is a simple example of how to build an arithmetic expression
// evaluator with placeholders.

#include <iostream>
#include <boost/mpl/int.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
using proto::_;

template<int I> struct placeholder {};

// Define some placeholders
proto::terminal< placeholder< 1 > >::type const _1 = {{{}}};
proto::terminal< placeholder< 2 > >::type const _2 = {{{}}};

// Define a calculator context, for evaluating arithmetic expressions
struct calculator_context
: proto::callable_context< calculator_context const >
{
    // The values bound to the placeholders
    double d[2];

    // The result of evaluating arithmetic expressions
    typedef double result_type;

    explicit calculator_context(double d1 = 0., double d2 = 0.)
    {
        d[0] = d1;
        d[1] = d2;
    }

    // Handle the evaluation of the placeholder terminals
    template<int I>
    double operator ()(proto::tag::terminal, placeholder<I>) const
    {
        return d[ I - 1 ];
    }
};

template<typename Expr>
double evaluate( Expr const &expr, double d1 = 0., double d2 = 0. )
{
    // Create a calculator context with d1 and d2 substituted for _1 and _2
    calculator_context const ctx(d1, d2);

    // Evaluate the calculator expression with the calculator_context
    return proto::eval(expr, ctx);
}

int main()
{
    // Displays "5"
    std::cout << evaluate( _1 + 2.0, 3.0 ) << std::endl;

    // Displays "6"
    std::cout << evaluate( _1 * _2, 3.0, 2.0 ) << std::endl;
}

```



```

// Displays "0.5"
std::cout << evaluate( (_1 - _2) / _2, 3.0, 2.0 ) << std::endl;

return 0;
}

```

Calc2: Adding Members Using `proto::extends<>`

An extension of the Calc1 example that uses `proto::extends<>` to make calculator expressions valid function objects that can be used with STL algorithms.

```

// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This example enhances the simple arithmetic expression evaluator
// in calc1.cpp by using proto::extends to make arithmetic
// expressions immediately evaluable with operator (), a-la a
// function object

#include <iostream>
#include <boost/mpl/int.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
using proto::_;

// Will be used to define the placeholders _1 and _2
template<int I> struct placeholder {};

// For expressions in the calculator domain, operator ()
// will be special; it will evaluate the expression.
struct calculator_domain;

// Define a calculator context, for evaluating arithmetic expressions
// (This is as before, in calc1.cpp)
struct calculator_context
: proto::callable_context< calculator_context const >
{
    // The values bound to the placeholders
    double d[2];

    // The result of evaluating arithmetic expressions
    typedef double result_type;

    explicit calculator_context(double d1 = 0., double d2 = 0.)
    {
        d[0] = d1;
        d[1] = d2;
    }

    // Handle the evaluation of the placeholder terminals
    template<int I>
    double operator ()(proto::tag::terminal, placeholder<I>) const
    {
        return d[ I - 1 ];
    }
};

```



```

// Wrap all calculator expressions in this type, which defines
// operator () to evaluate the expression.
template<typename Expr>
struct calculator_expression
: proto::extends<Expr, calculator_expression<Expr>, calculator_domain>
{
    typedef
        proto::extends<Expr, calculator_expression<Expr>, calculator_domain>
        base_type;

    explicit calculator_expression(Expr const &expr = Expr())
        : base_type(expr)
    {}

    BOOST_PROTO_EXTENDS_USING_ASSIGN(calculator_expression)

    // Override operator () to evaluate the expression
    double operator ()() const
    {
        calculator_context const ctx;
        return proto::eval(*this, ctx);
    }

    double operator ()(double d1) const
    {
        calculator_context const ctx(d1);
        return proto::eval(*this, ctx);
    }

    double operator ()(double d1, double d2) const
    {
        calculator_context const ctx(d1, d2);
        return proto::eval(*this, ctx);
    }
};

// Tell proto how to generate expressions in the calculator_domain
struct calculator_domain
: proto::domain<proto::generator<calculator_expression> >
{};

// Define some placeholders (notice they're wrapped in calculator_expression<>)
calculator_expression<proto::terminal< placeholder< 1 > >::type> const _1;
calculator_expression<proto::terminal< placeholder< 2 > >::type> const _2;

// Now, our arithmetic expressions are immediately executable function objects:
int main()
{
    // Displays "5"
    std::cout << (_1 + 2.0)( 3.0 ) << std::endl;

    // Displays "6"
    std::cout << ( _1 * _2 )( 3.0, 2.0 ) << std::endl;
}

```



```

// Displays "0.5"
std::cout << ( (_1 - _2) / _2 )( 3.0, 2.0 ) << std::endl;

return 0;
}

```

Calc3: Defining a Simple Transform

An extension of the Calc2 example that uses a Proto transform to calculate the arity of a calculator expression and statically assert that the correct number of arguments are passed.

```

// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This example enhances the arithmetic expression evaluator
// in calc2.cpp by using a proto transform to calculate the
// number of arguments an expression requires and using a
// compile-time assert to guarantee that the right number of
// arguments are actually specified.

#include <iostream>
#include <boost/mpl/int.hpp>
#include <boost/mpl/assert.hpp>
#include <boost/mpl/min_max.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
#include <boost/proto/transform.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
using proto::_;

// Will be used to define the placeholders _1 and _2
template<typename I> struct placeholder : I {};

// This grammar basically says that a calculator expression is one of:
// - A placeholder terminal
// - Some other terminal
// - Some non-terminal whose children are calculator expressions
// In addition, it has transforms that say how to calculate the
// expression arity for each of the three cases.
struct CalculatorGrammar
: proto::or_<

    // placeholders have a non-zero arity ...
    proto::when< proto::terminal< placeholder<_> >, proto::_value >

    // Any other terminals have arity 0 ...
    , proto::when< proto::terminal<_>, mpl::int_<0>() >

    // For any non-terminals, find the arity of the children and
    // take the maximum. This is recursive.
    , proto::when< proto::nary_expr<_, proto::vararg<_> >
        , proto::fold<_, mpl::int_<0>(), mpl::max<CalculatorGrammar, proto::_state>() > >

    >
{};

// Simple wrapper for calculating a calculator expression's arity.
// It specifies mpl::int_<0> as the initial state. The data, which

```



```

// is not used, is mpl::void_.
template<typename Expr>
struct calculator_arity
: boost::result_of<CalculatorGrammar(Expr, mpl::int_<0>, mpl::void_)>
{};

// For expressions in the calculator domain, operator ()
// will be special; it will evaluate the expression.
struct calculator_domain;

// Define a calculator context, for evaluating arithmetic expressions
// (This is as before, in calc1.cpp and calc2.cpp)
struct calculator_context
: proto::callable_context< calculator_context const >
{
    // The values bound to the placeholders
    double d[2];

    // The result of evaluating arithmetic expressions
    typedef double result_type;

    explicit calculator_context(double d1 = 0., double d2 = 0.)
    {
        d[0] = d1;
        d[1] = d2;
    }

    // Handle the evaluation of the placeholder terminals
    template<typename I>
    double operator ()(proto::tag::terminal, placeholder<I>) const
    {
        return d[ I() - 1 ];
    }
};

// Wrap all calculator expressions in this type, which defines
// operator () to evaluate the expression.
template<typename Expr>
struct calculator_expression
: proto::extends<Expr, calculator_expression<Expr>, calculator_domain>
{
    typedef
        proto::extends<Expr, calculator_expression<Expr>, calculator_domain>
        base_type;

    explicit calculator_expression(Expr const &expr = Expr())
    : base_type(expr)
    {}

    BOOST_PROTO_EXTENDS_USING_ASSIGN(calculator_expression)

    // Override operator () to evaluate the expression
    double operator ()() const
    {
        // Assert that the expression has arity 0
        BOOST_MPL_ASSERT_RELATION(0, ==, calculator_arity<Expr>::type::value);
        calculator_context const ctx;
        return proto::eval(*this, ctx);
    }

    double operator ()(double d1) const
    {
        // Assert that the expression has arity 1

```



```

    BOOST_MPL_ASSERT_RELATION(1, ==, calculator_arity<Expr>::type::value);
    calculator_context const ctx(d1);
    return proto::eval(*this, ctx);
}

double operator()(double d1, double d2) const
{
    // Assert that the expression has arity 2
    BOOST_MPL_ASSERT_RELATION(2, ==, calculator_arity<Expr>::type::value);
    calculator_context const ctx(d1, d2);
    return proto::eval(*this, ctx);
}
};

// Tell proto how to generate expressions in the calculator_domain
struct calculator_domain
: proto::domain<proto::generator<calculator_expression> >
{};

// Define some placeholders (notice they're wrapped in calculator_expression<>)
calculator_expression<proto::terminal< placeholder< mpl::int_<1> > >::type> const _1;
calculator_expression<proto::terminal< placeholder< mpl::int_<2> > >::type> const _2;

// Now, our arithmetic expressions are immediately executable function objects:
int main()
{
    // Displays "5"
    std::cout << (_1 + 2.0)( 3.0 ) << std::endl;

    // Displays "6"
    std::cout << ( _1 * _2 )( 3.0, 2.0 ) << std::endl;

    // Displays "0.5"
    std::cout << ( (_1 - _2) / _2 )( 3.0, 2.0 ) << std::endl;

    // This won't compile because the arity of the
    // expression doesn't match the number of arguments
    // ( (_1 - _2) / _2 )( 3.0 );

    return 0;
}

```

Lazy Vector: Controlling Operator Overloads

This example constructs a mini-library for linear algebra, using expression templates to eliminate the need for temporaries when adding vectors of numbers.

This example uses a domain with a grammar to prune the set of overloaded operators. Only those operators that produce valid lazy vector expressions are allowed.


```

////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This example constructs a mini-library for linear algebra, using
// expression templates to eliminate the need for temporaries when
// adding vectors of numbers.
//
// This example uses a domain with a grammar to prune the set
// of overloaded operators. Only those operators that produce
// valid lazy vector expressions are allowed.

#include <vector>
#include <iostream>
#include <boost/mpl/int.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
using proto::_;

// This grammar describes which lazy vector expressions
// are allowed; namely, vector terminals and addition
// and subtraction of lazy vector expressions.
struct LazyVectorGrammar
: proto::or_<
    proto::terminal< std::vector<_> >
    , proto::plus< LazyVectorGrammar, LazyVectorGrammar >
    , proto::minus< LazyVectorGrammar, LazyVectorGrammar >
>
{};

// Expressions in the lazy vector domain must conform
// to the lazy vector grammar
struct lazy_vector_domain;

// Here is an evaluation context that indexes into a lazy vector
// expression, and combines the result.
template<typename Size = std::size_t>
struct lazy_subscript_context
{
    lazy_subscript_context(Size subscript)
    : subscript_(subscript)
    {}

    // Use default_eval for all the operations ...
    template<typename Expr, typename Tag = typename Expr::proto_tag>
    struct eval
    : proto::default_eval<Expr, lazy_subscript_context>
    {};

    // ... except for terminals, which we index with our subscript
    template<typename Expr>
    struct eval<Expr, proto::tag::terminal>
    {
        typedef typename proto::result_of::value<Expr>::type::value_type result_type;

        result_type operator ()( Expr const & expr, lazy_subscript_context & ctx ) const
        {
            return proto::value( expr )[ ctx.subscript_ ];
        }
    };
};

```



```

    Size subscript_;
};

// Here is the domain-specific expression wrapper, which overrides
// operator [] to evaluate the expression using the lazy_subscript_context.
template<typename Expr>
struct lazy_vector_expr
: proto::extends<Expr, lazy_vector_expr<Expr>, lazy_vector_domain>
{
    typedef proto::extends<Expr, lazy_vector_expr<Expr>, lazy_vector_domain> base_type;

    lazy_vector_expr( Expr const & expr = Expr() )
    : base_type( expr )
    {}

    // Use the lazy_subscript_context<> to implement subscripting
    // of a lazy vector expression tree.
    template< typename Size >
    typename proto::result_of::eval< Expr, lazy_subscript_context<Size> >::type
    operator [] ( Size subscript ) const
    {
        lazy_subscript_context<Size> ctx(subscript);
        return proto::eval(*this, ctx);
    }
};

// Here is our lazy_vector terminal, implemented in terms of lazy_vector_expr
template< typename T >
struct lazy_vector
: lazy_vector_expr< typename proto::terminal< std::vector<T> >::type >
{
    typedef typename proto::terminal< std::vector<T> >::type expr_type;

    lazy_vector( std::size_t size = 0, T const & value = T() )
    : lazy_vector_expr<expr_type>( expr_type::make( std::vector<T>( size, value ) ) )
    {}

    // Here we define a += operator for lazy vector terminals that
    // takes a lazy vector expression and indexes it. expr[i] here
    // uses lazy_subscript_context<> under the covers.
    template< typename Expr >
    lazy_vector &operator += (Expr const & expr)
    {
        std::size_t size = proto::value(*this).size();
        for(std::size_t i = 0; i < size; ++i)
        {
            proto::value(*this)[i] += expr[i];
        }
        return *this;
    }
};

// Tell proto that in the lazy_vector_domain, all
// expressions should be wrapped in lazy_vector_expr<>
struct lazy_vector_domain
: proto::domain<proto::generator<lazy_vector_expr>, LazyVectorGrammar>
{};

int main()
{
    // lazy_vectors with 4 elements each.
    lazy_vector< double > v1( 4, 1.0 ), v2( 4, 2.0 ), v3( 4, 3.0 );

```



```
// Add two vectors lazily and get the 2nd element.
double d1 = ( v2 + v3 )[ 2 ]; // Look ma, no temporaries!
std::cout << d1 << std::endl;

// Subtract two vectors and add the result to a third vector.
v1 += v2 - v3; // Still no temporaries!
std::cout << '{' << v1[0] << ',' << v1[1]
          << ',' << v1[2] << ',' << v1[3] << '}' << std::endl;

// This expression is disallowed because it does not conform
// to the LazyVectorGrammar
//(v2 + v3) += v1;

return 0;
}
```

RGB: Type Manipulations with Proto Transforms

This is a simple example of doing arbitrary type manipulations with Proto transforms. It takes some expression involving primary colors and combines the colors according to arbitrary rules. It is a port of the RGB example from [PETE](#).


```

////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This is a simple example of doing arbitrary type manipulations with proto
// transforms. It takes some expression involving primary colors and combines
// the colors according to arbitrary rules. It is a port of the RGB example
// from PETE (http://www.codesourcery.com/pooma/download.html).

#include <iostream>
#include <boost/proto/core.hpp>
#include <boost/proto/transform.hpp>
namespace proto = boost::proto;

struct RedTag
{
    friend std::ostream &operator <<(std::ostream &sout, RedTag)
    {
        return sout << "This expression is red.";
    }
};

struct BlueTag
{
    friend std::ostream &operator <<(std::ostream &sout, BlueTag)
    {
        return sout << "This expression is blue.";
    }
};

struct GreenTag
{
    friend std::ostream &operator <<(std::ostream &sout, GreenTag)
    {
        return sout << "This expression is green.";
    }
};

typedef proto::terminal<RedTag>::type RedT;
typedef proto::terminal<BlueTag>::type BlueT;
typedef proto::terminal<GreenTag>::type GreenT;

struct Red;
struct Blue;
struct Green;

////////////////////////////////////
// A transform that produces new colors according to some arbitrary rules:
// red & green give blue, red & blue give green, blue and green give red.
struct Red
: proto::or_<
    proto::plus<Green, Blue>
    , proto::plus<Blue, Green>
    , proto::plus<Red, Red>
    , proto::terminal<RedTag>
>
{};

struct Green
: proto::or_<
    proto::plus<Red, Blue>
    , proto::plus<Blue, Red>

```



```
        , proto::plus<Green, Green>
        , proto::terminal<GreenTag>
    >
    {};
```

```
struct Blue
: proto::or_<
    proto::plus<Red, Green>
    , proto::plus<Green, Red>
    , proto::plus<Blue, Blue>
    , proto::terminal<BlueTag>
>
{};
```

```
struct RGB
: proto::or_<
    proto::when< Red, RedTag() >
    , proto::when< Blue, BlueTag() >
    , proto::when< Green, GreenTag() >
>
{};
```

```
template<typename Expr>
void printColor(Expr const & expr)
{
    int i = 0; // dummy state and data parameter, not used
    std::cout << RGB()(expr, i, i) << std::endl;
}
```

```
int main()
{
    printColor(RedT() + GreenT());
    printColor(RedT() + GreenT() + BlueT());
    printColor(RedT() + (GreenT() + BlueT()));

    return 0;
}
```

TArray: A Simple Linear Algebra Library

This example constructs a mini-library for linear algebra, using expression templates to eliminate the need for temporaries when adding arrays of numbers. It duplicates the TArray example from [PETE](#).


```

////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This example constructs a mini-library for linear algebra, using
// expression templates to eliminate the need for temporaries when
// adding arrays of numbers. It duplicates the TArray example from
// PETE (http://www.codesourcery.com/pooma/download.html)

#include <iostream>
#include <boost/mpl/int.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
using proto::_;

// This grammar describes which TArray expressions
// are allowed; namely, int and array terminals
// plus, minus, multiplies and divides of TArray expressions.
struct TArrayGrammar
: proto::or_<
    proto::terminal< int >
    , proto::terminal< int[3] >
    , proto::plus< TArrayGrammar, TArrayGrammar >
    , proto::minus< TArrayGrammar, TArrayGrammar >
    , proto::multiplies< TArrayGrammar, TArrayGrammar >
    , proto::divides< TArrayGrammar, TArrayGrammar >
>
{};

template<typename Expr>
struct TArrayExpr;

// Tell proto that in the TArrayDomain, all
// expressions should be wrapped in TArrayExpr<> and
// must conform to the TArrayGrammar
struct TArrayDomain
: proto::domain<proto::generator<TArrayExpr>, TArrayGrammar>
{};

// Here is an evaluation context that indexes into a TArray
// expression, and combines the result.
struct TArraySubscriptCtx
: proto::callable_context< TArraySubscriptCtx const >
{
    typedef int result_type;

    TArraySubscriptCtx(std::ptrdiff_t i)
    : i_(i)
    {}

    // Index array terminals with our subscript. Everything
    // else will be handled by the default evaluation context.
    int operator ()(proto::tag::terminal, int const (&data)[3]) const
    {
        return data[this->i_];
    }

    std::ptrdiff_t i_;
};

```



```

// Here is an evaluation context that prints a TArray expression.
struct TArrayPrintCtx
: proto::callable_context< TArrayPrintCtx const >
{
    typedef std::ostream &result_type;

    TArrayPrintCtx() {}

    std::ostream &operator ()(proto::tag::terminal, int i) const
    {
        return std::cout << i;
    }

    std::ostream &operator ()(proto::tag::terminal, int const (&arr)[3]) const
    {
        return std::cout << '{' << arr[0] << ", " << arr[1] << ", " << arr[2] << '>';
    }

    template<typename L, typename R>
    std::ostream &operator ()(proto::tag::plus, L const &l, R const &r) const
    {
        return std::cout << '(' << l << " + " << r << ')';
    }

    template<typename L, typename R>
    std::ostream &operator ()(proto::tag::minus, L const &l, R const &r) const
    {
        return std::cout << '(' << l << " - " << r << ')';
    }

    template<typename L, typename R>
    std::ostream &operator ()(proto::tag::multiplies, L const &l, R const &r) const
    {
        return std::cout << l << " * " << r;
    }

    template<typename L, typename R>
    std::ostream &operator ()(proto::tag::divides, L const &l, R const &r) const
    {
        return std::cout << l << " / " << r;
    }
};

// Here is the domain-specific expression wrapper, which overrides
// operator [] to evaluate the expression using the TArraySubscriptCtx.
template<typename Expr>
struct TArrayExpr
: proto::extends<Expr, TArrayExpr<Expr>, TArrayDomain>
{
    typedef proto::extends<Expr, TArrayExpr<Expr>, TArrayDomain> base_type;

    TArrayExpr( Expr const & expr = Expr() )
    : base_type( expr )
    {}

    // Use the TArraySubscriptCtx to implement subscripting
    // of a TArray expression tree.
    int operator [] ( std::ptrdiff_t i ) const
    {
        TArraySubscriptCtx const ctx(i);
        return proto::eval(*this, ctx);
    }
}

```



```

// Use the TArrayPrintCtx to display a TArray expression tree.
friend std::ostream &operator <<(std::ostream &sout, TArrayExpr<Expr> const &expr)
{
    TArrayPrintCtx const ctx;
    return proto::eval(expr, ctx);
}

};

// Here is our TArray terminal, implemented in terms of TArrayExpr
// It is basically just an array of 3 integers.
struct TArray
: TArrayExpr< proto::terminal< int[3] >::type >
{
    explicit TArray( int i = 0, int j = 0, int k = 0 )
    {
        (*this)[0] = i;
        (*this)[1] = j;
        (*this)[2] = k;
    }

    // Here we override operator [] to give read/write access to
    // the elements of the array. (We could use the TArrayExpr
    // operator [] if we made the subscript context smarter about
    // returning non-const reference when appropriate.)
    int &operator [](std::ptrdiff_t i)
    {
        return proto::value(*this)[i];
    }

    int const &operator [](std::ptrdiff_t i) const
    {
        return proto::value(*this)[i];
    }

    // Here we define a operator = for TArray terminals that
    // takes a TArray expression.
    template< typename Expr >
    TArray &operator =(Expr const & expr)
    {
        // proto::as_expr<TArrayDomain>(expr) is the same as
        // expr unless expr is an integer, in which case it
        // is made into a TArrayExpr terminal first.
        return this->assign(proto::as_expr<TArrayDomain>(expr));
    }

    template< typename Expr >
    TArray &printAssign(Expr const & expr)
    {
        *this = expr;
        std::cout << *this << " = " << expr << std::endl;
        return *this;
    }
}

private:
    template< typename Expr >
    TArray &assign(Expr const & expr)
    {
        // expr[i] here uses TArraySubscriptCtx under the covers.
        (*this)[0] = expr[0];
        (*this)[1] = expr[1];
        (*this)[2] = expr[2];
        return *this;
    }
}

```



```
};

int main()
{
    TArray a(3,1,2);

    TArray b;

    std::cout << a << std::endl;
    std::cout << b << std::endl;

    b[0] = 7; b[1] = 33; b[2] = -99;

    TArray c(a);

    std::cout << c << std::endl;

    a = 0;

    std::cout << a << std::endl;
    std::cout << b << std::endl;
    std::cout << c << std::endl;

    a = b + c;

    std::cout << a << std::endl;

    a.printAssign(b+c*(b + 3*c));

    return 0;
}
```

Vec3: Computing With Transforms and Contexts

This is a simple example using `proto::extends<>` to extend a terminal type with additional behaviors, and using custom contexts and `proto::eval()` for evaluating expressions. It is a port of the Vec3 example from [PETE](#).


```

////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This is a simple example using proto::extends to extend a terminal type with
// additional behaviors, and using custom contexts and proto::eval for
// evaluating expressions. It is a port of the Vec3 example
// from PETE (http://www.codesourcery.com/pooma/download.html).

#include <iostream>
#include <functional>
#include <boost/assert.hpp>
#include <boost/mpl/int.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
#include <boost/proto/proto_typeof.hpp>
#include <boost/proto/transform.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
using proto::_;

// Here is an evaluation context that indexes into a Vec3
// expression, and combines the result.
struct Vec3SubscriptCtx
: proto::callable_context< Vec3SubscriptCtx const >
{
    typedef int result_type;

    Vec3SubscriptCtx(int i)
        : i_(i)
    {}

    // Index array terminals with our subscript. Everything
    // else will be handled by the default evaluation context.
    int operator ()(proto::tag::terminal, int const (&arr)[3]) const
    {
        return arr[this->i_];
    }

    int i_;
};

// Here is an evaluation context that counts the number
// of Vec3 terminals in an expression.
struct CountLeavesCtx
: proto::callable_context< CountLeavesCtx, proto::null_context >
{
    CountLeavesCtx()
        : count(0)
    {}

    typedef void result_type;

    void operator ()(proto::tag::terminal, int const (&)[3])
    {
        ++this->count;
    }

    int count;
};

struct iplus : std::plus<int>, proto::callable {};

```



```

// Here is a transform that does the same thing as the above context.
// It demonstrates the use of the std::plus<> function object
// with the fold transform. With minor modifications, this
// transform could be used to calculate the leaf count at compile
// time, rather than at runtime.
struct CountLeaves
: proto::or_<
    // match a Vec3 terminal, return 1
    proto::when<proto::terminal<int[3]>, mpl::int_<1>() >
    // match a terminal, return int() (which is 0)
    , proto::when<proto::terminal<_>, int() >
    // fold everything else, using std::plus<> to add
    // the leaf count of each child to the accumulated state.
    , proto::otherwise< proto::fold<_, int(), iplus(CountLeaves, proto::_state) > >
>
{};

// Here is the Vec3 struct, which is a vector of 3 integers.
struct Vec3
: proto::extends<proto::terminal<int[3]>::type, Vec3>
{
    explicit Vec3(int i=0, int j=0, int k=0)
    {
        (*this)[0] = i;
        (*this)[1] = j;
        (*this)[2] = k;
    }

    int &operator [](int i)
    {
        return proto::value(*this)[i];
    }

    int const &operator [](int i) const
    {
        return proto::value(*this)[i];
    }

    // Here we define a operator = for Vec3 terminals that
    // takes a Vec3 expression.
    template< typename Expr >
    Vec3 &operator =(Expr const & expr)
    {
        typedef Vec3SubscriptCtx const CVec3SubscriptCtx;
        (*this)[0] = proto::eval(proto::as_expr(expr), CVec3SubscriptCtx(0));
        (*this)[1] = proto::eval(proto::as_expr(expr), CVec3SubscriptCtx(1));
        (*this)[2] = proto::eval(proto::as_expr(expr), CVec3SubscriptCtx(2));
        return *this;
    }

    void print() const
    {
        std::cout << '{' << (*this)[0]
                    << ", " << (*this)[1]
                    << ", " << (*this)[2]
                    << '}' << std::endl;
    }
};

// The count_leaves() function uses the CountLeaves transform and
// to count the number of leaves in an expression.
template<typename Expr>

```



```
int count_leaves(Expr const &expr)
{
    // Count the number of Vec3 terminals using the
    // CountLeavesCtx evaluation context.
    CountLeavesCtx ctx;
    proto::eval(expr, ctx);

    // This is another way to count the leaves using a transform.
    int i = 0;
    BOOST_ASSERT( CountLeaves()(expr, i, i) == ctx.count );

    return ctx.count;
}

int main()
{
    Vec3 a, b, c;

    c = 4;

    b[0] = -1;
    b[1] = -2;
    b[2] = -3;

    a = b + c;

    a.print();

    Vec3 d;
    BOOST_PROTO_AUTO(expr1, b + c);
    d = expr1;
    d.print();

    int num = count_leaves(expr1);
    std::cout << num << std::endl;

    BOOST_PROTO_AUTO(expr2, b + 3 * c);
    num = count_leaves(expr2);
    std::cout << num << std::endl;

    BOOST_PROTO_AUTO(expr3, b + c * d);
    num = count_leaves(expr3);
    std::cout << num << std::endl;

    return 0;
}
```

Vector: Adapting a Non-Proto Terminal Type

This is an example of using `BOOST_PROTO_DEFINE_OPERATORS()` to Protify expressions using `std::vector<>`, a non-Proto type. It is a port of the Vector example from [PETE](#).


```

////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This is an example of using BOOST_PROTO_DEFINE_OPERATORS to Protofy
// expressions using std::vector<>, a non-boost type. It is a port of the
// Vector example from PETE (http://www.codesourcery.com/pooma/download.html).

#include <vector>
#include <iostream>
#include <stdexcept>
#include <boost/mpl/bool.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/debug.hpp>
#include <boost/proto/context.hpp>
#include <boost/utility/enable_if.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
using proto::_;

template<typename Expr>
struct VectorExpr;

// Here is an evaluation context that indexes into a std::vector
// expression and combines the result.
struct VectorSubscriptCtx
{
    VectorSubscriptCtx(std::size_t i)
        : i_(i)
    {}

    // Unless this is a vector terminal, use the
    // default evaluation context
    template<typename Expr, typename EnableIf = void>
    struct eval
        : proto::default_eval<Expr, VectorSubscriptCtx const>
    {};

    // Index vector terminals with our subscript.
    template<typename Expr>
    struct eval<
        Expr
        , typename boost::enable_if<
            proto::matches<Expr, proto::terminal<std::vector<_, _> > >
        >::type
    >
    {
        typedef typename proto::result_of::value<Expr>::type::value_type result_type;

        result_type operator ()(Expr &expr, VectorSubscriptCtx const &ctx) const
        {
            return proto::value(expr)[ctx.i_];
        }
    };

    std::size_t i_;
};

// Here is an evaluation context that verifies that all the
// vectors in an expression have the same size.
struct VectorSizeCtx
{

```



```

VectorSizeCtx(std::size_t size)
    : size_(size)
{}

// Unless this is a vector terminal, use the
// null evaluation context
template<typename Expr, typename EnableIf = void>
struct eval
    : proto::null_eval<Expr, VectorSizeCtx const>
{};

// Index array terminals with our subscript. Everything
// else will be handled by the default evaluation context.
template<typename Expr>
struct eval<
    Expr
    , typename boost::enable_if<
        proto::matches<Expr, proto::terminal<std::vector<_, _> > >
        >::type
    >
{
    typedef void result_type;

    result_type operator ()(Expr &expr, VectorSizeCtx const &ctx) const
    {
        if(ctx.size_ != proto::value(expr).size())
        {
            throw std::runtime_error("LHS and RHS are not compatible");
        }
    }
};

std::size_t size_;
};

// A grammar which matches all the assignment operators,
// so we can easily disable them.
struct AssignOps
    : proto::switch_<struct AssignOpsCases>
{};

// Here are the cases used by the switch_ above.
struct AssignOpsCases
{
    template<typename Tag, int D = 0> struct case_ : proto::not_<_> {};

    template<int D> struct case_< proto::tag::plus_assign, D > : _ {};
    template<int D> struct case_< proto::tag::minus_assign, D > : _ {};
    template<int D> struct case_< proto::tag::multiplies_assign, D > : _ {};
    template<int D> struct case_< proto::tag::divides_assign, D > : _ {};
    template<int D> struct case_< proto::tag::modulus_assign, D > : _ {};
    template<int D> struct case_< proto::tag::shift_left_assign, D > : _ {};
    template<int D> struct case_< proto::tag::shift_right_assign, D > : _ {};
    template<int D> struct case_< proto::tag::bitwise_and_assign, D > : _ {};
    template<int D> struct case_< proto::tag::bitwise_or_assign, D > : _ {};
    template<int D> struct case_< proto::tag::bitwise_xor_assign, D > : _ {};
};

// A vector grammar is a terminal or some op that is not an
// assignment op. (Assignment will be handled specially.)
struct VectorGrammar
    : proto::or_<
        proto::terminal<_>

```



```

    , proto::and_<proto::nary_expr<_, proto::vararg<VectorGrammar> >, proto::not_<AssignOps> >
    >
};

// Expressions in the vector domain will be wrapped in VectorExpr<>
// and must conform to the VectorGrammar
struct VectorDomain
    : proto::domain<proto::generator<VectorExpr>, VectorGrammar>
{};

// Here is VectorExpr, which extends a proto expr type by
// giving it an operator [] which uses the VectorSubscriptCtx
// to evaluate an expression with a given index.
template<typename Expr>
struct VectorExpr
    : proto::extends<Expr, VectorExpr<Expr>, VectorDomain>
{
    explicit VectorExpr(Expr const &expr)
        : proto::extends<Expr, VectorExpr<Expr>, VectorDomain>(expr)
    {}

    // Use the VectorSubscriptCtx to implement subscripting
    // of a Vector expression tree.
    typename proto::result_of::eval<Expr const, VectorSubscriptCtx const>::type
    operator [] ( std::size_t i ) const
    {
        VectorSubscriptCtx ctx(i);
        return proto::eval(*this, ctx);
    }
};

// Define a trait type for detecting vector terminals, to
// be used by the BOOST_PROTO_DEFINE_OPERATORS macro below.
template<typename T>
struct IsVector
    : mpl::false_
{};

template<typename T, typename A>
struct IsVector<std::vector<T, A> >
    : mpl::true_
{};

namespace VectorOps
{
    // This defines all the overloads to make expressions involving
    // std::vector to build expression templates.
    BOOST_PROTO_DEFINE_OPERATORS(IsVector, VectorDomain)

    typedef VectorSubscriptCtx const CVectorSubscriptCtx;

    // Assign to a vector from some expression.
    template<typename T, typename A, typename Expr>
    std::vector<T, A> &assign(std::vector<T, A> &arr, Expr const &expr)
    {
        VectorSizeCtx ctx(size(arr.size()));
        proto::eval(proto::as_expr<VectorDomain>(expr), ctx); // will throw if the sizes don't match
        for(std::size_t i = 0; i < arr.size(); ++i)
        {
            arr[i] = proto::as_expr<VectorDomain>(expr)[i];
        }
        return arr;
    }
}

```



```

}

// Add-assign to a vector from some expression.
template<typename T, typename A, typename Expr>
std::vector<T, A> &operator +=(std::vector<T, A> &arr, Expr const &expr)
{
    VectorSizeCtx const size(arr.size());
    proto::eval(proto::as_expr<VectorDomain>(expr), size); // will throw if the sizes don't
match
    for(std::size_t i = 0; i < arr.size(); ++i)
    {
        arr[i] += proto::as_expr<VectorDomain>(expr)[i];
    }
    return arr;
}
}

int main()
{
    using namespace VectorOps;

    int i;
    const int n = 10;
    std::vector<int> a,b,c,d;
    std::vector<double> e(n);

    for (i = 0; i < n; ++i)
    {
        a.push_back(i);
        b.push_back(2*i);
        c.push_back(3*i);
        d.push_back(i);
    }

    VectorOps::assign(b, 2);
    VectorOps::assign(d, a + b * c);
    a += if_else(d < 30, b, c);

    VectorOps::assign(e, c);
    e += e - 4 / (c + 1);

    for (i = 0; i < n; ++i)
    {
        std::cout
            << " a(" << i << ") = " << a[i]
            << " b(" << i << ") = " << b[i]
            << " c(" << i << ") = " << c[i]
            << " d(" << i << ") = " << d[i]
            << " e(" << i << ") = " << e[i]
            << std::endl;
    }
}

```

Mixed: Adapting Several Non-Proto Terminal Types

This is an example of using `BOOST_PROTO_DEFINE_OPERATORS()` to Protify expressions using `std::vector<>` and `std::list<>`, non-Proto types. It is a port of the Mixed example from [PETE](#).


```

////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This is an example of using BOOST_PROTO_DEFINE_OPERATORS to Protofy
// expressions using std::vector<> and std::list, non-proto types. It is a port
// of the Mixed example from PETE.
// (http://www.codesourcery.com/pooma/download.html).

#include <list>
#include <cmath>
#include <vector>
#include <complex>
#include <iostream>
#include <stdexcept>
#include <boost/proto/core.hpp>
#include <boost/proto/debug.hpp>
#include <boost/proto/context.hpp>
#include <boost/proto/transform.hpp>
#include <boost/utility/enable_if.hpp>
#include <boost/typeof/std/list.hpp>
#include <boost/typeof/std/vector.hpp>
#include <boost/typeof/std/complex.hpp>
#include <boost/type_traits/remove_reference.hpp>
namespace proto = boost::proto;
namespace mpl = boost::mpl;
using proto::_;

template<typename Expr>
struct MixedExpr;

template<typename Iter>
struct iterator_wrapper
{
    typedef Iter iterator;

    explicit iterator_wrapper(Iter iter)
        : it(iter)
    {}

    Iter it;
};

struct begin : proto::callable
{
    template<class Sig>
    struct result;

    template<class This, class Cont>
    struct result<This(Cont)>
        : proto::result_of::as_expr<
            iterator_wrapper<typename boost::remove_reference<Cont>::type::const_iterator>
        >
    {}

    template<typename Cont>
    typename result<begin(Cont const &)>::type
    operator()(Cont const &cont) const
    {
        iterator_wrapper<typename Cont::const_iterator> it(cont.begin());
        return proto::as_expr(it);
    }
}

```



```

};

// Here is a grammar that replaces vector and list terminals with their
// begin iterators
struct Begin
: proto::or_<
    proto::when< proto::terminal< std::vector<_, _> >, begin(proto::_value) >
    , proto::when< proto::terminal< std::list<_, _> >, begin(proto::_value) >
    , proto::when< proto::terminal<_> >
    , proto::when< proto::nary_expr<_, proto::vararg<Begin> > >
>
{};

// Here is an evaluation context that dereferences iterator
// terminals.
struct DereferenceCtx
{
    // Unless this is an iterator terminal, use the
    // default evaluation context
    template<typename Expr, typename EnableIf = void>
    struct eval
    : proto::default_eval<Expr, DereferenceCtx const>
    {};

    // Dereference iterator terminals.
    template<typename Expr>
    struct eval<
        Expr
        , typename boost::enable_if<
            proto::matches<Expr, proto::terminal<iterator_wrapper<_> > >
        >::type
    >
    {
        typedef typename proto::result_of::value<Expr>::type IteratorWrapper;
        typedef typename IteratorWrapper::iterator iterator;
        typedef typename std::iterator_traits<iterator>::reference result_type;

        result_type operator()(Expr &expr, DereferenceCtx const &) const
        {
            return *proto::value(expr).it;
        }
    };
};

// Here is an evaluation context that increments iterator
// terminals.
struct IncrementCtx
{
    // Unless this is an iterator terminal, use the
    // default evaluation context
    template<typename Expr, typename EnableIf = void>
    struct eval
    : proto::null_eval<Expr, IncrementCtx const>
    {};

    // advance iterator terminals.
    template<typename Expr>
    struct eval<
        Expr
        , typename boost::enable_if<
            proto::matches<Expr, proto::terminal<iterator_wrapper<_> > >
        >::type
    >

```



```

{
    typedef void result_type;

    result_type operator ()(Expr &expr, IncrementCtx const &) const
    {
        ++proto::value(expr).it;
    }
};

// A grammar which matches all the assignment operators,
// so we can easily disable them.
struct AssignOps
    : proto::switch_<struct AssignOpsCases>
{
};

// Here are the cases used by the switch_ above.
struct AssignOpsCases
{
    template<typename Tag, int D = 0> struct case_ : proto::not_<_> {};

    template<int D> struct case_< proto::tag::plus_assign, D > : _ {};
    template<int D> struct case_< proto::tag::minus_assign, D > : _ {};
    template<int D> struct case_< proto::tag::multiplies_assign, D > : _ {};
    template<int D> struct case_< proto::tag::divides_assign, D > : _ {};
    template<int D> struct case_< proto::tag::modulus_assign, D > : _ {};
    template<int D> struct case_< proto::tag::shift_left_assign, D > : _ {};
    template<int D> struct case_< proto::tag::shift_right_assign, D > : _ {};
    template<int D> struct case_< proto::tag::bitwise_and_assign, D > : _ {};
    template<int D> struct case_< proto::tag::bitwise_or_assign, D > : _ {};
    template<int D> struct case_< proto::tag::bitwise_xor_assign, D > : _ {};
};

// An expression conforms to the MixedGrammar if it is a terminal or some
// op that is not an assignment op. (Assignment will be handled specially.)
struct MixedGrammar
    : proto::or_<
        proto::terminal<_>
        , proto::and_<
            proto::nary_expr<_, proto::vararg<MixedGrammar> >
            , proto::not_<AssignOps>
        >
    >
{
};

// Expressions in the MixedDomain will be wrapped in MixedExpr<>
// and must conform to the MixedGrammar
struct MixedDomain
    : proto::domain<proto::generator<MixedExpr>, MixedGrammar>
{
};

// Here is MixedExpr, a wrapper for expression types in the MixedDomain.
template<typename Expr>
struct MixedExpr
    : proto::extends<Expr, MixedExpr<Expr>, MixedDomain>
{
    explicit MixedExpr(Expr const &expr)
        : proto::extends<Expr, MixedExpr<Expr>, MixedDomain>(expr)
    {}
private:
    // hide this:
    using proto::extends<Expr, MixedExpr<Expr>, MixedDomain>::operator [];
};

```



```

// Define a trait type for detecting vector and list terminals, to
// be used by the BOOST_PROTO_DEFINE_OPERATORS macro below.
template<typename T>
struct IsMixed
    : mpl::false_
{
};

template<typename T, typename A>
struct IsMixed<std::list<T, A> >
    : mpl::true_
{
};

template<typename T, typename A>
struct IsMixed<std::vector<T, A> >
    : mpl::true_
{
};

namespace MixedOps
{
    // This defines all the overloads to make expressions involving
    // std::vector to build expression templates.
    BOOST_PROTO_DEFINE_OPERATORS(IsMixed, MixedDomain)

    struct assign_op
    {
        template<typename T, typename U>
        void operator()(T &t, U const &u) const
        {
            t = u;
        }
    };

    struct plus_assign_op
    {
        template<typename T, typename U>
        void operator()(T &t, U const &u) const
        {
            t += u;
        }
    };

    struct minus_assign_op
    {
        template<typename T, typename U>
        void operator()(T &t, U const &u) const
        {
            t -= u;
        }
    };

    struct sin_
    {
        template<typename Sig>
        struct result;

        template<typename This, typename Arg>
        struct result<This(Arg)>
            : boost::remove_const<typename boost::remove_reference<Arg>::type>
        {
        };

        template<typename Arg>
        Arg operator()(Arg const &a) const
    };
}

```



```

    {
        return std::sin(a);
    }
};

template<typename A>
typename proto::result_of::make_expr<
    proto::tag::function
    , MixedDomain
    , sin_ const
    , A const &
>::type sin(A const &a)
{
    return proto::make_expr<proto::tag::function, MixedDomain>(sin_(), boost::ref(a));
}

template<typename FwdIter, typename Expr, typename Op>
void evaluate(FwdIter begin, FwdIter end, Expr const &expr, Op op)
{
    IncrementCtx const inc = {};
    DereferenceCtx const deref = {};
    typename boost::result_of<Begin(Expr const &)>::type expr2 = Begin()(expr);
    for(; begin != end; ++begin)
    {
        op(*begin, proto::eval(expr2, deref));
        proto::eval(expr2, inc);
    }
}

// Add-assign to a vector from some expression.
template<typename T, typename A, typename Expr>
std::vector<T, A> &assign(std::vector<T, A> &arr, Expr const &expr)
{
    evaluate(arr.begin(), arr.end(), proto::as_expr<MixedDomain>(expr), assign_op());
    return arr;
}

// Add-assign to a list from some expression.
template<typename T, typename A, typename Expr>
std::list<T, A> &assign(std::list<T, A> &arr, Expr const &expr)
{
    evaluate(arr.begin(), arr.end(), proto::as_expr<MixedDomain>(expr), assign_op());
    return arr;
}

// Add-assign to a vector from some expression.
template<typename T, typename A, typename Expr>
std::vector<T, A> &operator +=(std::vector<T, A> &arr, Expr const &expr)
{
    evaluate(arr.begin(), arr.end(), proto::as_expr<MixedDomain>(expr), plus_assign_op());
    return arr;
}

// Add-assign to a list from some expression.
template<typename T, typename A, typename Expr>
std::list<T, A> &operator +=(std::list<T, A> &arr, Expr const &expr)
{
    evaluate(arr.begin(), arr.end(), proto::as_expr<MixedDomain>(expr), plus_assign_op());
    return arr;
}

// Minus-assign to a vector from some expression.
template<typename T, typename A, typename Expr>

```



```

std::vector<T, A> &operator -= (std::vector<T, A> &arr, Expr const &expr)
{
    evaluate(arr.begin(), arr.end(), proto::as_expr<MixedDomain>(expr), minus_assign_op());
    return arr;
}

// Minus-assign to a list from some expression.
template<typename T, typename A, typename Expr>
std::list<T, A> &operator -= (std::list<T, A> &arr, Expr const &expr)
{
    evaluate(arr.begin(), arr.end(), proto::as_expr<MixedDomain>(expr), minus_assign_op());
    return arr;
}
}

int main()
{
    using namespace MixedOps;

    int n = 10;
    std::vector<int> a,b,c,d;
    std::list<double> e;
    std::list<std::complex<double> > f;

    int i;
    for(i = 0; i < n; ++i)
    {
        a.push_back(i);
        b.push_back(2*i);
        c.push_back(3*i);
        d.push_back(i);
        e.push_back(0.0);
        f.push_back(std::complex<double>(1.0, 1.0));
    }

    MixedOps::assign(b, 2);
    MixedOps::assign(d, a + b * c);
    a += if_else(d < 30, b, c);

    MixedOps::assign(e, c);
    e += e - 4 / (c + 1);

    f -= sin(0.1 * e * std::complex<double>(0.2, 1.2));

    std::list<double>::const_iterator ei = e.begin();
    std::list<std::complex<double> >::const_iterator fi = f.begin();
    for (i = 0; i < n; ++i)
    {
        std::cout
            << "a(" << i << ") = " << a[i]
            << " b(" << i << ") = " << b[i]
            << " c(" << i << ") = " << c[i]

```



```

    << " d(" << i << " ) = " << d[i]
    << " e(" << i << " ) = " << *ei++
    << " f(" << i << " ) = " << *fi++
    << std::endl;
}
}

```

Map Assign: An Intermediate Transform

A demonstration of how to implement `map_list_of()` from the Boost.Assign library using Proto. `map_list_assign()` is used to conveniently initialize a `std::map<>`. By using Proto, we can avoid any dynamic allocation while building the intermediate representation.

```

// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This is a port of map_list_of() from the Boost.Assign library.
// It has the advantage of being more efficient at runtime by not
// building any temporary container that requires dynamic allocation.

#include <map>
#include <string>
#include <iostream>
#include <boost/proto/core.hpp>
#include <boost/proto/transform.hpp>
#include <boost/type_traits/add_reference.hpp>
namespace proto = boost::proto;
using proto::_;

struct map_list_of_tag
{
};

// A simple callable function object that inserts a
// (key,value) pair into a map.
struct insert
: proto::callable
{
    template<typename Sig>
    struct result;

    template<typename This, typename Map, typename Key, typename Value>
    struct result<This(Map, Key, Value)>
    : boost::add_reference<Map>
    {
};

    template<typename Map, typename Key, typename Value>
    Map &operator()(Map &map, Key const &key, Value const &value) const
    {
        map.insert(typename Map::value_type(key, value));
        return map;
    }
};

// Work-arounds for Microsoft Visual C++ 7.1
#if BOOST_WORKAROUND(BOOST_MSVC, == 1310)
#define MapListof(x) proto::call<MapListof(x)>
#define _value(x) call<proto::_value(x)>
#endif

// The grammar for valid map-list expressions, and a

```



```

// transform that populates the map.
struct MapListOf
: proto::or_<
    proto::when<
        proto::function<
            proto::terminal<map_list_of_tag>
            , proto::terminal<_>
            , proto::terminal<_>
        >
        , insert(
            proto::_data
            , proto::_value(proto::_child1)
            , proto::_value(proto::_child2)
        )
    >
    , proto::when<
        proto::function<
            MapListOf
            , proto::terminal<_>
            , proto::terminal<_>
        >
        , insert(
            MapListOf(proto::_child0)
            , proto::_value(proto::_child1)
            , proto::_value(proto::_child2)
        )
    >
>
{};

#if BOOST_WORKAROUND(BOOST_MSVC, == 1310)
#undef MapListOf
#undef _value
#endif

template<typename Expr>
struct map_list_of_expr;

struct map_list_of_dom
: proto::domain<proto::pod_generator<map_list_of_expr>, MapListOf>
{};

// An expression wrapper that provides a conversion to a
// map that uses the MapListOf
template<typename Expr>
struct map_list_of_expr
{
    BOOST_PROTO_BASIC_EXTENDS(Expr, map_list_of_expr, map_list_of_dom)
    BOOST_PROTO_EXTENDS_FUNCTION()

    template<typename Key, typename Value, typename Cmp, typename Al>
    operator std::map<Key, Value, Cmp, Al> () const
    {
        BOOST_MPL_ASSERT((proto::matches<Expr, MapListOf>));
        std::map<Key, Value, Cmp, Al> map;
        return MapListOf>(*this, 0, map);
    }
};

map_list_of_expr<proto::terminal<map_list_of_tag>::type> const map_list_of = {{{{}}};

int main()
{

```



```
// Initialize a map:
std::map<std::string, int> op =
    map_list_of
        (<"", 1)
        (<"<=", 2)
        (<">", 3)
        (<">=", 4)
        (<"=", 5)
        (<"<>", 6)
    ;

std::cout << "\"<\" --> " << op["<"] << std::endl;
std::cout << "\"<=\" --> " << op["<="] << std::endl;
std::cout << "\">\" --> " << op[">"] << std::endl;
std::cout << "\">=\" --> " << op[">="] << std::endl;
std::cout << "\"=\" --> " << op["="] << std::endl;
std::cout << "\"<>\" --> " << op["<>"] << std::endl;

return 0;
}
```

Future Group: A More Advanced Transform

An advanced example of a Proto transform that implements Howard Hinnant's design for *future groups* that block for all or some asynchronous operations to complete and returns their results in a tuple of the appropriate type.


```

// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This is an example of using Proto transforms to implement
// Howard Hinnant's future group proposal.

#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/include/as_vector.hpp>
#include <boost/fusion/include/joint_view.hpp>
#include <boost/fusion/include/single_view.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/transform.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
namespace fusion = boost::fusion;
using proto::_;

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

// Work-arounds for Microsoft Visual C++ 7.1
#if BOOST_WORKAROUND(BOOST_MSVC, == 1310)
#define FutureGroup(x) proto::call<FutureGroup(x)>
#endif

// Define the grammar of future group expression, as well as a
// transform to turn them into a Fusion sequence of the correct
// type.
struct FutureGroup
: proto::or_<
    // terminals become a single-element Fusion sequence
    proto::when<
        proto::terminal<_>
        , fusion::single_view<proto::_value>(proto::_value)
    >
    // (a && b) becomes a concatenation of the sequence
    // from 'a' and the one from 'b':
    , proto::when<
        proto::logical_and<FutureGroup, FutureGroup>
        , fusion::joint_view<
            boost::add_const<FutureGroup(proto::_left)> >
            , boost::add_const<FutureGroup(proto::_right)> >
            >(FutureGroup(proto::_left), FutureGroup(proto::_right))
        >
    // (a || b) becomes the sequence for 'a', so long
    // as it is the same as the sequence for 'b'.
    , proto::when<
        proto::logical_or<FutureGroup, FutureGroup>
        , pick_left<
            FutureGroup(proto::_left)
            , FutureGroup(proto::_right)
            >(FutureGroup(proto::_left))
        >
    >
{};

#if BOOST_WORKAROUND(BOOST_MSVC, == 1310)
#undef FutureGroup

```



```

#endif

template<class E>
struct future_expr;

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

// Expressions in the future group domain have a .get()
// member function that (ostensibly) blocks for the futures
// to complete and returns the results in an appropriate
// tuple.
template<class E>
struct future_expr
: proto::extends<E, future_expr<E>, future_dom>
{
    explicit future_expr(E const &e)
    : proto::extends<E, future_expr<E>, future_dom>(e)
    {}

    typename fusion::result_of::as_vector<
        typename boost::result_of<FutureGroup(E,int,int)>::type
    >::type
    get() const
    {
        int i = 0;
        return fusion::as_vector(FutureGroup()(*this, i, i));
    }
};

// The future<> type has an even simpler .get()
// member function.
template<class T>
struct future
: future_expr<typename proto::terminal<T>::type>
{
    future(T const &t = T())
    : future_expr<typename proto::terminal<T>::type>(
        proto::terminal<T>::type::make(t)
    )
    {}

    T get() const
    {
        return proto::value(*this);
    }
};

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

int main()
{
    using fusion::vector;
    future<A> a;
    future<B> b;
    future<C> c;
    future<vector<A,B> > ab;

    // Verify that various future groups have the

```



```

// correct return types.
A          t0 = a.get();
vector<A, B, C> t1 = (a && b && c).get();
vector<A, C>  t2 = ((a || a) && c).get();
vector<A, B, C> t3 = ((a && b || a && b) && c).get();
vector<vector<A, B>, C> t4 = ((ab || ab) && c).get();

return 0;
}

```

Lambda: A Simple Lambda Library with Proto

This is an advanced example that shows how to implement a simple lambda DSEL with Proto, like the Boost.Lambda_library. It uses contexts, transforms and expression extension.

```

////////////////////////////////////
// Copyright 2008 Eric Niebler. Distributed under the Boost
// Software License, Version 1.0. (See accompanying file
// LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
// This example builds a simple but functional lambda library using Proto.

#include <iostream>
#include <algorithm>
#include <boost/mpl/int.hpp>
#include <boost/mpl/min_max.hpp>
#include <boost/mpl/eval_if.hpp>
#include <boost/mpl/identity.hpp>
#include <boost/mpl/next_prior.hpp>
#include <boost/fusion/tuple.hpp>
#include <boost/typeof/typeof.hpp>
#include <boost/typeof/std/ostream.hpp>
#include <boost/typeof/std/iostream.hpp>
#include <boost/proto/core.hpp>
#include <boost/proto/context.hpp>
#include <boost/proto/transform.hpp>
namespace mpl = boost::mpl;
namespace proto = boost::proto;
namespace fusion = boost::fusion;
using proto::_;

// Forward declaration of the lambda expression wrapper
template<typename T>
struct lambda;

struct lambda_domain
: proto::domain<proto::pod_generator<lambda> >
{};

template<typename I>
struct placeholder
{
    typedef I arity;
};

template<typename T>
struct placeholder_arity
{
    typedef typename T::arity type;
};

```



```

// The lambda grammar, with the transforms for calculating the max arity
struct lambda_arity
: proto::or_  

    proto::when<  

        proto::terminal< placeholder<_> >  

        , mpl::next<placeholder_arity<proto::_value> >()  

    >  

    , proto::when< proto::terminal<_>  

        , mpl::int_<0>()  

    >  

    , proto::when<  

        proto::nary_expr<_, proto::vararg<_> >  

        , proto::fold<_, mpl::int_<0>(), mpl::max<lambda_arity, proto::_state>()>  

    >  

>  

{};

// The lambda context is the same as the default context
// with the addition of special handling for lambda placeholders
template<typename Tuple>
struct lambda_context
: proto::callable_context<lambda_context<Tuple> const>
{
    lambda_context(Tuple const &args)
    : args_(args)
    {}

    template<typename Sig>
    struct result;

    template<typename This, typename I>
    struct result<This(proto::tag::terminal, placeholder<I> const &)>
    : fusion::result_of::at<Tuple, I>
    {};

    template<typename I>
    typename fusion::result_of::at<Tuple, I>::type
    operator()(proto::tag::terminal, placeholder<I> const &) const
    {
        return fusion::at<I>(this->args_);
    }

    Tuple args_;
};

// The lambda<> expression wrapper makes expressions polymorphic
// function objects
template<typename T>
struct lambda
{
    BOOST_PROTO_BASIC_EXTENDS(T, lambda<T>, lambda_domain)
    BOOST_PROTO_EXTENDS_ASSIGN()
    BOOST_PROTO_EXTENDS_SUBSCRIPT()

    // Calculate the arity of this lambda expression
    static int const arity = boost::result_of<lambda_arity(T)>::type::value;

    template<typename Sig>
    struct result;

    // Define nested result<> specializations to calculate the return
    // type of this lambda expression. But be careful not to evaluate
    // the return type of the nullary function unless we have a nullary

```



```

// lambda!
template<typename This>
struct result<This()>
    : mpl::eval_if_c<
        0 == arity
        , proto::result_of::eval<T const, lambda_context<fusion::tuple<> > >
        , mpl::identity<void>
    >
    {};

template<typename This, typename A0>
struct result<This(A0)>
    : proto::result_of::eval<T const, lambda_context<fusion::tuple<A0> > >
    {};

template<typename This, typename A0, typename A1>
struct result<This(A0, A1)>
    : proto::result_of::eval<T const, lambda_context<fusion::tuple<A0, A1> > >
    {};

// Define our operator () that evaluates the lambda expression.
typename result<lambda()>::type
operator ()() const
{
    fusion::tuple<> args;
    lambda_context<fusion::tuple<> > ctx(args);
    return proto::eval(*this, ctx);
}

template<typename A0>
typename result<lambda(A0 const &)>::type
operator ()(A0 const &a0) const
{
    fusion::tuple<A0 const &> args(a0);
    lambda_context<fusion::tuple<A0 const &> > ctx(args);
    return proto::eval(*this, ctx);
}

template<typename A0, typename A1>
typename result<lambda(A0 const &, A1 const &)>::type
operator ()(A0 const &a0, A1 const &a1) const
{
    fusion::tuple<A0 const &, A1 const &> args(a0, a1);
    lambda_context<fusion::tuple<A0 const &, A1 const &> > ctx(args);
    return proto::eval(*this, ctx);
}
};

// Define some lambda placeholders
lambda<proto::terminal<placeholder<mpl::int_<0> > >::type> const _1 = {{{}};
lambda<proto::terminal<placeholder<mpl::int_<1> > >::type> const _2 = {{{}};

template<typename T>
lambda<typename proto::terminal<T>::type> const val(T const &t)
{
    lambda<typename proto::terminal<T>::type> that = {{{t}}};
    return that;
}

template<typename T>
lambda<typename proto::terminal<T &>::type> const var(T &t)
{
    lambda<typename proto::terminal<T &>::type> that = {{{t}}};

```



```

    return that;
}

template<typename T>
struct construct_helper
{
    typedef T result_type; // for TR1 result_of

    T operator()() const
    { return T(); }

    // Generate BOOST_PROTO_MAX_ARITY overloads of the
    // following function call operator.
#define BOOST_PROTO_LOCAL_MACRO(N, typename_A, A_const_ref, A_const_ref_a, a)\
    template<typename_A(N)> \
    T operator()(A_const_ref_a(N)) const \
    { return T(a(N)); }
#define BOOST_PROTO_LOCAL_a BOOST_PROTO_a
#include BOOST_PROTO_LOCAL_ITERATE()
};

// Generate BOOST_PROTO_MAX_ARITY-1 overloads of the
// following construct() function template.
#define M0(N, typename_A, A_const_ref, A_const_ref_a, ref_a) \
template<typename T, typename_A(N)> \
typename proto::result_of::make_expr< \
    proto::tag::function \
    , lambda_domain \
    , construct_helper<T> \
    , A_const_ref(N) \
>::type const \
construct(A_const_ref_a(N)) \
{ \
    return proto::make_expr< \
        proto::tag::function \
        , lambda_domain \
    >( \
        construct_helper<T>() \
        , ref_a(N) \
    ); \
}
BOOST_PROTO_REPEAT_FROM_TO(1, BOOST_PROTO_MAX_ARITY, M0)
#undef M0

struct S
{
    S() {}
    S(int i, char c)
    {
        std::cout << "S(" << i << "," << c << ")\n";
    }
};

int main()
{
    // Create some lambda objects and immediately
    // invoke them by applying their operator():
    int i = ( (_1 + 2) / 4 )(42);
    std::cout << i << std::endl; // prints 11

    int j = ( (-(_1 + 2)) / 4 )(42);
    std::cout << j << std::endl; // prints -11
}

```



```
double d = ( (4 - _2) * 3 )(42, 3.14);
std::cout << d << std::endl; // prints 2.58

// check non-const ref terminals
(std::cout << _1 << " -- " << _2 << '\n')(42, "Life, the Universe and Everything!");
// prints "42 -- Life, the Universe and Everything!"

// "Nullary" lambdas work too
int k = (val(1) + val(2))();
std::cout << k << std::endl; // prints 3

// check array indexing for kicks
int integers[5] = {0};
(var(integers)[2] = 2)();
(var(integers)[_1] = _1)(3);
std::cout << integers[2] << std::endl; // prints 2
std::cout << integers[3] << std::endl; // prints 3

// Now use a lambda with an STL algorithm!
int rgi[4] = {1,2,3,4};
char rgc[4] = {'a','b','c','d'};
S rgs[4];

std::transform(rgi, rgi+4, rgc, rgs, construct<S>(_1, _2));
return 0;
}
```

Background and Resources

Proto was initially developed as part of [Boost.Xpressive](#) to simplify the job of transforming an expression template into an executable finite state machine capable of matching a regular expression. Since then, Proto has found application in the redesigned and improved Spirit-2 and the related Karma library. As a result of these efforts, Proto evolved into a generic and abstract grammar and tree transformation framework applicable in a wide variety of DSEL scenarios.

The grammar and tree transformation framework is modeled on Spirit's grammar and semantic action framework. The expression tree data structure is similar to Fusion data structures in many respects, and is interoperable with Fusion's iterators and algorithms.

The syntax for the grammar-matching features of `proto::matches<>` is inspired by MPL's lambda expressions.

The idea for using function types for Proto's composite transforms is inspired by Aleksey Gurtovoy's "round" [lambda](#) notation.

References

Ren, D. and Erwig, M. 2006. A generic recursion toolbox for Haskell or: scrap your boilerplate systematically. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell* (Portland, Oregon, USA, September 17 - 17, 2006). Haskell '06. ACM, New York, NY, 13-24. DOI=<http://doi.acm.org/10.1145/1159842.1159845>

Further Reading

A technical paper about an earlier version of Proto was accepted into the [ACM SIGPLAN Symposium on Library-Centric Software Design LCSD'07](#), and can be found at http://lcsd.cs.tamu.edu/2007/final/1/1_Paper.pdf. The tree transforms described in that paper differ from what exists today.

Glossary

callable transform

A transform of the form `R(A0,A1,...)` (i.e., a function type) where `proto::is_callable<R>::value` is true. `R` is treated as a polymorphic function object and the arguments are treated as transforms that yield the arguments to the function object.

context	In Proto, the term <i>context</i> refers to an object that can be passed, along with an expression to evaluate, to the <code>proto::eval()</code> function. The context determines how the expression is evaluated. All context structs define a nested <code>eval<></code> template that, when instantiated with a node tag type (e.g., <code>proto::tag::plus</code>), is a binary polymorphic function object that accepts an expression of that type and the context object. In this way, contexts associate behaviors with expression nodes.
domain	In Proto, the term <i>domain</i> refers to a type that associates expressions within that domain with a <i>generator</i> for that domain and optionally a <i>grammar</i> for the domain. Domains are used primarily to imbue expressions within that domain with additional members and to restrict Proto's operator overloads such that expressions not conforming to the domain's grammar are never created. Domains are empty structs that inherit from <code>proto::domain<></code> .
domain-specific embedded language	A domain-specific language implemented as a library. The language in which the library is written is called the "host" language, and the language implemented by the library is called the "embedded" language.
domain-specific language	A programming language that targets a particular problem space by providing programming idioms, abstractions and constructs that match the constructs within that problem space.
expression	In Proto, an <i>expression</i> is a heterogeneous tree where each node is either an instantiation of <code>boost::proto::expr<></code> , <code>boost::proto::basic_expr<></code> or some type that is an extension (via <code>boost::proto::extends<></code> or <code>BOOST_PROTO_EXTENDS()</code>) of such an instantiation.
expression template	A C++ technique using templates and operator overloading to cause expressions to build trees that represent the expression for lazy evaluation later, rather than evaluating the expression eagerly. Some C++ libraries use expression templates to build domain-specific embedded languages.
generator	In Proto, a <i>generator</i> is a unary polymorphic function object that you specify when defining a <i>domain</i> . After constructing a new expression, Proto passes the expression to your domain's generator for further processing. Often, the generator wraps the expression in an extension wrapper that adds additional members to it.
grammar	In Proto, a <i>grammar</i> is a type that describes a subset of Proto expression types. Expressions in a domain must conform to that domain's grammar. The <code>proto::matches<></code> metafunction evaluates whether an expression type matches a grammar. Grammars are either primitives such as <code>proto::_</code> , composites such as <code>proto::plus<></code> , control structures such as <code>proto::or_<></code> , or some type derived from a grammar.
object transform	A transform of the form <code>R(A0,A1,...)</code> (i.e., a function type) where <code>proto::is_callable<R>::value</code> is false. <code>R</code> is treated as the type of an object to construct and the arguments are treated as transforms that yield the parameters to the constructor.
polymorphic function object	An instance of a class type with an overloaded function call operator and a nested <code>result_type</code> typedef or <code>result<></code> template for calculating the return type of the function call operator.
primitive transform	A type that defines a kind of polymorphic function object that takes three arguments: expression, state, and data. Primitive transforms can be used to compose callable transforms and object transforms.
transform	Transforms are used to manipulate expression trees. They come in three flavors: primitive transforms, callable transforms, or object transforms. A transform <i>T</i> can be made into a ternary polymorphic function object with <code>proto::when<></code> , as in <code>proto::when<proto::_ , T></code> . Such a function object accepts <i>expression</i> , <i>state</i> , and <i>data</i> parameters, and computes a result from them.

Reference

Concepts

- [CallableTransform](#)
- [Domain](#)
- [Expr](#)
- [ObjectTransform](#)
- [PolymorphicFunctionObject](#)
- [PrimitiveTransform](#)
- [Transform](#)

Classes

- [proto::_](#)
- [proto::_byref](#)
- [proto::_byval](#)
- [proto::_child_c](#)
- [proto::_data](#)
- [proto::_default](#)
- [proto::_expr](#)
- [proto::_state](#)
- [proto::_value](#)
- [proto::_void](#)
- [proto::address_of](#)
- [proto::and_](#)
- [proto::arity_of](#)
- [proto::assign](#)
- [proto::basic_expr](#)
- [proto::binary_expr](#)
- [proto::bitwise_and](#)
- [proto::bitwise_and_assign](#)
- [proto::bitwise_or](#)
- [proto::bitwise_or_assign](#)

- `proto::bitwise_xor`
- `proto::bitwise_xor_assign`
- `proto::by_value_generator`
- `proto::call`
- `proto::callable`
- `proto::comma`
- `proto::complement`
- `proto::compose_generators`
- `proto::context::callable_context`
- `proto::context::callable_eval`
- `proto::context::default_context`
- `proto::context::default_eval`
- `proto::context::null_context`
- `proto::context::null_eval`
- `proto::convertible_to`
- `proto::deduce_domain`
- `proto::default_domain`
- `proto::default_generator`
- `proto::dereference`
- `proto::divides`
- `proto::divides_assign`
- `proto::domain`
- `proto::domain::as_child`
- `proto::domain::as_expr`
- `proto::domain_of`
- `proto::equal_to`
- `proto::exact`
- `proto::expr`
- `proto::extends`
- `proto::fold`
- `proto::fold_tree`
- `proto::function`

- `proto::functional::as_child`
- `proto::functional::as_expr`
- `proto::functional::child`
- `proto::functional::child_c`
- `proto::functional::deep_copy`
- `proto::functional::display_expr`
- `proto::functional::eval`
- `proto::functional::flatten`
- `proto::functional::left`
- `proto::functional::make_expr`
- `proto::functional::pop_front`
- `proto::functional::reverse`
- `proto::functional::right`
- `proto::functional::unpack_expr`
- `proto::functional::value`
- `proto::generator`
- `proto::greater`
- `proto::greater_equal`
- `proto::if_`
- `proto::if_else_`
- `proto::is_aggregate`
- `proto::is_callable`
- `proto::is_domain`
- `proto::is_expr`
- `proto::is_extension`
- `proto::is_proto_expr`
- `proto::lazy`
- `proto::less`
- `proto::less_equal`
- `proto::list1<>, proto::list2<>, ...`
- `proto::literal`
- `proto::logical_and`

- `proto::logical_not`
- `proto::logical_or`
- `proto::make`
- `proto::matches`
- `proto::mem_ptr`
- `proto::minus`
- `proto::minus_assign`
- `proto::modulus`
- `proto::modulus_assign`
- `proto::multiplies`
- `proto::multiplies_assign`
- `proto::nary_expr`
- `proto::negate`
- `proto::noinvoke`
- `proto::not_`
- `proto::not_equal_to`
- `proto::nullary_expr`
- `proto::or_`
- `proto::otherwise`
- `proto::pass_through`
- `proto::plus`
- `proto::plus_assign`
- `proto::pod_generator`
- `proto::post_dec`
- `proto::post_inc`
- `proto::pre_dec`
- `proto::pre_inc`
- `proto::protect`
- `proto::result_of::as_child`
- `proto::result_of::as_expr`
- `proto::result_of::child`
- `proto::result_of::child_c`

- `proto::result_of::deep_copy`
- `proto::result_of::eval`
- `proto::result_of::flatten`
- `proto::result_of::left`
- `proto::result_of::make_expr`
- `proto::result_of::right`
- `proto::result_of::unpack_expr`
- `proto::result_of::value`
- `proto::reverse_fold`
- `proto::reverse_fold_tree`
- `proto::shift_left`
- `proto::shift_left_assign`
- `proto::shift_right`
- `proto::shift_right_assign`
- `proto::subscript`
- `proto::switch_`
- `proto::tag::address_of`
- `proto::tag::assign`
- `proto::tag::bitwise_and`
- `proto::tag::bitwise_and_assign`
- `proto::tag::bitwise_or`
- `proto::tag::bitwise_or_assign`
- `proto::tag::bitwise_xor`
- `proto::tag::bitwise_xor_assign`
- `proto::tag::comma`
- `proto::tag::complement`
- `proto::tag::dereference`
- `proto::tag::divides`
- `proto::tag::divides_assign`
- `proto::tag::equal_to`
- `proto::tag::function`
- `proto::tag::greater`

- `proto::tag::greater_equal`
- `proto::tag::if_else_`
- `proto::tag::less`
- `proto::tag::less_equal`
- `proto::tag::logical_and`
- `proto::tag::logical_not`
- `proto::tag::logical_or`
- `proto::tag::mem_ptr`
- `proto::tag::minus`
- `proto::tag::minus_assign`
- `proto::tag::modulus`
- `proto::tag::modulus_assign`
- `proto::tag::multiplies`
- `proto::tag::multiplies_assign`
- `proto::tag::negate`
- `proto::tag::not_equal_to`
- `proto::tag::plus`
- `proto::tag::plus_assign`
- `proto::tag::post_dec`
- `proto::tag::post_inc`
- `proto::tag::pre_dec`
- `proto::tag::pre_inc`
- `proto::tag::shift_left`
- `proto::tag::shift_left_assign`
- `proto::tag::shift_right`
- `proto::tag::shift_right_assign`
- `proto::tag::subscript`
- `proto::tag::terminal`
- `proto::tag::unary_plus`
- `proto::tag_of`
- `proto::term`
- `proto::terminal`

- `proto::transform`
- `proto::transform_impl`
- `proto::unary_expr`
- `proto::unary_plus`
- `proto::use_basic_expr`
- `proto::unexpr`
- `proto::vararg`
- `proto::wants_basic_expr`
- `proto::when`

Functions

- `proto::as_child()`
- `proto::as_expr()`
- `proto::assert_matches()`
- `proto::assert_matches_not()`
- `proto::child()`
- `proto::child_c()`
- `proto::deep_copy()`
- `proto::display_expr()`
- `proto::eval()`
- `proto::flatten()`
- `proto::if_else()`
- `proto::left()`
- `proto::lit()`
- `proto::make_expr()`
- `proto::right()`
- `proto::unpack_expr()`
- `proto::value()`

Header **<boost/proto/args.hpp>**

Contains definitions of the `proto::term<>`, `proto::list1<>`, `proto::list2<>`, etc. class templates.


```
namespace boost {  
  namespace proto {  
    template<typename T> struct term;  
    template<typename... Arg> struct listN;  
  }  
}
```


Struct template term

`boost::proto::term` — A type sequence, for use as the 2nd parameter to the `proto::expr<>` and `proto::basic_expr<>` class templates.

Synopsis

```
// In header: <boost/proto/args.hpp>

template<typename T>
struct term {
    // types
    typedef T child0;
    static const long arity; // = 0;
};
```

Description

A type sequence with one element, for use as the 2nd parameter to the `proto::expr<>` and `proto::basic_expr<>` class templates. The sequence element represents the value of a terminal.

Struct template listN

`boost::proto::listN` — `proto::list1<>`, `proto::list2<>`, etc., are type sequences for use as the 2nd parameter to the `proto::expr<>` or `proto::basic_expr<>` class templates.

Synopsis

```
// In header: <boost/proto/args.hpp>

template<typename... Arg>
struct listN {
    // types
    typedef ArgM childM; // For each M in [0,N)
    static const long arity; // = N;
};
```

Description

Type sequences, for use as the 2nd parameter to the `proto::expr<>` or `proto::basic_expr<>` class template. The types in the sequence correspond to the children of a node in an expression tree. There is no type literally named "listN"; rather, there is a set of types named `proto::list1<>`, `proto::list2<>`, etc.

Header <boost/proto/core.hpp>

Includes all of Proto, except the contexts, transforms, debug utilities and Boost.Typeof registrations.

Header <boost/proto/debug.hpp>

Utilities for debugging Proto expression trees

```
BOOST_PROTO_ASSERT_MATCHES(expr, Grammar)
BOOST_PROTO_ASSERT_MATCHES_NOT(expr, Grammar)
```

```
namespace boost {
    namespace proto {
        template<typename Expr> void display_expr(Expr const &, std::ostream &);
        template<typename Expr> void display_expr(Expr const &);
        template<typename Grammar, typename Expr>
            void assert_matches(Expr const &);
        template<typename Grammar, typename Expr>
            void assert_matches_not(Expr const &);
        namespace functional {
            struct display_expr;
        }
    }
}
```


Struct `display_expr`

`boost::proto::functional::display_expr` — Pretty-print a Proto expression tree.

Synopsis

```
// In header: <boost/proto/debug.hpp>

struct display_expr {
    // types
    typedef void result_type;

    // construct/copy/destruct
    display_expr(std::ostream & = std::cout, int = 0);

    // public member functions
    template<typename Expr> void operator()(Expr const &) const;
};
```

Description

A [PolymorphicFunctionObject](#) which accepts a Proto expression tree and pretty-prints it to an `ostream` for debugging purposes.

`display_expr` public construct/copy/destruct

- `display_expr(std::ostream & sout = std::cout, int depth = 0);`

Parameters:	<code>depth</code>	The starting indentation depth for this node. Children nodes will be displayed at a starting depth of <code>depth+4</code> .
	<code>sout</code>	The <code>ostream</code> to which the expression tree will be written.

`display_expr` public member functions

- `template<typename Expr> void operator()(Expr const & expr) const;`

Function `display_expr`

`boost::proto::display_expr` — Pretty-print a Proto expression tree.

Synopsis

```
// In header: <boost/proto/debug.hpp>

template<typename Expr>
    void display_expr(Expr const & expr, std::ostream & sout);
template<typename Expr> void display_expr(Expr const & expr);
```

Description

Parameters: `expr` The Proto expression tree to pretty-print
 `sout` The ostream to which the output should be written. If not specified, defaults to `std::cout`.
Notes: Equivalent to `proto::functional::display_expr(0, sout)(expr)`.

Function template `assert_matches`

`boost::proto::assert_matches` — Assert at compile time that a particular expression matches the specified grammar.

Synopsis

```
// In header: <boost/proto/debug.hpp>

template<typename Grammar, typename Expr>
void assert_matches(Expr const & expr);
```

Description

Use `proto::assert_matches()` to assert at compile-time that an expression matches a grammar.

Example:

```
typedef proto::plus< proto::terminal< int >, proto::terminal< int > > PlusInts;

proto::assert_matches<PlusInts>( proto::lit(1) + 42 );
```

See also:

- `proto::assert_matches_not()`
- `BOOST_PROTO_ASSERT_MATCHES()`
- `BOOST_PROTO_ASSERT_MATCHES_NOT()`

Notes: Equivalent to `BOOST_MPL_ASSERT((proto::matches<Expr, Grammar>))`.

Function template `assert_matches_not`

`boost::proto::assert_matches_not` — Assert at compile time that a particular expression does not match the specified grammar.

Synopsis

```
// In header: <boost/proto/debug.hpp>

template<typename Grammar, typename Expr>
void assert_matches_not(Expr const & expr);
```

Description

Use `proto::assert_matches_not()` to assert at compile-time that an expression does not match a grammar.

Example:

```
typedef proto::plus< proto::terminal< int >, proto::terminal< int > > PlusInts;

proto::assert_matches_not<PlusInts>( proto::lit("a string") + 42 );
```

See also:

- `proto::assert_matches()`
- `BOOST_PROTO_ASSERT_MATCHES()`
- `BOOST_PROTO_ASSERT_MATCHES_NOT()`

Notes: Equivalent to `BOOST_MPL_ASSERT_NOT((proto::matches<Expr, Grammar>))`.

Macro BOOST_PROTO_ASSERT_MATCHES

BOOST_PROTO_ASSERT_MATCHES — Assert at compile time that a particular expression matches the specified grammar.

Synopsis

```
// In header: <boost/proto/debug.hpp>

BOOST_PROTO_ASSERT_MATCHES(expr, Grammar)
```

Description

Use BOOST_PROTO_ASSERT_MATCHES() to assert at compile-time that an expression matches a grammar.

Example:

```
typedef proto::plus< proto::terminal< int >, proto::terminal< int > > PlusInts;

BOOST_PROTO_ASSERT_MATCHES( proto::lit(1) + 42, PlusInts );
```

See also:

- `proto::assert_matches()`
- `proto::assert_matches_not()`
- `BOOST_PROTO_ASSERT_MATCHES_NOT()`

Macro BOOST_PROTO_ASSERT_MATCHES_NOT

BOOST_PROTO_ASSERT_MATCHES_NOT — Assert at compile time that a particular expression does not match the specified grammar.

Synopsis

```
// In header: <boost/proto/debug.hpp>

BOOST_PROTO_ASSERT_MATCHES_NOT(expr, Grammar)
```

Description

Use BOOST_PROTO_ASSERT_MATCHES_NOT() to assert at compile-time that an expression does not match a grammar.

Example:

```
typedef proto::plus< proto::terminal< int >, proto::terminal< int > > PlusInts;

BOOST_PROTO_ASSERT_MATCHES_NOT( proto::lit("a string") + 42, PlusInts );
```

See also:

- `proto::assert_matches()`
- `proto::assert_matches_not()`
- `BOOST_PROTO_ASSERT_MATCHES()`

Header **<boost/proto/deep_copy.hpp>**

Replace all nodes stored by reference by nodes stored by value.

```
namespace boost {
  namespace proto {
    template<typename Expr>
      typename proto::result_of::deep_copy<Expr>::type deep_copy(Expr const &);
    namespace result_of {
      template<typename Expr> struct deep_copy;
    }
    namespace functional {
      struct deep_copy;
    }
  }
}
```


Struct template `deep_copy`

`boost::proto::result_of::deep_copy` — A metafunction for calculating the return type of `proto::deep_copy()`.

Synopsis

```
// In header: <boost/proto/deep_copy.hpp>

template<typename Expr>
struct deep_copy {
    // types
    typedef unspecified type;
};
```

Description

A metafunction for calculating the return type of `proto::deep_copy()`. The type parameter `Expr` should be the type of a Proto expression tree. It should not be a reference type, nor should it be cv-qualified.

Struct `deep_copy`

`boost::proto::functional::deep_copy` — A [PolymorphicFunctionObject](#) type for deep-copying Proto expression trees.

Synopsis

```
// In header: <boost/proto/deep_copy.hpp>

struct deep_copy : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : result_of::deep_copy<Expr> {
    };

    // public member functions
    template<typename Expr>
    result_of::deep_copy<Expr>::type operator()(Expr const &) const;
};
```

Description

A [PolymorphicFunctionObject](#) type for deep-copying Proto expression trees. When a tree is deep-copied, all internal nodes and terminals held by reference are instead held by value. The only exception is function references, which continue to be held by reference.

`deep_copy` public member functions

1.

```
template<typename Expr>
    result_of::deep_copy<Expr>::type operator()(Expr const & expr) const;
```

Deep-copies a Proto expression tree, turning all nodes and terminals held by reference into ones held by value.

Struct template result<This(Expr)>

boost::proto::functional::deep_copy::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/deep_copy.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : result_of::deep_copy<Expr> {
};
```


Function template `deep_copy`

`boost::proto::deep_copy` — A function for deep-copying Proto expression trees.

Synopsis

```
// In header: <boost/proto/deep_copy.hpp>

template<typename Expr>
    typename proto::result_of::deep_copy<Expr>::type
    deep_copy(Expr const & expr);
```

Description

A function for deep-copying Proto expression trees. When a tree is deep-copied, all internal nodes and terminals held by reference are instead held by value.

Notes: Terminals of reference-to-function type are left unchanged.

 Equivalent to `proto::functional::deep_copy()(expr)` .

Header `<boost/proto/domain.hpp>`

Contains definition of the `proto::domain<>` class template and helpers for defining domains with a generator for customizing expression construction and a grammar for controlling operator overloading.

```
namespace boost {
    namespace proto {
        template<typename Generator = proto::default_generator,
                typename Grammar = proto::_, typename Super = unspecified>
            struct domain;
        struct default_domain;
        struct deduce_domain;
        template<typename T> struct is_domain;
        template<typename T> struct domain_of;
    }
}
```


Struct template domain

`boost::proto::domain` — For use in defining domain tags to be used with `proto::extends<>`, `BOOST_PROTO_EXTENDS()` and `BOOST_PROTO_DEFINE_OPERATORS()`. A *domain* associates an expression type with a *generator*, and optionally a *grammar*. It may also have a super-domain. Expressions in a sub-domain are interoperable (i.e. can be combined freely with) expressions in a super-domain. Finally, domains control how non-Proto objects are turned into Proto expressions and how they are combined to form larger Proto expressions.

Synopsis

```
// In header: <boost/proto/domain.hpp>

template<typename Generator = proto::default_generator,
        typename Grammar = proto::_ , typename Super = unspecified>
struct domain : Generator {
    // types
    typedef Grammar    proto_grammar;
    typedef Generator  proto_generator;
    typedef Super      proto_super_domain;

    // member classes/structs/unions

    // A callable unary MonomorphicFunctionObject that specifies how objects are
    // turned into Proto expressions in this domain. The resulting expression
    // object is suitable for storage in a local variable.
    template<typename T>
    struct as_expr : proto::callable {
        // types
        typedef see-below result_type;

        // public member functions
        result_type operator()(T &) const;
    };

    // A callable unary MonomorphicFunctionObject that specifies how objects are
    // turned into Proto expressions in this domain, for use in scenarios where
    // the resulting expression is intended to be made a child of another
    // expression.
    template<typename T>
    struct as_child : proto::callable {
        // types
        typedef see-below result_type;

        // public member functions
        result_type operator()(T &) const;
    };
};
```

Description

The Generator parameter determines how new expressions in the domain are post-processed. Typically, a generator wraps all new expressions in a wrapper that imparts domain-specific behaviors to expressions within its domain. (See `proto::extends<>`.)

The Grammar parameter determines whether a given expression is valid within the domain, and automatically disables any operator overloads which would cause an invalid expression to be created. By default, the Grammar parameter defaults to the wildcard, `proto::_`, which makes all expressions valid within the domain.

The Super parameter declares the domain currently being defined to be a sub-domain of Super. An expression in a sub-domain can be freely combined with expressions in its super-domain (and *its* super-domain, etc.).

Example:

```
template<typename Expr>
struct MyExpr;

struct MyGrammar
: proto::or_< proto::terminal<_>, proto::plus<MyGrammar, MyGrammar> >
{};

// Define MyDomain, in which all expressions are
// wrapped in MyExpr<> and only expressions that
// conform to MyGrammar are allowed.
struct MyDomain
: proto::domain<proto::generator<MyExpr>, MyGrammar>
{};

// Use MyDomain to define MyExpr
template<typename Expr>
struct MyExpr
: proto::extends<Expr, MyExpr<Expr>, MyDomain>
{
    // ...
};
```

The `domain::as_expr<>` and `domain::as_child<>` member templates define how non-Proto objects are turned into Proto terminals and how Proto expressions should be processed before they are combined to form larger expressions. They can be overridden in a derived domain for customization. See their descriptions to understand how Proto uses these two templates and what their default behavior is.

Struct template `as_expr`

`boost::proto::domain::as_expr` — A callable unary `MonomorphicFunctionObject` that specifies how objects are turned into Proto expressions in this domain. The resulting expression object is suitable for storage in a local variable.

Synopsis

```
// In header: <boost/proto/domain.hpp>

// A callable unary MonomorphicFunctionObject that specifies how objects are
// turned into Proto expressions in this domain. The resulting expression
// object is suitable for storage in a local variable.
template<typename T>
struct as_expr : proto::callable {
    // types
    typedef see-below result_type;

    // public member functions
    result_type operator()(T &) const;
};
```

Description

A unary `MonomorphicFunctionObject` that specifies how objects are turned into Proto expressions in this domain. The resulting expression object is suitable for storage in a local variable. In that scenario, it is usually preferable to return expressions by value; and, in the case of objects that are not yet Proto expressions, to wrap them by value (if possible) in a new Proto terminal expression. (Contrast this description with the description for `proto::domain::as_child`.)

The `as_expr` function object turns objects into Proto expressions, if they are not already, by making them Proto terminals held by value if possible. Objects that are already Proto expressions are simply returned by value. If `wants_basic_expr<Generator>::value` is true, then let E be `proto::basic_expr`; otherwise, let E be `proto::expr`. Given an lvalue t of type T :

- If T is not a Proto expression type, the resulting terminal is calculated as follows:
 - If T is a function type, an abstract type, or a type derived from `std::ios_base`, let A be $T \&$.
 - Otherwise, let A be the type T stripped of cv-qualifiers.
 Then, the result of `as_expr<T>()(t)` is `Generator()(E<tag::terminal, term< A > >::make(t))`.
- Otherwise, the result is t converted to an (un-const) rvalue.

`as_expr` public member functions

1. `result_type operator()(T & t) const;`

Parameters: t The object to wrap.

Struct template `as_child`

`boost::proto::domain::as_child` — A callable unary `MonomorphicFunctionObject` that specifies how objects are turned into Proto expressions in this domain, for use in scenarios where the resulting expression is intended to be made a child of another expression.

Synopsis

```
// In header: <boost/proto/domain.hpp>

// A callable unary MonomorphicFunctionObject that specifies how objects are
// turned into Proto expressions in this domain, for use in scenarios where
// the resulting expression is intended to be made a child of another
// expression.
template<typename T>
struct as_child : proto::callable {
    // types
    typedef see-below result_type;

    // public member functions
    result_type operator()(T &) const;
};
```

Description

A unary `MonomorphicFunctionObject` that specifies how objects are turned into Proto expressions in this domain. The resulting expression object is suitable for storage as a child of another expression. In that scenario, it is usually preferable to store child expressions by reference; or, in the case of objects that are not yet Proto expressions, to wrap them by reference in a new Proto terminal expression. (Contrast this description with the description for `proto::domain::as_expr`.)

The `as_child` function object turns objects into Proto expressions, if they are not already, by making them Proto terminals held by reference. Objects that are already Proto expressions are simply returned by reference. If `wants_basic_expr<Generator>::value` is true, then let E be `proto::basic_expr`; otherwise, let E be `proto::expr`. Given an lvalue t of type T :

- If T is not a Proto expression type, the resulting terminal is `Generator()(E<tag::terminal, term< T & > >::make(t))`.
- Otherwise, the result is the lvalue t .

`as_child` public member functions

1. `result_type operator()(T & t) const;`

Parameters: t The object to wrap.

Struct default_domain

boost::proto::default_domain — The domain expressions have by default, if `proto::extends<>` has not been used to associate a domain with an expression.

Synopsis

```
// In header: <boost/proto/domain.hpp>

struct default_domain : proto::domain<> {
};
```


Struct deduce_domain

`boost::proto::deduce_domain` — A pseudo-domain for use in functions and metafunctions that require a domain parameter. It indicates that the domain of the parent node should be inferred from the domains of the child nodes.

Synopsis

```
// In header: <boost/proto/domain.hpp>

struct deduce_domain {
};
```

Description

When `proto::deduce_domain` is used as a domain — either explicitly or implicitly by `proto::make_expr()`, `proto::unpack_expr()`, or Proto's operator overloads — Proto will use the domains of the child expressions to compute the domain of the parent. It is done in such a way that (A) expressions in domains that share a common super-domain are interoperable, and (B) expressions that are in the default domain (or a sub-domain thereof) are interoperable with *all* expressions. The rules are as follows:

- A sub-domain is *stronger* than its super-domain.
- `proto::default_domain` and all its sub-domains are *weaker* than all other domains.
- For each child, define a set of domains S_N that includes the child's domain and all its super-domains.
- Define a set I_S that is the intersection of all the individual sets S_N that don't contain `proto::default_domain`.
- Define a set I_W that is the intersection of all the individual sets S_N that contain `proto::default_domain`.
- Define a set P that is the union of I_S and I_W .
- The common domain is the strongest domain in set P , with the following caveat.
- Let U be the union of all sets S_N . If the result is `proto::default_domain` and U contains an element that is *not* `proto::default_domain`, it is an error.

Note: the above description sounds like it would be expensive to compute at compile time. In fact, it can all be done using C++ function overloading.

Struct template `is_domain`

`boost::proto::is_domain`

Synopsis

```
// In header: <boost/proto/domain.hpp>

template<typename T>
struct is_domain : mpl::bool_< true-or-false > {
};
```

Description

A metafunction that returns `mpl::true_` if the type `T` is the type of a Proto domain; `mpl::false_` otherwise. If `T` inherits from `proto::domain<>`, `is_domain<T>` is `mpl::true_`.

Struct template domain_of

boost::proto::domain_of

Synopsis

```
// In header: <boost/proto/domain.hpp>

template<typename T>
struct domain_of {
    // types
    typedef domain_of-T type;
};
```

Description

A metafunction that returns the domain of a given type. If `T` is a Proto expression type, it returns that expression's associated domain. If not, it returns `proto::default_domain`.

Header <boost/proto/eval.hpp>

Contains the `proto::eval()` expression evaluator.

```
namespace boost {
namespace proto {
    template<typename Expr, typename Context>
        typename proto::result_of::eval< Expr, Context >::type
        eval(Expr &, Context &);
    template<typename Expr, typename Context>
        typename proto::result_of::eval< Expr, Context >::type
        eval(Expr &, Context const &);
    namespace functional {
        struct eval;
    }
    namespace result_of {
        template<typename Expr, typename Context> struct eval;
    }
}
}
```


Struct eval

`boost::proto::functional::eval` — A [PolymorphicFunctionObject](#) type for evaluating a given Proto expression with a given context.

Synopsis

```
// In header: <boost/proto/eval.hpp>

struct eval : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr, typename Context>
    struct result<This(Expr, Context)> :
        proto::result_of::eval<
            typename boost::remove_reference< Expr >::type,
            typename boost::remove_reference< Context >::type
        >
    {
    };

    // public member functions
    template<typename Expr, typename Context>
    typename proto::result_of::eval< Expr, Context >::type
    operator()(Expr &, Context &) const;
    template<typename Expr, typename Context>
    typename proto::result_of::eval< Expr, Context >::type
    operator()(Expr &, Context const &) const;
};
```

Description

`eval` public member functions

1.

```
template<typename Expr, typename Context>
    typename proto::result_of::eval< Expr, Context >::type
    operator()(Expr & expr, Context & context) const;
```

Evaluate a given Proto expression with a given context.

Parameters: `context` The context in which the expression should be evaluated.
 `expr` The Proto expression to evaluate.

Returns: `typename Context::template eval<Expr>()(expr, context)`

2.

```
template<typename Expr, typename Context>
    typename proto::result_of::eval< Expr, Context >::type
    operator()(Expr & expr, Context const & context) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(Expr, Context)>

boost::proto::functional::eval::result<This(Expr, Context)>

Synopsis

```
// In header: <boost/proto/eval.hpp>

template<typename This, typename Expr, typename Context>
struct result<This(Expr, Context)> :
    proto::result_of::eval<
        typename boost::remove_reference< Expr >::type,
        typename boost::remove_reference< Context >::type
    >
{
};
```


Struct template eval

boost::proto::result_of::eval — A metafunction for calculating the return type of `proto::eval()` given a certain Expr and Context types.

Synopsis

```
// In header: <boost/proto/eval.hpp>

template<typename Expr, typename Context>
struct eval {
    // types
    typedef typename Context::template eval< Expr >::result_type type;
};
```


Function eval

boost::proto::eval — Evaluate a given Proto expression with a given context.

Synopsis

```
// In header: <boost/proto/eval.hpp>

template<typename Expr, typename Context>
    typename proto::result_of::eval< Expr, Context >::type
    eval(Expr & expr, Context & context);
template<typename Expr, typename Context>
    typename proto::result_of::eval< Expr, Context >::type
    eval(Expr & expr, Context const & context);
```

Description

Parameters: context The context in which the expression should be evaluated.
 expr The Proto expression to evaluate.

Returns: typename Context::template eval<Expr>()(expr, context)

Header <boost/proto/expr.hpp>

```
namespace boost {
    namespace proto {
        template<typename Tag, typename Args, long Arity = Args::arity>
            struct basic_expr;
        template<typename Tag, typename Args, long Arity = Args::arity> struct expr;
        template<typename Expr> struct unexpr;
    }
}
```


Struct template `basic_expr`

`boost::proto::basic_expr` — Simplified representation of a node in an expression tree.

Synopsis

```
// In header: <boost/proto/expr.hpp>

template<typename Tag, typename Args, long Arity = Args::arity>
struct basic_expr {
    // types
    typedef Tag                proto_tag;
    typedef Args               proto_args;
    typedef mpl::long_< Arity > proto_arity;
    typedef proto::default_domain proto_domain;
    typedef basic_expr         proto_grammar;
    typedef basic_expr         proto_base_expr;
    typedef basic_expr         proto_derived_expr;
    typedef typename Args::childN proto_childN;    // For each N in [0,max(Arity,1)).

    // public static functions
    template<typename... A> static basic_expr const make(A const &...);

    // public member functions
    basic_expr & proto_base();
    basic_expr const & proto_base() const;
};
```

Description

`proto::basic_expr<>` is a node in an expression template tree. It is a container for its child sub-trees. It also serves as the terminal nodes of the tree.

`Tag` is type that represents the operation encoded by this expression. It is typically one of the structs in the `boost::proto::tag` namespace, but it doesn't have to be. If `Arity` is 0 then this `expr<>` type represents a leaf in the expression tree.

`Args` is a list of types representing the children of this expression. It is an instantiation of one of `proto::list1<>`, `proto::list2<>`, etc. The child types must all themselves be either `proto::expr<>` or `proto::basic_expr<>&` (or extensions thereof via `proto::extends<>` or `BOOST_PROTO_EXTENDS()`), unless `Arity` is 0, in which case `Args` must be `proto::term<T>`, where `T` can be any type.

`proto::basic_expr<>` is a valid Fusion random-access sequence, where the elements of the sequence are the child expressions.

`basic_expr` public static functions

1.

```
template<typename... A> static basic_expr const make(A const &... a);
```

Requires: The number of supplied arguments must be `max(Arity,1)`.
Returns: A new `basic_expr` object initialized with the specified arguments.

`basic_expr` public member functions

1.

```
basic_expr & proto_base();
```

Returns: `*this`

2.

```
basic_expr const & proto_base() const;
```


This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template `expr`

`boost::proto::expr` — Representation of a node in an expression tree.

Synopsis

```
// In header: <boost/proto/expr.hpp>

template<typename Tag, typename Args, long Arity = Args::arity>
struct expr {
    // types
    typedef Tag                                proto_tag;
    typedef Args                              proto_args;
    typedef mpl::long_< Arity >              proto_arity;
    typedef proto::default_domain            proto_domain;
    typedef proto::basic_expr< Tag, Args, Arity > proto_grammar;
    typedef expr                              proto_base_expr;
    typedef expr                              proto_derived_expr;
    typedef typename Args::childN            proto_childN;    // For each N in [0,max(Arity,1)).

    // member classes/structs/unions
    template<typename Signature>
    struct result {
        // types
        typedef unspecified type;
    };

    // public static functions
    template<typename... A> static expr const make(A const &...);

    // public member functions
    expr & proto_base();
    expr const & proto_base() const;
    template<typename A> unspecified operator=(A &);
    template<typename A> unspecified operator=(A const &);
    template<typename A> unspecified operator=(A &) const;
    template<typename A> unspecified operator=(A const &) const;
    template<typename A> unspecified operator[](A &);
    template<typename A> unspecified operator[](A const &);
    template<typename A> unspecified operator[](A &) const;
    template<typename A> unspecified operator[](A const &) const;
    template<typename... A> unspecified operator()(A const &...);
    template<typename... A> unspecified operator()(A const &...) const;
    proto_childN childN;    // For each N in [0,max(Arity,1)).
    static const long proto_arity_c;    // = Arity;
};
```

Description

`proto::expr<>` is a node in an expression template tree. It is a container for its child sub-trees. It also serves as the terminal nodes of the tree.

`Tag` is type that represents the operation encoded by this expression. It is typically one of the structs in the `boost::proto::tag` namespace, but it doesn't have to be. If `Arity` is 0 then this `expr<>` type represents a leaf in the expression tree.

`Args` is a list of types representing the children of this expression. It is an instantiation of one of `proto::list1<>`, `proto::list2<>`, etc. The child types must all themselves be either `proto::expr<>` or `proto::basic_expr<>` (or extensions thereof via `proto::extends<>` or `BOOST_PROTO_EXTENDS()`), unless `Arity` is 0, in which case `Args` must be `proto::term<T>`, where `T` can be any type.

`proto::expr<>` is a valid Fusion random-access sequence, where the elements of the sequence are the child expressions.

expr public static functions

1.

```
template<typename... A> static expr const make(A const &... a);
```

Requires: The number of supplied arguments must be `max(Arity,1)`.

Returns: A new `expr` object initialized with the specified arguments.

expr public member functions

1.

```
expr & proto_base();
```

Returns: `*this`

2.

```
expr const & proto_base() const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3.

```
template<typename A> unspecified operator=(A & a);
```

Lazy assignment expression

Returns: A new expression node representing the assignment operation.

4.

```
template<typename A> unspecified operator=(A const & a);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

5.

```
template<typename A> unspecified operator=(A & a) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

6.

```
template<typename A> unspecified operator=(A const & a) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

7.

```
template<typename A> unspecified operator[] (A & a);
```

Lazy subscript expression

Returns: A new expression node representing the subscript operation.

8.

```
template<typename A> unspecified operator[] (A const & a);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

9.

```
template<typename A> unspecified operator[] (A & a) const;
```


This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

10.

```
template<typename A> unspecified operator[] (A const & a) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

11.

```
template<typename... A> unspecified operator() (A const &... a);
```

Lazy function call

Returns: A new expression node representing the function call operation.

12.

```
template<typename... A> unspecified operator() (A const &... a) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result

boost::proto::expr::result

Synopsis

```
// In header: <boost/proto/expr.hpp>

template<typename Signature>
struct result {
    // types
    typedef unspecified type;
};
```

Description

Encodes the return type of `proto::expr<>::operator()`. Makes `proto::expr<>` a TR1-style function object type usable with `boost::result_of<>`

Struct template unexpr

`boost::proto::unexpr` — Lets you inherit the interface of an expression while hiding from Proto the fact that the type is a Proto expression.

Synopsis

```
// In header: <boost/proto/expr.hpp>

template<typename Expr>
struct unexpr : Expr {
    // construct/copy/destruct
    unexpr(Expr const &);
};
```

Description

For an expression type `E`, `proto::is_expr<E>::value` is true, but `proto::is_expr<proto::unexpr<E> >::value` is false.

unexpr public construct/copy/destruct

1. `unexpr(Expr const & expr);`

Header <boost/proto/extends.hpp>

Macros and a base class for defining end-user expression types

```
BOOST_PROTO_EXTENDS(Expr, Derived, Domain)
BOOST_PROTO_BASIC_EXTENDS(Expr, Derived, Domain)
BOOST_PROTO_EXTENDS_ASSIGN()
BOOST_PROTO_EXTENDS_FUNCTION()
BOOST_PROTO_EXTENDS_SUBSCRIPT()
BOOST_PROTO_EXTENDS_USING_ASSIGN(Derived)
BOOST_PROTO_EXTENDS_USING_ASSIGN_NON_DEPENDENT(Derived)
```

```
namespace boost {
    namespace proto {
        struct is_proto_expr;
        template<typename Expr, typename Derived,
                typename Domain = proto::default_domain>
            struct extends;
    }
}
```


Struct `is_proto_expr`

`boost::proto::is_proto_expr` — Empty type to be used as a dummy template parameter of POD expression wrappers. It allows argument-dependent lookup to find Proto's operator overloads.

Synopsis

```
// In header: <boost/proto/extends.hpp>

struct is_proto_expr {
};
```

Description

`proto::is_proto_expr` allows argument-dependent lookup to find Proto's operator overloads. For example:

```
template<typename T, typename Dummy = proto::is_proto_expr>
struct my_terminal
{
    BOOST_PROTO_BASIC_EXTENDS(
        typename proto::terminal<T>::type
        , my_terminal<T>
        , proto::default_domain
    )
};

// ...
my_terminal<int> _1, _2;
_1 + _2;
```

Without the second `Dummy` template parameter, Proto's operator overloads would not be considered by name lookup.

Struct template extends

boost::proto::extends — For adding behaviors to a Proto expression template.

Synopsis

```
// In header: <boost/proto/extends.hpp>

template<typename Expr, typename Derived,
        typename Domain = proto::default_domain>
struct extends {
    // types
    typedef typename Expr::proto_base_expr      proto_base_expr;
    typedef Domain                               proto_domain;
    typedef Derived                             proto_derived_expr;
    typedef typename Expr::proto_tag            proto_tag;
    typedef typename Expr::proto_args           proto_args;
    typedef typename Expr::proto_arity          proto_arity;
    typedef typename Expr::proto_grammar        proto_grammar;
    typedef typename Expr::proto_childN         proto_childN;      // For ↵

    each N in [0,max(1,proto_arity_c))

    // member classes/structs/unions
    template<typename Signature>
    struct result {
        // types
        typedef unspecified type;
    };

    // construct/copy/destruct
    extends();
    extends(extends const &);
    extends(Expr const &);

    // public static functions
    static Derived const make(Expr const &);

    // public member functions
    proto_base_expr & proto_base();
    proto_base_expr const & proto_base() const;
    template<typename A> unspecified operator=(A &);
    template<typename A> unspecified operator=(A const &);
    template<typename A> unspecified operator=(A &) const;
    template<typename A> unspecified operator=(A const &) const;
    template<typename A> unspecified operator[](A &);
    template<typename A> unspecified operator[](A const &);
    template<typename A> unspecified operator[](A &) const;
    template<typename A> unspecified operator[](A const &) const;
    template<typename... A> unspecified operator()(A const &...);
    template<typename... A> unspecified operator()(A const &...) const;
    Expr proto_expr;      // For exposition only.
    static const long proto_arity_c; // = proto_base_expr::proto_arity_c;
};
```

Description

Use `proto::extends<>` to give expressions in your domain custom data members and member functions.

Conceptually, using `proto::extends<>` is akin to inheriting from `proto::expr<>` and adding your own members. Using `proto::extends<>` is generally preferable to straight inheritance because the members that would be inherited from `proto::expr<>` would be wrong; they would incorrectly slice off your additional members when building larger expressions from

smaller ones. `proto::extends<>` automatically gives your expression types the appropriate operator overloads that preserve your domain-specific members when composing expression trees.

Expression extensions are typically defined as follows:

```
template< typename Expr >
struct my_expr
: proto::extends<
    Expr                // The expression type we're extending
    , my_expr< Expr >    // The type we're defining
    , my_domain          // The domain associated with this expression extension
>
{
    typedef proto::extends< Expr, my_expr< Expr >, my_domain > base_type;

    // An expression extension is constructed from the expression
    // it is extending.
    my_expr( Expr const & e = Expr() )
        : base_type( e )
    {}

    // Unhide proto::extends::operator=
    // (This is only necessary if a lazy assignment operator
    // makes sense for your domain-specific language.)
    BOOST_PROTO_EXTENDS_USING_ASSIGN(my_expr)

    /*
    ... domain-specific members go here ...
    */
};
```

See also:

- [BOOST_PROTO_EXTENDS\(\)](#)
- [BOOST_PROTO_EXTENDS_USING_ASSIGN\(\)](#)
- [BOOST_PROTO_EXTENDS_USING_ASSIGN_NON_DEPENDENT\(\)](#)

extends public construct/copy/destruct

1. `extends();`
2. `extends(extends const & that);`
3. `extends(Expr const & expr_);`

extends public static functions

1. `static Derived const make(Expr const & expr);`

Construct an expression extension from the base expression.

extends public member functions

1. `proto_base_expr & proto_base();`

Returns: `proto_expr_.proto_base()`

Throws: Will not throw.

2. `proto_base_expr const & proto_base() const;`

Returns: `proto_expr_.proto_base()`

Throws: Will not throw.

3. `template<typename A> unspecified operator=(A & a);`

Lazy assignment expression

Returns: A new expression node representing the assignment operation.

4. `template<typename A> unspecified operator=(A const & a);`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

5. `template<typename A> unspecified operator=(A & a) const;`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

6. `template<typename A> unspecified operator=(A const & a) const;`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

7. `template<typename A> unspecified operator[](A & a);`

Lazy subscript expression

Returns: A new expression node representing the subscript operation.

8. `template<typename A> unspecified operator[](A const & a);`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

9. `template<typename A> unspecified operator[](A & a) const;`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

10. `template<typename A> unspecified operator[](A const & a) const;`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

11. `template<typename... A> unspecified operator()(A const &... a);`

Lazy function call

Returns: A new expression node representing the function call operation.

12. `template<typename... A> unspecified operator()(A const &... a) const;`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result

boost::proto::extends::result

Synopsis

```
// In header: <boost/proto/extends.hpp>

template<typename Signature>
struct result {
    // types
    typedef unspecified type;
};
```

Description

So that `boost::result_of<>` can compute the return type of `proto::extends::operator()`.

Macro BOOST_PROTO_EXTENDS

BOOST_PROTO_EXTENDS — For creating expression wrappers that add behaviors to a Proto expression template, like `proto::extends<>`, but while retaining POD-ness of the expression wrapper.

Synopsis

```
// In header: <boost/proto/extends.hpp>

BOOST_PROTO_EXTENDS(Expr, Derived, Domain)
```

Description

Equivalent to:

```
BOOST_PROTO_BASIC_EXTENDS(Expr, Derived, Domain)
BOOST_PROTO_EXTENDS_ASSIGN()
BOOST_PROTO_EXTENDS_SUBSCRIPT()
BOOST_PROTO_EXTENDS_FUNCTION()
```

Example:

```
template< class Expr >
struct my_expr;

struct my_domain
: proto::domain< proto::pod_generator< my_expr > >
{
};

template< class Expr >
struct my_expr
{
    // OK, this makes my_expr<> a valid Proto expression extension.
    // my_expr<> has overloaded assignment, subscript,
    // and function call operators that build expression templates.
    BOOST_PROTO_EXTENDS(Expr, my_expr, my_domain)
};

// OK, my_expr<> is POD, so this is statically initialized:
my_expr< proto::terminal<int>::type > const _1 = {{1}};
```


Macro BOOST_PROTO_BASIC_EXTENDS

BOOST_PROTO_BASIC_EXTENDS — For creating expression wrappers that add members to a Proto expression template, like `proto::extends<>`, but while retaining POD-ness of the expression wrapper.

Synopsis

```
// In header: <boost/proto/extends.hpp>

BOOST_PROTO_BASIC_EXTENDS(Expr, Derived, Domain)
```

Description

BOOST_PROTO_BASIC_EXTENDS() adds the basic typedefs, member functions, and data members necessary to make a struct a valid Proto expression extension. It does *not* add any constructors, virtual functions or access control blocks that would render the containing struct non-POD.

Expr is the Proto expression that the enclosing struct extends. Derived is the type of the enclosing struct. Domain is the Proto domain to which this expression extension belongs. (See `proto::domain<>`.)

BOOST_PROTO_BASIC_EXTENDS() adds to its enclosing struct exactly one data member of type Expr.

Example:

```
template< class Expr >
struct my_expr;

struct my_domain
: proto::domain< proto::pod_generator< my_expr > >
{ };

template< class Expr >
struct my_expr
{
    // OK, this makes my_expr<> a valid Proto expression extension.
    // my_expr<> does /not/ have overloaded assignment, subscript,
    // and function call operators that build expression templates, however.
    BOOST_PROTO_BASIC_EXTENDS(Expr, my_expr, my_domain)
};

// OK, my_expr<> is POD, so this is statically initialized:
my_expr< proto::terminal<int>::type > const _1 = {{1}};
```

See also:

- [BOOST_PROTO_EXTENDS_ASSIGN\(\)](#)
- [BOOST_PROTO_EXTENDS_SUBSCRIPT\(\)](#)
- [BOOST_PROTO_EXTENDS_FUNCTION\(\)](#)
- [BOOST_PROTO_EXTENDS\(\)](#)

Macro `BOOST_PROTO_EXTENDS_ASSIGN`

`BOOST_PROTO_EXTENDS_ASSIGN` — For adding to an expression extension class an overloaded assignment operator that builds an expression template.

Synopsis

```
// In header: <boost/proto/extends.hpp>

BOOST_PROTO_EXTENDS_ASSIGN( )
```

Description

Use `BOOST_PROTO_EXTENDS_ASSIGN()` after `BOOST_PROTO_BASIC_EXTENDS()` to give an expression extension class an overloaded assignment operator that builds an expression template.

See also:

- `BOOST_PROTO_BASIC_EXTENDS()`
- `BOOST_PROTO_EXTENDS_SUBSCRIPT()`
- `BOOST_PROTO_EXTENDS_FUNCTION()`
- `BOOST_PROTO_EXTENDS()`

Macro `BOOST_PROTO_EXTENDS_FUNCTION`

`BOOST_PROTO_EXTENDS_FUNCTION` — For adding to an expression extension class a set of overloaded function call operators that build expression templates.

Synopsis

```
// In header: <boost/proto/extends.hpp>

BOOST_PROTO_EXTENDS_FUNCTION( )
```

Description

Use `BOOST_PROTO_EXTENDS_FUNCTION()` after `BOOST_PROTO_BASIC_EXTENDS()` to give an expression extension class a set of overloaded function call operators that build expression templates. In addition, `BOOST_PROTO_EXTENDS_FUNCTION()` adds a nested `result<>` class template that is a metafunction for calculating the return type of the overloaded function call operators.

See also:

- `BOOST_PROTO_BASIC_EXTENDS()`
- `BOOST_PROTO_EXTENDS_ASSIGN()`
- `BOOST_PROTO_EXTENDS_SUBSCRIPT()`
- `BOOST_PROTO_EXTENDS()`

Macro **BOOST_PROTO_EXTENDS_SUBSCRIPT**

BOOST_PROTO_EXTENDS_SUBSCRIPT — For adding to an expression extension class an overloaded subscript operator that builds an expression template.

Synopsis

```
// In header: <boost/proto/extends.hpp>

BOOST_PROTO_EXTENDS_SUBSCRIPT( )
```

Description

Use **BOOST_PROTO_EXTENDS_SUBSCRIPT()** after **BOOST_PROTO_BASIC_EXTENDS()** to give an expression extension class an overloaded subscript operator that builds an expression template.

See also:

- **BOOST_PROTO_BASIC_EXTENDS()**
- **BOOST_PROTO_EXTENDS_ASSIGN()**
- **BOOST_PROTO_EXTENDS_FUNCTION()**
- **BOOST_PROTO_EXTENDS()**

Macro `BOOST_PROTO_EXTENDS_USING_ASSIGN`

`BOOST_PROTO_EXTENDS_USING_ASSIGN` — For exposing in classes that inherit from `proto::extends<>` the overloaded assignment operators defined therein.

Synopsis

```
// In header: <boost/proto/extends.hpp>

BOOST_PROTO_EXTENDS_USING_ASSIGN(Derived)
```

Description

The standard usage of `proto::extends<>` is to inherit from it. However, the derived class automatically gets a compiler-generated assignment operator that will hide the ones defined in `proto::extends<>`. Use `BOOST_PROTO_EXTENDS_USING_ASSIGN()` in the derived class to unhide the assignment operators defined in `proto::extends<>`.

See `proto::extends<>` for an example that demonstrates usage of `BOOST_PROTO_EXTENDS_USING_ASSIGN()`.

Macro BOOST_PROTO_EXTENDS_USING_ASSIGN_NON_DEPENDENT

BOOST_PROTO_EXTENDS_USING_ASSIGN_NON_DEPENDENT — For exposing in classes that inherit from `proto::extends<>` the overloaded assignment operators defined therein. Unlike the `BOOST_PROTO_EXTENDS_USING_ASSIGN()` macro, `BOOST_PROTO_EXTENDS_USING_ASSIGN_NON_DEPENDENT()` is for use in non-dependent contexts.

Synopsis

```
// In header: <boost/proto/extends.hpp>

BOOST_PROTO_EXTENDS_USING_ASSIGN_NON_DEPENDENT(Derived)
```

Description

The standard usage of `proto::extends<>` is to define a class template that inherits from it. The derived class template automatically gets a compiler-generated assignment operator that hides the ones defined in `proto::extends<>`. Using `BOOST_PROTO_EXTENDS_USING_ASSIGN()` in the derived class solves this problem.

However, if the expression extension is an ordinary class and not a class template, the usage of `BOOST_PROTO_EXTENDS_USING_ASSIGN()` is in a so-called non-dependent context. In plain English, it means it is illegal to use `typename` in some places where it is required in a class template. In those cases, you should use `BOOST_PROTO_EXTENDS_USING_ASSIGN_NON_DEPENDENT()` instead.

See also:

- `proto::extends<>`
- `BOOST_PROTO_EXTENDS_USING_ASSIGN()`

Header <boost/proto/fusion.hpp>

Make any Proto expression a valid Fusion sequence

```
namespace boost {
  namespace proto {
    template<typename Expr>
      typename proto::result_of::flatten< Expr >::type const flatten(Expr &);
    template<typename Expr>
      typename proto::result_of::flatten< Expr const >::type const
        flatten(Expr const &);
    namespace functional {
      struct flatten;
      struct pop_front;
      struct reverse;
    }
    namespace result_of {
      template<typename Expr> struct flatten;
    }
  }
}
```


Struct flatten

`boost::proto::functional::flatten` — A [PolymorphicFunctionObject](#) type that returns a "flattened" view of a Proto expression tree.

Synopsis

```
// In header: <boost/proto/fusion.hpp>

struct flatten : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : result< This(Expr const &) > {
    };
    template<typename This, typename Expr>
    struct result<This(Expr &)> : proto::result_of::flatten< Expr > {
    };

    // public member functions
    template<typename Expr>
        typename proto::result_of::flatten< Expr >::type const
        operator()(Expr &) const;
    template<typename Expr>
        typename proto::result_of::flatten< Expr const >::type const
        operator()(Expr const &) const;
};
```

Description

A [PolymorphicFunctionObject](#) type that returns a "flattened" view of a Proto expression tree. For a tree with a top-most node tag of type `T`, the elements of the flattened sequence are determined by recursing into each child node with the same tag type and returning those nodes of different type. So for instance, the Proto expression tree corresponding to the expression `a | b | c` has a flattened view with elements `[a, b, c]`, even though the tree is grouped as `((a | b) | c)`.

`flatten` public member functions

1.

```
template<typename Expr>
    typename proto::result_of::flatten< Expr >::type const
    operator()(Expr & expr) const;
```
2.

```
template<typename Expr>
    typename proto::result_of::flatten< Expr const >::type const
    operator()(Expr const & expr) const;
```


Struct template result<This(Expr)>

boost::proto::functional::flatten::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : result< This(Expr const &) > {
};
```


Struct template result<This(Expr &)>

boost::proto::functional::flatten::result<This(Expr &)>

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename This, typename Expr>
struct result<This(Expr &)> : proto::result_of::flatten< Expr > {
};
```


Struct pop_front

`boost::proto::functional::pop_front` — A [PolymorphicFunctionObject](#) type that invokes the `fusion::pop_front()` algorithm on its argument.

Synopsis

```
// In header: <boost/proto/fusion.hpp>

struct pop_front : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : result< This(Expr const &) > {
    };
    template<typename This, typename Expr>
    struct result<This(Expr &)> : fusion::result_of::pop_front< Expr > {
    };

    // public member functions
    template<typename Expr>
    typename fusion::result_of::pop_front< Expr >::type
    operator()(Expr &) const;
    template<typename Expr>
    typename fusion::result_of::pop_front< Expr const >::type
    operator()(Expr const &) const;
};
```

Description

A [PolymorphicFunctionObject](#) type that invokes the `fusion::pop_front()` algorithm on its argument. This is useful for defining a [CallableTransform](#) such as `pop_front()`, which removes the first child from a Proto expression node. Such a transform might be used as the first argument to the `proto::fold<>` transform; that is, fold all but the first child.

pop_front public member functions

1.

```
template<typename Expr>
    typename fusion::result_of::pop_front< Expr >::type
    operator()(Expr & expr) const;
```

Returns: `fusion::pop_front(expr)`

2.

```
template<typename Expr>
    typename fusion::result_of::pop_front< Expr const >::type
    operator()(Expr const & expr) const;
```

Returns: `fusion::pop_front(expr)`

Struct template result<This(Expr)>

boost::proto::functional::pop_front::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : result< This(Expr const &) > {
};
```


Struct template result<This(Expr &)>

boost::proto::functional::pop_front::result<This(Expr &)>

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename This, typename Expr>
struct result<This(Expr &)> : fusion::result_of::pop_front< Expr > {
};
```


Struct reverse

`boost::proto::functional::reverse` — A [PolymorphicFunctionObject](#) type that invokes the `fusion::reverse()` algorithm on its argument.

Synopsis

```
// In header: <boost/proto/fusion.hpp>

struct reverse : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : result< This(Expr const &) > {
    };
    template<typename This, typename Expr>
    struct result<This(Expr &)> : fusion::result_of::reverse< Expr > {
    };

    // public member functions
    template<typename Expr>
    typename fusion::result_of::reverse< Expr >::type operator()(Expr &) const;
    template<typename Expr>
    typename fusion::result_of::reverse< Expr const >::type
    operator()(Expr const &) const;
};
```

Description

A [PolymorphicFunctionObject](#) type that invokes the `fusion::reverse()` algorithm on its argument. This is useful for defining a [CallableTransform](#) like `reverse(_)`, which reverses the order of the children of a Proto expression node.

`reverse` public member functions

1.

```
template<typename Expr>
    typename fusion::result_of::reverse< Expr >::type
    operator()(Expr & expr) const;
```

Returns: `fusion::reverse(expr)`

2.

```
template<typename Expr>
    typename fusion::result_of::reverse< Expr const >::type
    operator()(Expr const & expr) const;
```

Returns: `fusion::reverse(expr)`

Struct template result<This(Expr)>

boost::proto::functional::reverse::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : result< This(Expr const &) > {
};
```


Struct template result<This(Expr &)>

boost::proto::functional::reverse::result<This(Expr &)>

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename This, typename Expr>
struct result<This(Expr &)> : fusion::result_of::reverse< Expr > {
};
```


Struct template flatten

boost::proto::result_of::flatten — Metafunction that computes the return type of `proto::flatten()`

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename Expr>
struct flatten {
    // types
    typedef unspecified type;
};
```


Function flatten

`boost::proto::flatten` — A function that returns a "flattened" view of a Proto expression tree.

Synopsis

```
// In header: <boost/proto/fusion.hpp>

template<typename Expr>
    typename proto::result_of::flatten< Expr >::type const flatten(Expr & expr);
template<typename Expr>
    typename proto::result_of::flatten< Expr const >::type const
    flatten(Expr const & expr);
```

Description

For a tree with a top-most node tag of type `T`, the elements of the flattened sequence are determined by recursing into each child node with the same tag type and returning those nodes of different type. So for instance, the Proto expression tree corresponding to the expression `a | b | c` has a flattened view with elements `[a, b, c]`, even though the tree is grouped as `((a | b) | c)`.

Header **<boost/proto/generate.hpp>**

Contains definition of `proto::default_generator`, `proto::generator<>`, `proto::pod_generator<>` and other utilities that users can use to post-process new expression objects that Proto creates.

```
namespace boost {
    namespace proto {
        struct default_generator;
        template<template< typename > class Extends> struct generator;
        template<template< typename > class Extends> struct pod_generator;
        struct by_value_generator;
        template<typename First, typename Second> struct compose_generators;
        template<typename Generator> struct use_basic_expr;
        template<typename Generator> struct wants_basic_expr;
    }
}
```


Struct default_generator

boost::proto::default_generator — A simple generator that passes an expression through unchanged.

Synopsis

```
// In header: <boost/proto/generate.hpp>

struct default_generator : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> {
        // types
        typedef Expr type;
    };

    // public member functions
    template<typename Expr> Expr operator()(Expr const &) const;
};
```

Description

Generators are intended for use as the first template parameter to the `proto::domain<>` class template and control if and how expressions within that domain are to be customized. The `proto::default_generator` makes no modifications to the expressions passed to it.

default_generator public member functions

1.

```
template<typename Expr> Expr operator()(Expr const & expr) const;
```

Parameters: `expr` A Proto expression

Returns: `expr`

Struct template result<This(Expr)>

boost::proto::default_generator::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> {
    // types
    typedef Expr type;
};
```


Struct template generator

`boost::proto::generator` — A generator that wraps expressions passed to it in the specified extension wrapper.

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<template< typename > class Extends>
struct generator {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> {
        // types
        typedef Extends< Expr > type;
    };

    // public member functions
    template<typename Expr> Extends< Expr > operator()(Expr const &) const;
};
```

Description

Generators are intended for use as the first template parameter to the `proto::domain<>` class template and control if and how expressions within that domain are to be customized. `proto::generator<>` wraps each expression passed to it in the `Extends<>` wrapper.

`generator` public member functions

1.

```
template<typename Expr> Extends< Expr > operator()(Expr const & expr) const;
```

Parameters: `expr` A Proto expression

Returns: `Extends<Expr>(expr)`

Struct template result<This(Expr)>

boost::proto::generator::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> {
    // types
    typedef Extends< Expr > type;
};
```


Struct template pod_generator

`boost::proto::pod_generator` — A generator that wraps expressions passed to it in the specified extension wrapper and uses aggregate initialization for the wrapper.

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<template< typename > class Extends>
struct pod_generator : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> {
        // types
        typedef Extends< Expr > type;
    };

    // public member functions
    template<typename Expr> Extends< Expr > operator()(Expr const &) const;
};
```

Description

Generators are intended for use as the first template parameter to the `proto::domain<>` class template and control if and how expressions within that domain are to be customized. `proto::pod_generator<>` wraps each expression passed to it in the `Extends<>` wrapper, and uses aggregate initialization for the wrapped object.

pod_generator public member functions

1.

```
template<typename Expr> Extends< Expr > operator()(Expr const & expr) const;
```

Parameters: `expr` A Proto expression
Returns: `Extends<Expr> that = {expr};` return that;

Struct template result<This(Expr)>

boost::proto::pod_generator::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> {
    // types
    typedef Extends< Expr > type;
};
```


Struct `by_value_generator`

`boost::proto::by_value_generator` — A generator that replaces child nodes held by reference with ones held by value. Use with `proto::compose_generators<>` to forward that result to another generator.

Synopsis

```
// In header: <boost/proto/generate.hpp>

struct by_value_generator : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> {
        // types
        typedef unspecified type;
    };

    // public member functions
    template<typename Expr> unspecified operator()(Expr const &) const;
};
```

Description

Generators are intended for use as the first template parameter to the `proto::domain<>` class template and control if and how expressions within that domain are to be customized. `proto::by_value_generator` ensures all child nodes are held by value. This generator is typically composed with a second generator for further processing, as `proto::compose_generators<proto::by_value_generator, MyGenerator>`.

`by_value_generator` public member functions

1.

```
template<typename Expr> unspecified operator()(Expr const & expr) const;
```

Parameters: `expr` A Proto expression.
Returns: Equivalent to `proto::deep_copy(expr)`

Struct template result<This(Expr)>

boost::proto::by_value_generator::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> {
    // types
    typedef unspecified type;
};
```


Struct template `compose_generators`

`boost::proto::compose_generators` — A composite generator that first applies one transform to an expression and then forwards the result on to another generator for further transformation.

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename First, typename Second>
struct compose_generators : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> :
        boost::result_of<
            Second(typename boost::result_of<First(Expr)>::type)
        >
    {
    };

    // public member functions
    template<typename Expr>
    typename boost::result_of<
        Second(typename boost::result_of<First(Expr)>::type)
        >::type
    operator()(Expr const &) const;
};
```

Description

Generators are intended for use as the first template parameter to the `proto::domain<>` class template and control if and how expressions within that domain are to be customized. `proto::compose_generators<>` is a composite generator that first applies one transform to an expression and then forwards the result on to another generator for further transformation.

`compose_generators` public member functions

1.

```
template<typename Expr>
typename boost::result_of<
    Second(typename boost::result_of<First(Expr)>::type)
>::type
operator()(Expr const & expr) const;
```

Parameters: `expr` A Proto expression.
Returns: `Second()(First()(expr))`

Struct template result<This(Expr)>

boost::proto::compose_generators::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> :
    boost::result_of<
        Second(typename boost::result_of<First(Expr)>::type)
    >
{
};
```


Struct template use_basic_expr

boost::proto::use_basic_expr

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename Generator>
struct use_basic_expr : Generator {
};
```

Description

Annotate a generator to indicate that it would prefer to be passed instances of `proto::basic_expr<>` rather than `proto::expr<>`.

`use_basic_expr< Generator >` is itself a generator.

Struct template wants_basic_expr

boost::proto::wants_basic_expr

Synopsis

```
// In header: <boost/proto/generate.hpp>

template<typename Generator>
struct wants_basic_expr : mpl::bool_< true-or-false > {
};
```

Description

A Boolean metafunction that tests a generator to see whether it would prefer to be passed instances of `proto::basic_expr<>` rather than `proto::expr<>`.

Header <boost/proto/literal.hpp>

The `proto::literal<>` terminal wrapper, and the `proto::lit()` function for creating `proto::literal<>` wrappers.

```
namespace boost {
  namespace proto {
    template<typename T, typename Domain = proto::default_domain>
      struct literal;
    template<typename T> proto::literal< T & > const lit(T &);
    template<typename T> proto::literal< T const & > const lit(T const &);
  }
}
```


Struct template literal

`boost::proto::literal` — A simple wrapper for a terminal, provided for ease of use.

Synopsis

```
// In header: <boost/proto/literal.hpp>

template<typename T, typename Domain = proto::default_domain>
struct literal :
    proto::extends<proto::basic_expr<proto::tag::terminal, proto::term< T > >, proto::literal<T, Domain>, Domain>
{
    // types
    typedef proto::basic_expr<proto::tag::terminal, proto::term< T > > X; // For exposition only
    typedef typename proto::result_of::value<X>::type value_type;
    typedef typename proto::result_of::value<X &>::type reference;
    typedef typename proto::result_of::value<X const &>::type const_reference;

    // construct/copy/destruct
    literal();
    template<typename U> literal(U & u);
    template<typename U> literal(U const & u);
    template<typename U> literal(proto::literal< U, Domain > const & u);

    // public member functions
    reference get();
    const_reference get() const;
};
```

Description

A simple wrapper for a terminal, provided for ease of use. In all cases, `proto::literal<X> l(x);` is equivalent to `proto::terminal<X>::type l = {x};`.

The Domain template parameter defaults to `proto::default_domain`.

literal public construct/copy/destruct

1. `literal();`
2. `template<typename U> literal(U & u);`
3. `template<typename U> literal(U const & u);`
4. `template<typename U> literal(proto::literal< U, Domain > const & u);`

literal public member functions

1. `reference get();`

Returns: `proto::value(*this)`

2. `const_reference get() const;`

Returns: `proto::value(*this)`

Function lit

`boost::proto::lit` — A helper function for creating a `proto::literal<>` wrapper.

Synopsis

```
// In header: <boost/proto/literal.hpp>

template<typename T> proto::literal< T & > const lit(T & t);
template<typename T> proto::literal< T const & > const lit(T const & t);
```

Description

Parameters: `t` The object to wrap.
Returns: `proto::literal<T &>(t)`
Throws: Will not throw.
Notes: The returned value holds the argument by reference.

Header <boost/proto/make_expr.hpp>

Definition of the `proto::make_expr()` and `proto::unpack_expr()` utilities for building Proto expression nodes from child nodes or from a Fusion sequence of child nodes, respectively.

```
namespace boost {
namespace proto {
template<typename Tag, typename... A>
    typename proto::result_of::make_expr<Tag, A const...>::type const
    make_expr(A const &...);
template<typename Tag, typename Domain, typename... A>
    typename proto::result_of::make_expr<Tag, Domain, A const...>::type const
    make_expr(A const &...);
template<typename Tag, typename Sequence>
    typename proto::result_of::unpack_expr<Tag, Sequence const>::type const
    unpack_expr(Sequence const &);
template<typename Tag, typename Domain, typename Sequence>
    typename proto::result_of::unpack_expr<Tag, Domain, Sequence const>::type const
    unpack_expr(Sequence const &);
namespace functional {
    template<typename Tag, typename Domain = proto::deduce_domain>
        struct make_expr;
    template<typename Tag, typename Domain = proto::deduce_domain>
        struct unpack_expr;
}
namespace result_of {
    template<typename Tag, typename... A> struct make_expr;

    template<typename Tag, typename Domain, typename... A>
        struct make_expr<Tag, Domain, A...>;

    template<typename Tag, typename Sequence, typename Void = void>
        struct unpack_expr;

    template<typename Tag, typename Domain, typename Sequence>
        struct unpack_expr<Tag, Domain, Sequence>;
}
}
```


Struct template `make_expr`

`boost::proto::functional::make_expr` — A [PolymorphicFunctionObject](#) equivalent to the `proto::make_expr()` function.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename Domain = proto::deduce_domain>
struct make_expr : proto::callable {
    // member classes/structs/unions
    template<typename This, typename... A>
    struct result<This(A...)> :
        proto::result_of::make_expr< Tag, Domain, A... > {
    };

    // public member functions
    template<typename... A>
        typename proto::result_of::make_expr< Tag, Domain, A const... >::type const
        operator()(A const &...) const;
};
```

Description

In all cases, `proto::functional::make_expr<Tag, Domain>()(a...)` is equivalent to `proto::make_expr<Tag, Domain>(a...)`.

`proto::functional::make_expr<Tag>()(a...)` is equivalent to `proto::make_expr<Tag>(a...)`.

`make_expr` public member functions

1.

```
template<typename... A>
    typename proto::result_of::make_expr< Tag, Domain, A const... >::type const
    operator()(A const &... a) const;
```

Construct an expression node with tag type `Tag` and in the domain `Domain`.

Returns: `proto::make_expr<Tag, Domain>(a...)`

Struct template result<This(A...)>

boost::proto::functional::make_expr::result<This(A...)>

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename This, typename... A>
struct result<This(A...)> :
    proto::result_of::make_expr< Tag, Domain, A... > {
};
```


Struct template `unpack_expr`

`boost::proto::functional::unpack_expr` — A [PolymorphicFunctionObject](#) equivalent to the `proto::unpack_expr()` function.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename Domain = proto::deduce_domain>
struct unpack_expr : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Sequence>
    struct result<This(Sequence)> :
        proto::result_of::unpack_expr<
            Tag,
            Domain,
            typename boost::remove_reference< Sequence >::type
        >
    {
    };

    // public member functions
    template<typename Sequence>
    typename proto::result_of::unpack_expr< Tag, Domain, Sequence const >::type const
    operator()(Sequence const &) const;
};
```

Description

In all cases, `proto::functional::unpack_expr<Tag, Domain>()(seq)` is equivalent to `proto::unpack_expr()<Tag, Domain>(seq)`.

`proto::functional::unpack_expr<Tag>()(seq)` is equivalent to `proto::unpack_expr()<Tag>(seq)`.

`unpack_expr` public member functions

1.

```
template<typename Sequence>
    typename proto::result_of::unpack_expr< Tag, Domain, Sequence const >::type const
    operator()(Sequence const & sequence) const;
```

Construct an expression node with tag type `Tag` and in the domain `Domain`.

Parameters: `sequence` A Fusion Forward Sequence

Returns: `proto::unpack_expr<Tag, Domain>(sequence)`

Struct template result<This(Sequence)>

boost::proto::functional::unpack_expr::result<This(Sequence)>

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename This, typename Sequence>
struct result<This(Sequence)> :
    proto::result_of::unpack_expr<
        Tag,
        Domain,
        typename boost::remove_reference< Sequence >::type
    >
{
};
```


Struct template `make_expr`

`boost::proto::result_of::make_expr` — Metafunction that computes the return type of the `proto::make_expr()` function, with a domain deduced from the domains of the children.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename... A>
struct make_expr {
    // types
    typedef domain-deduced-from-child-types D;
    typedef typename proto::result_of::make_expr<Tag, D, A...>::type type;
};
```

Description

Computes the return type of the `proto::make_expr()` function.

In this specialization, the domain is deduced from the domains of the child types. If `proto::is_domain<A0>::value` is true, then another specialization is selected.

`make_expr` public types

1. `typedef domain-deduced-from-child-types D;`

In this specialization, Proto uses the domains of the child expressions to compute the domain of the parent. See `proto::deduce_domain` for a full description of the procedure used.

Struct template `make_expr<Tag, Domain, A...>`

`boost::proto::result_of::make_expr<Tag, Domain, A...>` — Metafunction that computes the return type of the `proto::make_expr()` function, within the specified domain.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename Domain, typename... A>
struct make_expr<Tag, Domain, A...> {
    // types
    typedef see-below type;
};
```

Description

Computes the return type of the `proto::make_expr()` function.

`make_expr` public types

1. typedef *see-below* type;

Let `WRAP<X>` be defined such that:

- If `X` is `Y` & or (possibly cv-qualified) `boost::reference_wrapper<Y>`, then `WRAP<X>` is equivalent to `proto::result_of::as_child<Y, Domain>`.
- Otherwise, `WRAP<X>` is equivalent to `proto::result_of::as_expr<X, Domain>`.

If `proto::wants_basic_expr<typename Domain::proto_generator>::value` is true, then let `E` be `proto::basic_expr`; otherwise, let `E` be `proto::expr`.

If `Tag` is `proto::tag::terminal`, then type is a typedef for `typename WRAP<A0>::type`.

Otherwise, type is a typedef for `boost::result_of<Domain(E< Tag, proto::listN< typename WRAP<A>::type...>>)>::type`

Struct template `unpack_expr`

`boost::proto::result_of::unpack_expr` — Metafunction that computes the return type of the `proto::unpack_expr()` function, with a domain deduced from the domains of the children.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename Sequence, typename Void = void>
struct unpack_expr {
    // types
    typedef
        typename proto::result_of::make_expr<
            Tag,
            typename fusion::result_of::value_at_c<S, 0>::type,
            ...
            typename fusion::result_of::value_at_c<S, N-1>::type
        >::type
    type; // Where S is a Fusion RandomAccessSequence equivalent to Sequence, and N is the size of S.
};
```

Description

Compute the return type of the `proto::unpack_expr()` function.

Sequence is a Fusion Forward Sequence.

In this specialization, the domain is deduced from the domains of the child types. If `proto::is_domain<Sequence>::value` is true, then another specialization is selected.

Struct template `unpack_expr<Tag, Domain, Sequence>`

`boost::proto::result_of::unpack_expr<Tag, Domain, Sequence>` — Metafunction that computes the return type of the `proto::unpack_expr()` function, within the specified domain.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename Domain, typename Sequence>
struct unpack_expr<Tag, Domain, Sequence> {
    // types
    typedef
        typename proto::result_of::make_expr<
            Tag,
            Domain,
            typename fusion::result_of::value_at_c<S, 0>::type,
            ...
            typename fusion::result_of::value_at_c<S, N-1>::type
        >::type
    type; // Where S is a RandomAccessSequence equivalent to Sequence, and N is the size of S.
};
```

Description

Computes the return type of the `proto::unpack_expr()` function.

Function `make_expr`

`boost::proto::make_expr` — Construct an expression of the requested tag type with a domain and with the specified arguments as children.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename... A>
    typename proto::result_of::make_expr<Tag, A const...>::type const
    make_expr(A const &... a);
template<typename Tag, typename Domain, typename... A>
    typename proto::result_of::make_expr<Tag, Domain, A const...>::type const
    make_expr(A const &... a);
```

Description

This function template may be invoked either with or without specifying a Domain template parameter. If no domain is specified, the domain is deduced by examining domains of the given arguments. See [proto::deduce_domain](#) for a full description of the procedure used.

Let $WRAP(x)$ be defined such that:

- If x is a `boost::reference_wrapper<>`, $WRAP(x)$ is equivalent to `proto::as_child<Domain>(x.get())`.
- Otherwise, $WRAP(x)$ is equivalent to `proto::as_expr<Domain>(x)`.

If `proto::wants_basic_expr<typename Domain::proto_generator>::value` is true, then let E be `proto::basic_expr`; otherwise, let E be `proto::expr`.

Let $MAKE(Tag, b...)$ be defined as $E<Tag, proto::listN<decltype(b)...> >::make(b...)$.

If Tag is `proto::tag::terminal`, then return $WRAP(a_0)$.

Otherwise, return $Domain()(MAKE(Tag, WRAP(a)...))$.

Function `unpack_expr`

`boost::proto::unpack_expr` — Construct an expression of the requested tag type with a domain and with children from the specified Fusion Forward Sequence.

Synopsis

```
// In header: <boost/proto/make_expr.hpp>

template<typename Tag, typename Sequence>
    typename proto::result_of::unpack_expr<Tag, Sequence const>::type const
    unpack_expr(Sequence const & sequence);
template<typename Tag, typename Domain, typename Sequence>
    typename proto::result_of::unpack_expr<Tag, Domain, Sequence const>::type const
    unpack_expr(Sequence const & sequence);
```

Description

This function template may be invoked either with or without specifying a Domain argument. If no domain is specified, the domain is deduced by examining domains of each element of the sequence. See [proto::deduce_domain](#) for a full description of the procedure used.

Let s be a Fusion RandomAccessSequence equivalent to `sequence`. Let $WRAP(N, s)$ be defined such that:

- If `fusion::result_of::value_at_c<decltype(s), N>::type` is a reference type or an instantiation of `boost::reference_wrapper<>`, $WRAP(N, s)$ is equivalent to `proto::as_child<Domain>(fusion::at_c<N>(s))`.
- Otherwise, $WRAP(N, s)$ is equivalent to `proto::as_expr<Domain>(fusion::at_c<N>(s))`.

If `proto::wants_basic_expr<typename Domain::proto_generator>::value` is true, then let E be `proto::basic_expr`; otherwise, let E be `proto::expr`.

Let $MAKE(Tag, b...)$ be defined as $E<Tag, proto::listN<decltype(b)...>::make(b...)>$.

If Tag is `proto::tag::terminal`, then return $WRAP(0, s)$.

Otherwise, return `Domain()(MAKE(Tag, WRAP(0, s), ... WRAP(N-1, s)))`, where N is the size of `Sequence`.

Parameters: `sequence` A Fusion Forward Sequence.

Header `<boost/proto/matches.hpp>`

Contains definition of the `proto::matches<>` metafunction for determining if a given expression matches a given pattern.


```
namespace boost {  
  namespace proto {  
    struct _;  
    template<typename Grammar> struct not_;  
    template<typename If, typename Then = proto::_,  
            typename Else = proto::not_<proto::_> >  
      struct if_;  
    template<typename... G> struct or_;  
    template<typename... G> struct and_;  
    template<typename Cases> struct switch_;  
    template<typename T> struct exact;  
    template<typename T> struct convertible_to;  
    template<typename Grammar> struct vararg;  
    template<typename Expr, typename Grammar> struct matches;  
  }  
}
```


Struct `_`

`boost::proto::_` — A wildcard grammar element that matches any expression, and a transform that returns the current expression unchanged.

Synopsis

```
// In header: <boost/proto/matches.hpp>

struct _ : proto::transform<_> {
    // types
    typedef _ proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl<Expr, State, Data> {
        // types
        typedef Expr result_type;

        // public member functions
        Expr operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

The wildcard type, `proto::_`, is a grammar element such that `proto::matches<E, proto::_>::value` is true for any expression type `E`.

The wildcard can also be used as a stand-in for a template argument when matching terminals. For instance, the following is a grammar that will match any `std::complex<>` terminal:

```
BOOST_MPL_ASSERT((
    proto::matches<
        proto::terminal<std::complex<double> >::type,
        proto::terminal<std::complex< proto::_ > >
    >
));
```

When used as a transform, `proto::_` returns the current expression unchanged. For instance, in the following, `proto::_` is used with the `proto::fold<>` transform to fold the children of a node:

```
struct CountChildren :
    proto::or_<
        // Terminals have no children
        proto::when<proto::terminal<proto::_>, mpl::int_<0>()>,
        // Use proto::fold<> to count the children of non-terminals
        proto::otherwise<
            proto::fold<
                proto::_, // <-- fold the current expression
                mpl::int_<0>(),
                mpl::plus<proto::_state, mpl::int_<1> >()
            >
        >
    >
{};
```


Struct template impl

boost::proto::_::impl

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl<Expr, State, Data> {
    // types
    typedef Expr result_type;

    // public member functions
    Expr operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1. Expr operator()(typename impl::expr_param expr, typename impl::state_param,
 typename impl::data_param) const;

Parameters: expr An expression
Returns: expr

Struct template not_

`boost::proto::not_` — Inverts the set of expressions matched by a grammar. When used as a transform, `proto::not_<>` returns the current expression unchanged.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Grammar>
struct not_ : proto::transform<not_<Grammar> > {
    // types
    typedef not_ proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl<Expr, State, Data> {
        // types
        typedef Expr result_type;

        // public member functions
        Expr operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

If an expression type *E* does not match a grammar *G*, then *E* *does* match `proto::not_<G>`. For example, `proto::not_<proto::terminal<proto::_> >` will match any non-terminal.

Struct template impl

boost::proto::not_::impl

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl<Expr, State, Data> {
    // types
    typedef Expr result_type;

    // public member functions
    Expr operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1. Expr operator()(typename impl::expr_param expr, typename impl::state_param,
 typename impl::data_param) const;

Parameters: expr An expression
Requires: proto::matches<Expr, proto::not_>::value is true.
Returns: expr

Struct template if_

`boost::proto::if_` — Used to select one grammar or another based on the result of a compile-time Boolean. When used as a transform, `proto::if_<>` selects between two transforms based on a compile-time Boolean.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename If, typename Then = proto::_,
        typename Else = proto::not_<proto::_> >
struct if_ : proto::transform<if_<If, Then, Else> > {
    // types
    typedef if_ proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef typename mpl::if_<
            typename boost::result_of<proto::when<proto::_>, If>(Expr, State, Data)>::type,
            typename boost::result_of<proto::when<proto::_>, Then>(Expr, State, Data)>::type,
            typename boost::result_of<proto::when<proto::_>, Else>(Expr, State, Data)>::type
        >::type result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

When `proto::if_<If, Then, Else>` is used as a grammar, `If` must be a Proto transform and `Then` and `Else` must be grammars. An expression type `E` matches `proto::if_<If, Then, Else>` if `boost::result_of<proto::when<proto::_>, If>(E)>::type::value` is true and `E` matches `Then`; or, if `boost::result_of<proto::when<proto::_>, If>(E)>::type::value` is false and `E` matches `Else`.

The template parameter `Then` defaults to `proto::_` and `Else` defaults to `proto::not_<proto::_>`, so an expression type `E` will match `proto::if_<If>` if and only if `boost::result_of<proto::when<proto::_>, If>(E)>::type::value` is true.

```
// A grammar that only matches integral terminals,
// using is_integral<> from Boost.Type_traits.
struct IsIntegral :
    proto::and_<
        proto::terminal<proto::_>,
        proto::if_< boost::is_integral<proto::_value>() >
    >
{};
```

When `proto::if_<If, Then, Else>` is used as a transform, `If`, `Then` and `Else` must be Proto transforms. When applying the transform to an expression `E`, state `S` and data `V`, if `boost::result_of<proto::when<proto::_>, If>(E,S,V)>::type::value` is true then the `Then` transform is applied; otherwise the `Else` transform is applied.


```
// Match a terminal. If the terminal is integral, return
// mpl::true_; otherwise, return mpl::false_.
struct IsIntegral2 :
    proto::when<
        proto::terminal<_>,
        proto::if_<
            boost::is_integral<proto::_value>(),
            mpl::true_(),
            mpl::false_()
        >
    >
{};
```


Struct template impl

boost::proto::if_::impl

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef typename mpl::if_<
        typename boost::result_of<proto::when<proto::_>, If>(Expr, State, Data)>::type,
        typename boost::result_of<proto::when<proto::_>, Then>(Expr, State, Data)>::type,
        typename boost::result_of<proto::when<proto::_>, Else>(Expr, State, Data)>::type
    >::type result_type;

    // public member functions
    result_type operator()(typename impl::expr_param expr,
                          typename impl::state_param state,
                          typename impl::data_param data) const;
};
```

Description

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                      typename impl::state_param state,
                      typename impl::data_param data) const;
```

Parameters: data A data of arbitrary type
 expr An expression
 state The current state

Returns: `proto::when<proto::_>, Then-or-Else>()(expr, state, data)`

Struct template or_

`boost::proto::or_` — For matching one of a set of alternate grammars. Alternates are tried in order to avoid ambiguity. When used as a transform, `proto::or_<>` applies the transform associated with the first grammar that matches the expression.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename... G>
struct or_ : proto::transform<or_<G...> > {
    // types
    typedef or_ proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef unspecified result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

An expression type E matches `proto::or_< G_0, G_1, \dots, G_n >` if E matches any G_x for x in $[0, n]$.

When applying `proto::or_< G_0, G_1, \dots, G_n >` as a transform with an expression e of type E , state s and data d , it is equivalent to $G_x()(e, s, d)$, where x is the lowest number such that `proto::matches< E, G_x >::value` is true.

The maximum number of template arguments `proto::or_<>` accepts is controlled by the `BOOST_PROTO_MAX_LOGICAL_ARITY` macro.

Struct template impl

boost::proto::or_::impl

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef unspecified result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                           typename impl::state_param,
                           typename impl::data_param) const;
};
```

Description

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                       typename impl::state_param state,
                       typename impl::data_param data) const;
```

Parameters:

data	A data of arbitrary type
expr	An expression
state	The current state

Returns: $G_x()(\text{expr}, \text{state}, \text{data})$, where x is the lowest number such that `proto::matches<Expr, G_x >::value` is true.

Struct template and_

`boost::proto::and_` — For matching all of a set of grammars. When used as a transform, `proto::and_<>` applies the transform associated with each grammar in the set and returns the result of the last.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename... G>
struct and_ : proto::transform<and_<G...> > {
    // types
    typedef and_ proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef typename boost::result_of<Gn(Expr, State, Data)>::type result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

An expression type E matches `proto::and_< G_0, G_1, \dots, G_n >` if E matches all G_x for x in $[0, n]$.

When applying `proto::and_< G_0, G_1, \dots, G_n >` as a transform with an expression e , state s and data d , it is equivalent to $(G_0())(e, s, d), G_1()(e, s, d), \dots, G_n()(e, s, d)$.

The maximum number of template arguments `proto::and_<>` accepts is controlled by the `BOOST_PROTO_MAX_LOGICAL_ARITY` macro.

Struct template impl

boost::proto::and_::impl

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef typename boost::result_of<Gn(Expr, State, Data)>::type result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                           typename impl::state_param,
                           typename impl::data_param) const;
};
```

Description

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                       typename impl::state_param state,
                       typename impl::data_param data) const;
```

Parameters: data A data of arbitrary type
 expr An expression
 state The current state

Returns: (G₀())(expr, state, data), G₁()(expr, state, data), ..., G_n()(expr, state, data))

Struct template `switch_`

`boost::proto::switch_` — For matching one of a set of alternate grammars, which are looked up based on an expression's tag type. When used as a transform, `proto::switch_<>` applies the transform associated with the sub-grammar that matches the expression.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Cases>
struct switch_ : proto::transform<switch_<Cases> > {
    // types
    typedef switch_ proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        Cases::template case_<typename Expr::tag_type>::template impl<Expr, State, Data>
    {
    };
};
```

Description

An expression type `E` matches `proto::switch_<C>` if `E` matches `C::case_<E::proto_tag>`.

When applying `proto::switch_<C>` as a transform with an expression `e` of type `E`, state `s` and data `d`, it is equivalent to `C::case_<E::proto_tag>()(e, s, d)`.

Struct template impl

boost::proto::switch_::impl

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    Cases::template case_<typename Expr::tag_type>::template impl<Expr, State, Data>
{
};
```


Struct template exact

boost::proto::exact — For forcing exact matches of terminal types.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename T>
struct exact {
};
```

Description

By default, matching terminals ignores references and cv-qualifiers. For instance, a terminal expression of type `proto::terminal<int const &>` type will match the grammar `proto::terminal<int>`. If that is not desired, you can force an exact match with `proto::terminal<proto::exact<int> >`. This will only match integer terminals where the terminal is held by value.

Struct template convertible_to

boost::proto::convertible_to — For matching terminals that are convertible to a type.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename T>
struct convertible_to {
};
```

Description

Use `proto::convertible_to<>` to match a terminal that is convertible to some type. For example, the grammar `proto::terminal<proto::convertible_to<int> >` will match any terminal whose argument is convertible to an integer.

Struct template vararg

boost::proto::vararg — For matching a Grammar to a variable number of sub-expressions.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Grammar>
struct vararg {
};
```

Description

An expression type `proto::basic_expr<AT, proto::listN<A0, ..., An, U0, ..., Um> >` matches a grammar `proto::basic_expr<BT, proto::listM<B0, ..., Bn, proto::vararg<V> > >` if BT is `proto::_` or AT, and if A_x matches B_x for each x in [0, n] and if U_x matches V for each x in [0, m].

For example:

```
// Match any function call expression, regardless
// of the number of function arguments:
struct Function :
    proto::function< proto::vararg<proto::_> >
{ };
```

When used as a transform, `proto::vararg<G>` applies G's transform.

Struct template matches

`boost::proto::matches` — A Boolean metafunction that evaluates whether a given expression type matches a grammar.

Synopsis

```
// In header: <boost/proto/matches.hpp>

template<typename Expr, typename Grammar>
struct matches : mpl::bool_<true-or-false> {
};
```

Description

`proto::matches<Expr, Grammar>` inherits from `mpl::true_` if `Expr::proto_grammar` matches `Grammar::proto_grammar`, and from `mpl::false_` otherwise.

Non-terminal expressions are matched against a grammar according to the following rules:

- The wildcard pattern, `proto::_`, matches any expression.
- An expression `proto::basic_expr<AT, proto::listN <A0, ..., An> >` matches a grammar `proto::basic_expr<BT, proto::listN <B0, ..., Bn> >` if BT is `proto::_` or AT, and if `Ax` matches `Bx` for each `x` in `[0, n]`.
- An expression `proto::basic_expr<AT, proto::listN <A0, ..., An, U0, ..., Um> >` matches a grammar `proto::basic_expr<BT, proto::listM <B0, ..., Bn, proto::vararg<V> > >` if BT is `proto::_` or AT, and if `Ax` matches `Bx` for each `x` in `[0, n]` and if `Ux` matches `V` for each `x` in `[0, m]`.
- An expression `E` matches `proto::or_<B0, ..., Bn>` if `E` matches some `Bx` for `x` in `[0, n]`.
- An expression `E` matches `proto::and_<B0, ..., Bn>` if `E` matches all `Bx` for `x` in `[0, n]`.
- An expression `E` matches `proto::if_<T, U, V>` if:
 - `boost::result_of<proto::when<proto::_>(E)>::type::value` is true and `E` matches `U`, or
 - `boost::result_of<proto::when<proto::_>(E)>::type::value` is false and `E` matches `V`.
 Note: `U` defaults to `proto::_` and `V` defaults to `proto::not_<proto::_>`.
- An expression `E` matches `proto::not_<T>` if `E` does *not* match `T`.
- An expression `E` matches `proto::switch_<C>` if `E` matches `C::case_<E::proto_tag>`.

A terminal expression `proto::basic_expr<AT, proto::term<A> >` matches a grammar `proto::basic_expr<BT, proto::term >` if BT is `proto::_` or AT and one of the following is true:

- `B` is the wildcard pattern, `proto::_`
- `A` is `B`
- `A` is `B &`
- `A` is `B const &`
- `B` is `proto::exact<A>`
- `B` is `proto::convertible_to<X>` and `boost::is_convertible<A, X>::value` is true.
- `A` is `X[M]` or `X(&)[M]` and `B` is `X[proto::N]`.

- A is $X(\&)[M]$ and B is $X(\&)[\text{proto}::N]$.
- A is $X[M]$ or $X(\&)[M]$ and B is X^* .
- B *lambda-matches* A (see below).

A type B *lambda-matches* A if one of the following is true:

- B is A
- B is the wildcard pattern, `proto::_`
- B is $T\langle B_0, \dots, B_n \rangle$ and A is $T\langle A_0, \dots, A_n \rangle$ and for each x in $[0, n]$, A_x and B_x are types such that A_x *lambda-matches* B_x

Header `<boost/proto/operators.hpp>`

Contains all the overloaded operators that make it possible to build Proto expression trees.

```
BOOST_PROTO_DEFINE_OPERATORS(Trait, Domain)
```



```

namespace boost {
namespace proto {
template<typename T> struct is_extension;
template<typename Arg> unspecified operator+(Arg & arg);
template<typename Arg> unspecified operator+(Arg const & arg);
template<typename Arg> unspecified operator-(Arg & arg);
template<typename Arg> unspecified operator-(Arg const & arg);
template<typename Arg> unspecified operator*(Arg & arg);
template<typename Arg> unspecified operator*(Arg const & arg);
template<typename Arg> unspecified operator~(Arg & arg);
template<typename Arg> unspecified operator~(Arg const & arg);
template<typename Arg> unspecified operator&(Arg & arg);
template<typename Arg> unspecified operator&(Arg const & arg);
template<typename Arg> unspecified operator!(Arg & arg);
template<typename Arg> unspecified operator!(Arg const & arg);
template<typename Arg> unspecified operator++(Arg & arg);
template<typename Arg> unspecified operator++(Arg const & arg);
template<typename Arg> unspecified operator--(Arg & arg);
template<typename Arg> unspecified operator--(Arg const & arg);
template<typename Arg> unspecified operator++(Arg & arg, int);
template<typename Arg> unspecified operator++(Arg const & arg, int);
template<typename Arg> unspecified operator--(Arg & arg, int);
template<typename Arg> unspecified operator--(Arg const & arg, int);
template<typename Left, typename Right>
unspecified operator<<(Left & left, Right & right);
template<typename Left, typename Right>
unspecified operator<<(Left & left, Right const & right);
template<typename Left, typename Right>
unspecified operator<<(Left const & left, Right & right);
template<typename Left, typename Right>
unspecified operator<<(Left const & left, Right const & right);
template<typename Left, typename Right>
unspecified operator>>(Left & left, Right & right);
template<typename Left, typename Right>
unspecified operator>>(Left & left, Right const & right);
template<typename Left, typename Right>
unspecified operator>>(Left const & left, Right & right);
template<typename Left, typename Right>
unspecified operator>>(Left const & left, Right const & right);
template<typename Left, typename Right>
unspecified operator*(Left & left, Right & right);
template<typename Left, typename Right>
unspecified operator*(Left & left, Right const & right);
template<typename Left, typename Right>
unspecified operator*(Left const & left, Right & right);
template<typename Left, typename Right>
unspecified operator*(Left const & left, Right const & right);
template<typename Left, typename Right>
unspecified operator/(Left & left, Right & right);
template<typename Left, typename Right>
unspecified operator/(Left & left, Right const & right);
template<typename Left, typename Right>
unspecified operator/(Left const & left, Right & right);
template<typename Left, typename Right>
unspecified operator/(Left const & left, Right const & right);
template<typename Left, typename Right>
unspecified operator%(Left & left, Right & right);
template<typename Left, typename Right>
unspecified operator%(Left & left, Right const & right);
template<typename Left, typename Right>
unspecified operator%(Left const & left, Right & right);
template<typename Left, typename Right>
unspecified operator%(Left const & left, Right const & right);

```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at <http://www.renderx.com/>

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at <http://www.renderx.com/>

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at <http://www.renderx.com/>


```
    unspecified operator|=(Left const & left, Right & right);
template<typename Left, typename Right>
    unspecified operator|=(Left const & left, Right const & right);
template<typename Left, typename Right>
    unspecified operator^=(Left & left, Right & right);
template<typename Left, typename Right>
    unspecified operator^=(Left & left, Right const & right);
template<typename Left, typename Right>
    unspecified operator^=(Left const & left, Right & right);
template<typename Left, typename Right>
    unspecified operator^=(Left const & left, Right const & right);
template<typename A0, typename A1, typename A2>
    typename proto::result_of::make_expr<
        proto::tag::if_else_,
        proto::deduce_domain,
        A0 const &,
        A1 const &,
        A2 const &
    >::type const
    if_else(A0 const & a0, A1 const & a1, A2 const & a2);
}
}
```


Struct template `is_extension`

`boost::proto::is_extension` — Boolean metafunction that can be used to enable the operator overloads in the `exops` namespace for the specified non-Proto terminal type.

Synopsis

```
// In header: <boost/proto/operators.hpp>

template<typename T>
struct is_extension : is_expr< T > {
};
```


Macro `BOOST_PROTO_DEFINE_OPERATORS`

`BOOST_PROTO_DEFINE_OPERATORS` — Defines a complete set of expression template-building operator overloads for use with non-Proto terminal types.

Synopsis

```
// In header: <boost/proto/operators.hpp>

BOOST_PROTO_DEFINE_OPERATORS(Trait, Domain)
```

Description

With `BOOST_PROTO_DEFINE_OPERATORS()`, it is possible to non-intrusively adapt an existing (non-Proto) type to be a Proto terminal.

`Trait` is the name of a unary Boolean metafunction that returns true for any types you would like to treat as Proto terminals.

`Domain` is the name of the Proto domain associated with these new Proto terminals. You may use `proto::default_domain` for the `Domain` if you do not wish to associate these terminals with any domain.

Example:

```
namespace My {
    // A non-Proto terminal type
    struct S {};

    // A unary Boolean metafunction that returns true for type S
    template<typename T> struct IsS : mpl::false_ {};
    template<> struct IsS<S> : mpl::true_ {};

    // Make S a Proto terminal non-intrusively by defining the
    // appropriate operator overloads. This should be in the same
    // namespace as S so that these overloads can be found by
    // argument-dependent lookup
    BOOST_PROTO_DEFINE_OPERATORS(IsS, proto::default_domain)
}

int main() {
    My::S s1, s2;

    // OK, this builds a Proto expression template:
    s1 + s2;
}
```

Header `<boost/proto/proto.hpp>`

Includes all of Proto, except the `Boost.Typeof` registrations.

Header `<boost/proto/proto_fwd.hpp>`

Forward declarations of all of proto's public types and functions.


```
BOOST_PROTO_MAX_ARITY
BOOST_PROTO_MAX_LOGICAL_ARITY
BOOST_PROTO_MAX_FUNCTION_CALL_ARITY
```

```
namespace boost {
  namespace proto {
    struct callable;

    int const N;

    typedef proto::functional::flatten _flatten;
    typedef proto::functional::pop_front _pop_front;
    typedef proto::functional::reverse _reverse;
    typedef proto::functional::eval _eval;
    typedef proto::functional::deep_copy _deep_copy;
    typedef proto::functional::make_expr< proto::tag::terminal > _make_terminal;
    typedef proto::functional::make_expr< proto::tag::unary_plus > _make_unary_plus;
    typedef proto::functional::make_expr< proto::tag::negate > _make_negate;
    typedef proto::functional::make_expr< proto::tag::dereference > _make_dereference;
    typedef proto::functional::make_expr< proto::tag::complement > _make_complement;
    typedef proto::functional::make_expr< proto::tag::address_of > _make_address_of;
    typedef proto::functional::make_expr< proto::tag::logical_not > _make_logical_not;
    typedef proto::functional::make_expr< proto::tag::pre_inc > _make_pre_inc;
    typedef proto::functional::make_expr< proto::tag::pre_dec > _make_pre_dec;
    typedef proto::functional::make_expr< proto::tag::post_inc > _make_post_inc;
    typedef proto::functional::make_expr< proto::tag::post_dec > _make_post_dec;
    typedef proto::functional::make_expr< proto::tag::shift_left > _make_shift_left;
    typedef proto::functional::make_expr< proto::tag::shift_right > _make_shift_right;
    typedef proto::functional::make_expr< proto::tag::multiplies > _make_multiplies;
    typedef proto::functional::make_expr< proto::tag::divides > _make_divides;
    typedef proto::functional::make_expr< proto::tag::modulus > _make_modulus;
    typedef proto::functional::make_expr< proto::tag::plus > _make_plus;
    typedef proto::functional::make_expr< proto::tag::minus > _make_minus;
    typedef proto::functional::make_expr< proto::tag::less > _make_less;
    typedef proto::functional::make_expr< proto::tag::greater > _make_greater;
    typedef proto::functional::make_expr< proto::tag::less_equal > _make_less_equal;
    typedef proto::functional::make_expr< proto::tag::greater_equal > _make_greater_equal;
    typedef proto::functional::make_expr< proto::tag::equal_to > _make_equal_to;
    typedef proto::functional::make_expr< proto::tag::not_equal_to > _make_not_equal_to;
    typedef proto::functional::make_expr< proto::tag::logical_or > _make_logical_or;
    typedef proto::functional::make_expr< proto::tag::logical_and > _make_logical_and;
    typedef proto::functional::make_expr< proto::tag::bitwise_and > _make_bitwise_and;
    typedef proto::functional::make_expr< proto::tag::bitwise_or > _make_bitwise_or;
    typedef proto::functional::make_expr< proto::tag::bitwise_xor > _make_bitwise_xor;
    typedef proto::functional::make_expr< proto::tag::comma > _make_comma;
    typedef proto::functional::make_expr< proto::tag::mem_ptr > _make_mem_ptr;
    typedef proto::functional::make_expr< proto::tag::assign > _make_assign;
    typedef proto::functional::make_expr< proto::tag::shift_left_assign > _make_shift_left_assign;
    typedef proto::functional::make_expr< proto::tag::shift_right_assign > _make_shift_right_assign;
    sign;
    typedef proto::functional::make_expr< proto::tag::multiplies_assign > _make_multiplies_assign;
    typedef proto::functional::make_expr< proto::tag::divides_assign > _make_divides_assign;
    typedef proto::functional::make_expr< proto::tag::modulus_assign > _make_modulus_assign;
    typedef proto::functional::make_expr< proto::tag::plus_assign > _make_plus_assign;
    typedef proto::functional::make_expr< proto::tag::minus_assign > _make_minus_assign;
    typedef proto::functional::make_expr< proto::tag::bitwise_and_assign > _make_bitwise_and_assign;
    sign;
    typedef proto::functional::make_expr< proto::tag::bitwise_or_assign > _make_bitwise_or_assign;
    typedef proto::functional::make_expr< proto::tag::bitwise_xor_assign > _make_bitwise_xor_assign;
    sign;
```



```

typedef proto::functional::make_expr< proto::tag::subscript > _make_subscript;
typedef proto::functional::make_expr< proto::tag::if_else_ > _make_if_else;
typedef proto::functional::make_expr< proto::tag::function > _make_function;
typedef proto::_child_c< N > _childN; // For each N in [0,BOOST_PROTO_MAX_ARITY)
typedef proto::_child0 _child;
typedef proto::_child0 _left;
typedef proto::_child1 _right;
namespace functional {
    typedef proto::functional::make_expr< proto::tag::terminal > make_terminal;
    typedef proto::functional::make_expr< proto::tag::unary_plus > make_unary_plus;
    typedef proto::functional::make_expr< proto::tag::negate > make_negate;
    typedef proto::functional::make_expr< proto::tag::dereference > make_dereference;
    typedef proto::functional::make_expr< proto::tag::complement > make_complement;
    typedef proto::functional::make_expr< proto::tag::address_of > make_address_of;
    typedef proto::functional::make_expr< proto::tag::logical_not > make_logical_not;
    typedef proto::functional::make_expr< proto::tag::pre_inc > make_pre_inc;
    typedef proto::functional::make_expr< proto::tag::pre_dec > make_pre_dec;
    typedef proto::functional::make_expr< proto::tag::post_inc > make_post_inc;
    typedef proto::functional::make_expr< proto::tag::post_dec > make_post_dec;
    typedef proto::functional::make_expr< proto::tag::shift_left > make_shift_left;
    typedef proto::functional::make_expr< proto::tag::shift_right > make_shift_right;
    typedef proto::functional::make_expr< proto::tag::multiplies > make_multiplies;
    typedef proto::functional::make_expr< proto::tag::divides > make_divides;
    typedef proto::functional::make_expr< proto::tag::modulus > make_modulus;
    typedef proto::functional::make_expr< proto::tag::plus > make_plus;
    typedef proto::functional::make_expr< proto::tag::minus > make_minus;
    typedef proto::functional::make_expr< proto::tag::less > make_less;
    typedef proto::functional::make_expr< proto::tag::greater > make_greater;
    typedef proto::functional::make_expr< proto::tag::less_equal > make_less_equal;
    typedef proto::functional::make_expr< proto::tag::greater_equal > make_greater_equal;
    typedef proto::functional::make_expr< proto::tag::equal_to > make_equal_to;
    typedef proto::functional::make_expr< proto::tag::not_equal_to > make_not_equal_to;
    typedef proto::functional::make_expr< proto::tag::logical_or > make_logical_or;
    typedef proto::functional::make_expr< proto::tag::logical_and > make_logical_and;
    typedef proto::functional::make_expr< proto::tag::bitwise_and > make_bitwise_and;
    typedef proto::functional::make_expr< proto::tag::bitwise_or > make_bitwise_or;
    typedef proto::functional::make_expr< proto::tag::bitwise_xor > make_bitwise_xor;
    typedef proto::functional::make_expr< proto::tag::comma > make_comma;
    typedef proto::functional::make_expr< proto::tag::mem_ptr > make_mem_ptr;
    typedef proto::functional::make_expr< proto::tag::assign > make_assign;
    typedef proto::functional::make_expr< proto::tag::shift_left_assign > make_shift_left_assign;
    typedef proto::functional::make_expr< proto::tag::shift_right_assign > make_shift_right_assign;
sign;
    typedef proto::functional::make_expr< proto::tag::multiplies_assign > make_multiplies_assign;
    typedef proto::functional::make_expr< proto::tag::divides_assign > make_divides_assign;
    typedef proto::functional::make_expr< proto::tag::modulus_assign > make_modulus_assign;
    typedef proto::functional::make_expr< proto::tag::plus_assign > make_plus_assign;
    typedef proto::functional::make_expr< proto::tag::minus_assign > make_minus_assign;
    typedef proto::functional::make_expr< proto::tag::bitwise_and_assign > make_bitwise_and_assign;
sign;
    typedef proto::functional::make_expr< proto::tag::bitwise_or_assign > make_bitwise_or_assign;
    typedef proto::functional::make_expr< proto::tag::bitwise_xor_assign > make_bitwise_xor_assign;
sign;
    typedef proto::functional::make_expr< proto::tag::subscript > make_subscript;
    typedef proto::functional::make_expr< proto::tag::if_else_ > make_if_else;
    typedef proto::functional::make_expr< proto::tag::function > make_function;
}
}
}

```


Struct callable

boost::proto::callable — Base class for callable [PolymorphicFunctionObjects](#)

Synopsis

```
// In header: <boost/proto/proto_fwd.hpp>

struct callable {
};
```

Description

When defining a callable [PolymorphicFunctionObject](#), inherit from `proto::callable` so that it can be used to create a [Callable-Transform](#).

`proto::is_callable<T>::value` is true for types that inherit from `proto::callable`.

Global N

boost::proto::N

Synopsis

```
// In header: <boost/proto/proto_fwd.hpp>

int const N;
```

Description

Array size wildcard for Proto grammars that match array terminals.

Macro **BOOST_PROTO_MAX_ARITY**

BOOST_PROTO_MAX_ARITY — Controls the maximum number of child nodes an expression may have.

Synopsis

```
// In header: <boost/proto/proto_fwd.hpp>

BOOST_PROTO_MAX_ARITY
```

Description

BOOST_PROTO_MAX_ARITY defaults to 5. It may be set higher or lower, but not lower than 3. Setting it higher will have a negative effect on compile times.

See also [BOOST_PROTO_MAX_FUNCTION_CALL_ARITY](#).

Macro `BOOST_PROTO_MAX_LOGICAL_ARITY`

`BOOST_PROTO_MAX_LOGICAL_ARITY` — Controls the maximum number of sub-grammars that `proto::or_<>` and `proto::and_<>` accept.

Synopsis

```
// In header: <boost/proto/proto_fwd.hpp>

BOOST_PROTO_MAX_LOGICAL_ARITY
```

Description

`BOOST_PROTO_MAX_LOGICAL_ARITY` defaults to 8. It may be set higher or lower. Setting it higher will have a negative effect on compile times.

Macro `BOOST_PROTO_MAX_FUNCTION_CALL_ARITY`

`BOOST_PROTO_MAX_FUNCTION_CALL_ARITY` — Controls the maximum number of arguments that `operator()` overloads accept.

Synopsis

```
// In header: <boost/proto/proto_fwd.hpp>

BOOST_PROTO_MAX_FUNCTION_CALL_ARITY
```

Description

When setting `BOOST_PROTO_MAX_ARITY` higher than the default, compile times slow down considerably. That is due in large part to the explosion in the number of `operator()` overloads that must be generated for each Proto expression type. By setting `BOOST_PROTO_MAX_FUNCTION_CALL_ARITY` lower than `BOOST_PROTO_MAX_ARITY`, compile times can be sped up considerably.

Header `<boost/proto/proto_typeof.hpp>`

Boost.Typeof registrations for Proto's types, and definition of the `BOOST_PROTO_AUTO()` macro.

```
BOOST_PROTO_AUTO(Var, Expr)
```


Macro BOOST_PROTO_AUTO

BOOST_PROTO_AUTO — For defining a local variable that stores a Proto expression template, deep-copying the expression so there are no dangling references.

Synopsis

```
// In header: <boost/proto/proto_typeof.hpp>

BOOST_PROTO_AUTO(Var, Expr)
```

Description

To define a local variable `ex` that stores the expression `proto::lit(1) + 2`, do the following:

```
BOOST_PROTO_AUTO( ex, proto::lit(1) + 2 );
```

. The above is equivalent to the following:

```
BOOST_AUTO( ex, proto::deep_copy( proto::lit(1) + 2 ) );
```

Header <boost/proto/repeat.hpp>

Contains macros to ease the generation of repetitious code constructs.

```
BOOST_PROTO_REPEAT(MACRO)
BOOST_PROTO_REPEAT_FROM_TO(FROM, TO, MACRO)
BOOST_PROTO_REPEAT_EX(MACRO, typename_A, A, A_a, a)
BOOST_PROTO_REPEAT_FROM_TO_EX(FROM, TO, MACRO, typename_A, A, A_a, a)
BOOST_PROTO_LOCAL_ITERATE( )
BOOST_PROTO_typename_A(N)
BOOST_PROTO_A_const_ref(N)
BOOST_PROTO_A_ref(N)
BOOST_PROTO_A(N)
BOOST_PROTO_A_const(N)
BOOST_PROTO_A_const_ref_a(N)
BOOST_PROTO_A_ref_a(N)
BOOST_PROTO_ref_a(N)
BOOST_PROTO_a(N)
```


Macro `BOOST_PROTO_REPEAT`

`BOOST_PROTO_REPEAT` — Repeatedly invoke the specified macro.

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_REPEAT(MACRO)
```

Description

`BOOST_PROTO_REPEAT()` is used to generate the kind of repetitive code that is typical of DSELs built with Proto. `BOOST_PROTO_REPEAT(MACRO)` is equivalent to:

```
MACRO(1, BOOST_PROTO_type_1
name_A, BOOST_PROTO_A_const_ref, BOOST_PROTO_A_const_ref_a, BOOST_PROTO_ref_a)
MACRO(2, BOOST_PROTO_type_1
name_A, BOOST_PROTO_A_const_ref, BOOST_PROTO_A_const_ref_a, BOOST_PROTO_ref_a)
* * *
MACRO(BOOST_PROTO_MAX_ARITY, BOOST_PROTO_type_1
name_A, BOOST_PROTO_A_const_ref, BOOST_PROTO_A_const_ref_a, BOOST_PROTO_ref_a)
```

Example:

See `BOOST_PROTO_REPEAT_FROM_TO()`.

Macro BOOST_PROTO_REPEAT_FROM_TO

BOOST_PROTO_REPEAT_FROM_TO — Repeatedly invoke the specified macro.

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_REPEAT_FROM_TO(FROM, TO, MACRO)
```

Description

BOOST_PROTO_REPEAT_FROM_TO() is used to generate the kind of repetitive code that is typical of DSELs built with Proto.

BOOST_PROTO_REPEAT_FROM_TO(FROM, TO, MACRO) is equivalent to:

```
MACRO(FROM, BOOST_PROTO_type_↓
name_A, BOOST_PROTO_A_const_ref, BOOST_PROTO_A_const_ref_a, BOOST_PROTO_ref_a)
MACRO(FROM+1, BOOST_PROTO_type_↓
name_A, BOOST_PROTO_A_const_ref, BOOST_PROTO_A_const_ref_a, BOOST_PROTO_ref_a)
...
MACRO(TO-1, BOOST_PROTO_type_↓
name_A, BOOST_PROTO_A_const_ref, BOOST_PROTO_A_const_ref_a, BOOST_PROTO_ref_a)
```

Example:

```
// Generate BOOST_PROTO_MAX_ARITY-1 overloads of the
// following construct() function template.
#define M0(N, typename_A, A_const_ref, A_const_ref_a, ref_a)      \
template<typename T, typename_A(N)>                                \
typename proto::result_of::make_expr<                             \
    proto::tag::function                                           \
    , construct_helper<T>                                           \
    , A_const_ref(N)                                               \
>::type const                                                      \
construct(A_const_ref_a(N))                                         \
{                                                                    \
    return proto::make_expr<                                       \
        proto::tag::function                                       \
    >(\                                                                \
        construct_helper<T>()                                       \
        , ref_a(N)                                                 \
    );                                                              \
}                                                                    \
BOOST_PROTO_REPEAT_FROM_TO(1, BOOST_PROTO_MAX_ARITY, M0)
#undef M0
```

The above invocation of BOOST_PROTO_REPEAT_FROM_TO() will generate the following code:


```
template<typename T, typename A0>
typename proto::result_of::make_expr<
    proto::tag::function
    , construct_helper<T>
    , A0 const &
>::type const
construct(A0 const & a0)
{
    return proto::make_expr<
        proto::tag::function
    >(
        construct_helper<T>()
        , boost::ref(a0)
    );
}

template<typename T, typename A0, typename A1>
typename proto::result_of::make_expr<
    proto::tag::function
    , construct_helper<T>
    , A0 const &
    , A1 const &
>::type const
construct(A0 const & a0, A1 const & a1)
{
    return proto::make_expr<
        proto::tag::function
    >(
        construct_helper<T>()
        , boost::ref(a0)
        , boost::ref(a1)
    );
}
```


Macro BOOST_PROTO_REPEAT_EX

BOOST_PROTO_REPEAT_EX — Repeatedly invoke the specified macro.

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_REPEAT_EX(MACRO, typename_A, A, A_a, a)
```

Description

BOOST_PROTO_REPEAT_EX() is used to generate the kind of repetitive code that is typical of DSELs built with Proto. BOOST_PROTO_REPEAT_EX(MACRO, typename_A, A, A_a, a) is equivalent to:

```
MACRO(1, typename_A, A, A_a, a)
MACRO(2, typename_A, A, A_a, a)
...
MACRO(BOOST_PROTO_MAX_ARITY, typename_A, A, A_a, a)
```

Example:

See [BOOST_PROTO_REPEAT_FROM_TO\(\)](#).

Macro BOOST_PROTO_REPEAT_FROM_TO_EX

BOOST_PROTO_REPEAT_FROM_TO_EX — Repeatedly invoke the specified macro.

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_REPEAT_FROM_TO_EX(FROM, TO, MACRO, typename_A, A, A_a, a)
```

Description

BOOST_PROTO_REPEAT_FROM_TO_EX() is used to generate the kind of repetitive code that is typical of DSELs built with Proto.

BOOST_PROTO_REPEAT_FROM_TO_EX(*FROM*, *TO*, *MACRO*, *typename_A*, *A*, *A_a*, *a*) is equivalent to:

```
MACRO(FROM, typename_A, A, A_a, a)
MACRO(FROM+1, typename_A, A, A_a, a)
...
MACRO(TO-1, typename_A, A, A_a, a)
```

Example:

See [BOOST_PROTO_REPEAT_FROM_TO\(\)](#).

Macro BOOST_PROTO_LOCAL_ITERATE

BOOST_PROTO_LOCAL_ITERATE — Vertical repetition of a user-supplied macro.

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_LOCAL_ITERATE()
```

Description

BOOST_PROTO_LOCAL_ITERATE() is used generate the kind of repetitive code that is typical of DSELS built with Proto. This macro causes the user-defined macro BOOST_PROTO_LOCAL_MACRO() to be expanded with values in the range specified by BOOST_PROTO_LOCAL_LIMITS.

Usage:

```
#include BOOST_PROTO_LOCAL_ITERATE()
```

Example:

```
// Generate BOOST_PROTO_MAX_ARITY-1 overloads of the
// following construct() function template.
#define BOOST_PROTO_LOCAL_MACRO(N, typename_A, A_const_ref, A_const_ref_a, ref_a)\
template<typename T, typename_A(N)>                                     \
typename proto::result_of::make_expr<                                \
    proto::tag::function                                             \
    , construct_helper<T>                                            \
    , A_const_ref(N)                                                 \
>::type const                                                         \
construct(A_const_ref_a(N))                                           \
{\                                                                      \
    return proto::make_expr<                                         \
        proto::tag::function                                         \
    >(\                                                                    \
        construct_helper<T>()                                         \
        , ref_a(N)                                                   \
    );                                                                \
}\
#define BOOST_PROTO_LOCAL_LIMITS (1, BOOST_PP_DEC(BOOST_PROTO_MAX_ARITY))
#include BOOST_PROTO_LOCAL_ITERATE()
```

The above inclusion of BOOST_PROTO_LOCAL_ITERATE() will generate the following code:


```
template<typename T, typename A0>
typename proto::result_of::make_expr<
    proto::tag::function
    , construct_helper<T>
    , A0 const &
>::type const
construct(A0 const & a0)
{
    return proto::make_expr<
        proto::tag::function
    >(
        construct_helper<T>()
        , boost::ref(a0)
    );
}

template<typename T, typename A0, typename A1>
typename proto::result_of::make_expr<
    proto::tag::function
    , construct_helper<T>
    , A0 const &
    , A1 const &
>::type const
construct(A0 const & a0, A1 const & a1)
{
    return proto::make_expr<
        proto::tag::function
    >(
        construct_helper<T>()
        , boost::ref(a0)
        , boost::ref(a1)
    );
}
```

If `BOOST_PROTO_LOCAL_LIMITS` is not defined by the user, it defaults to `(1, BOOST_PROTO_MAX_ARITY)`.

At each iteration, `BOOST_PROTO_LOCAL_MACRO()` is invoked with the current iteration number and the following 4 macro parameters:

- `BOOST_PROTO_LOCAL_typename_A`
- `BOOST_PROTO_LOCAL_A`
- `BOOST_PROTO_LOCAL_A_a`
- `BOOST_PROTO_LOCAL_a`

If these macros are not defined by the user, they default respectively to:

- `BOOST_PROTO_typename_A`
- `BOOST_PROTO_A_const_ref`
- `BOOST_PROTO_A_const_ref_a`
- `BOOST_PROTO_ref_a`

After including `BOOST_PROTO_LOCAL_ITERATE()`, the following macros are automatically undefined:

- `BOOST_PROTO_LOCAL_MACRO`

- BOOST_PROTO_LOCAL_LIMITS
- BOOST_PROTO_LOCAL_typename_A
- BOOST_PROTO_LOCAL_A
- BOOST_PROTO_LOCAL_A_a
- BOOST_PROTO_LOCAL_a

Macro `BOOST_PROTO_typename_A`

`BOOST_PROTO_typename_A` — Generates sequences like `typename A0, typename A1, ... typename AN-1` .

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_typename_A(N)
```

Description

Intended for use with the `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_LOCAL_ITERATE()` macros.

`BOOST_PROTO_typename_A(N)` generates sequences like:

```
typename A0, typename A1, ... typename AN-1
```


Macro `BOOST_PROTO_A_const_ref`

`BOOST_PROTO_A_const_ref` — Generates sequences like `A0 const &, A1 const &, ... AN-1 const & .`

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_A_const_ref(N)
```

Description

Intended for use with the `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_LOCAL_ITERATE()` macros.

`BOOST_PROTO_A_const_ref(N)` generates sequences like:

```
A0 const &, A1 const &, ... AN-1 const &
```


Macro `BOOST_PROTO_A_ref`

`BOOST_PROTO_A_ref` — Generates sequences like $A_0 \ \&, A_1 \ \&, \dots A_{N-1} \ \& .$

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_A_ref(N)
```

Description

Intended for use with the `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_LOCAL_ITERATE()` macros.

`BOOST_PROTO_A_ref(N)` generates sequences like:

```
 $A_0 \ \&, A_1 \ \&, \dots A_{N-1} \ \&$ 
```


Macro BOOST_PROTO_A

BOOST_PROTO_A — Generates sequences like A_0, A_1, \dots, A_{N-1} .

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_A(N)
```

Description

Intended for use with the [BOOST_PROTO_REPEAT\(\)](#) and [BOOST_PROTO_LOCAL_ITERATE\(\)](#) macros.

BOOST_PROTO_A(N) generates sequences like:

```
A0, A1, ... AN-1
```


Macro `BOOST_PROTO_A_const`

`BOOST_PROTO_A_const` — Generates sequences like A_0 const, A_1 const, ... A_{N-1} const .

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_A_const(N)
```

Description

Intended for use with the `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_LOCAL_ITERATE()` macros.

`BOOST_PROTO_A_const(N)` generates sequences like:

```
 $A_0$  const,  $A_1$  const, ...  $A_{N-1}$  const
```


Macro `BOOST_PROTO_A_const_ref_a`

`BOOST_PROTO_A_const_ref_a` — Generates sequences like A_0 const & a_0 , A_1 const & a_1 , ... A_{N-1} const & a_{N-1} .

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_A_const_ref_a(N)
```

Description

Intended for use with the `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_LOCAL_ITERATE()` macros.

`BOOST_PROTO_A_const_ref_a(N)` generates sequences like:

```
 $A_0$  const &  $a_0$ ,  $A_1$  const &  $a_1$ , ...  $A_{N-1}$  const &  $a_{N-1}$ 
```


Macro BOOST_PROTO_A_ref_a

BOOST_PROTO_A_ref_a — Generates sequences like $A_0 \ \& \ a_0, A_1 \ \& \ a_1, \dots A_{N-1} \ \& \ a_{N-1}$.

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_A_ref_a(N)
```

Description

Intended for use with the [BOOST_PROTO_REPEAT\(\)](#) and [BOOST_PROTO_LOCAL_ITERATE\(\)](#) macros.

BOOST_PROTO_A_ref_a(N) generates sequences like:

```
 $A_0 \ \& \ a_0, A_1 \ \& \ a_1, \dots A_{N-1} \ \& \ a_{N-1}$ 
```


Macro BOOST_PROTO_ref_a

BOOST_PROTO_ref_a — Generates sequences like `boost::ref(a0)`, `boost::ref(a1)`, ... `boost::ref(aN-1)` .

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_ref_a(N)
```

Description

Intended for use with the [BOOST_PROTO_REPEAT\(\)](#) and [BOOST_PROTO_LOCAL_ITERATE\(\)](#) macros.

BOOST_PROTO_ref_a(N) generates sequences like:

```
boost::ref(a0), boost::ref(a1), ... boost::ref(aN-1)
```


Macro `BOOST_PROTO_a`

`BOOST_PROTO_a` — Generates sequences like a_0, a_1, \dots, a_{N-1} .

Synopsis

```
// In header: <boost/proto/repeat.hpp>

BOOST_PROTO_a(N)
```

Description

Intended for use with the `BOOST_PROTO_REPEAT()` and `BOOST_PROTO_LOCAL_ITERATE()` macros.

`BOOST_PROTO_a(N)` generates sequences like:

```
 $a_0, a_1, \dots, a_{N-1}$ 
```

Header `<boost/proto/tags.hpp>`

Contains the tags for all the overloadable operators in C++


```
namespace boost {  
  namespace proto {  
    namespace tag {  
      struct terminal;  
      struct unary_plus;  
      struct negate;  
      struct dereference;  
      struct complement;  
      struct address_of;  
      struct logical_not;  
      struct pre_inc;  
      struct pre_dec;  
      struct post_inc;  
      struct post_dec;  
      struct shift_left;  
      struct shift_right;  
      struct multiplies;  
      struct divides;  
      struct modulus;  
      struct plus;  
      struct minus;  
      struct less;  
      struct greater;  
      struct less_equal;  
      struct greater_equal;  
      struct equal_to;  
      struct not_equal_to;  
      struct logical_or;  
      struct logical_and;  
      struct bitwise_and;  
      struct bitwise_or;  
      struct bitwise_xor;  
      struct comma;  
      struct mem_ptr;  
      struct assign;  
      struct shift_left_assign;  
      struct shift_right_assign;  
      struct multiplies_assign;  
      struct divides_assign;  
      struct modulus_assign;  
      struct plus_assign;  
      struct minus_assign;  
      struct bitwise_and_assign;  
      struct bitwise_or_assign;  
      struct bitwise_xor_assign;  
      struct subscript;  
      struct if_else_;  
      struct function;  
    }  
  }  
}
```


Struct terminal

boost::proto::tag::terminal — Tag type for terminals; aka, leaves in the expression tree.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct terminal {
};
```


Struct unary_plus

boost::proto::tag::unary_plus — Tag type for the unary + operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct unary_plus {
};
```


Struct negate

boost::proto::tag::negate — Tag type for the unary - operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct negate {
};
```


Struct dereference

boost::proto::tag::dereference — Tag type for the unary * operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct dereference {
};
```


Struct complement

boost::proto::tag::complement — Tag type for the unary ~ operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct complement {
};
```


Struct address_of

boost::proto::tag::address_of — Tag type for the unary & operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct address_of {
};
```


Struct logical_not

boost::proto::tag::logical_not — Tag type for the unary ! operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct logical_not {
};
```


Struct `pre_inc`

`boost::proto::tag::pre_inc` — Tag type for the unary prefix `++` operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct pre_inc {
};
```


Struct pre_dec

boost::proto::tag::pre_dec — Tag type for the unary prefix -- operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct pre_dec {
};
```


Struct `post_inc`

`boost::proto::tag::post_inc` — Tag type for the unary postfix `++` operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct post_inc {
};
```


Struct `post_dec`

`boost::proto::tag::post_dec` — Tag type for the unary postfix `--` operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct post_dec {
};
```


Struct shift_left

boost::proto::tag::shift_left — Tag type for the binary << operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct shift_left {
};
```


Struct shift_right

boost::proto::tag::shift_right — Tag type for the binary >> operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct shift_right {
};
```


Struct multiplies

boost::proto::tag::multiplies — Tag type for the binary * operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct multiplies {
};
```


Struct divides

boost::proto::tag::divides — Tag type for the binary / operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct divides {
};
```


Struct modulus

boost::proto::tag::modulus — Tag type for the binary % operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct modulus {
};
```


Struct plus

boost::proto::tag::plus — Tag type for the binary + operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct plus {
};
```


Struct minus

boost::proto::tag::minus — Tag type for the binary - operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct minus {
};
```


Struct less

boost::proto::tag::less — Tag type for the binary < operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct less {
};
```


Struct greater

boost::proto::tag::greater — Tag type for the binary > operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct greater {
};
```


Struct `less_equal`

`boost::proto::tag::less_equal` — Tag type for the binary `<=` operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct less_equal {
};
```


Struct greater_equal

boost::proto::tag::greater_equal — Tag type for the binary >= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct greater_equal {
};
```


Struct equal_to

boost::proto::tag::equal_to — Tag type for the binary == operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct equal_to {
};
```


Struct not_equal_to

boost::proto::tag::not_equal_to — Tag type for the binary != operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct not_equal_to {
};
```


Struct logical_or

boost::proto::tag::logical_or — Tag type for the binary || operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct logical_or {
};
```


Struct logical_and

boost::proto::tag::logical_and — Tag type for the binary && operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct logical_and {
};
```


Struct bitwise_and

boost::proto::tag::bitwise_and — Tag type for the binary & operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct bitwise_and {
};
```


Struct bitwise_or

boost::proto::tag::bitwise_or — Tag type for the binary | operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct bitwise_or {
};
```


Struct bitwise_xor

boost::proto::tag::bitwise_xor — Tag type for the binary ^ operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct bitwise_xor {
};
```


Struct comma

boost::proto::tag::comma — Tag type for the binary , operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct comma {
};
```


Struct mem_ptr

boost::proto::tag::mem_ptr — Tag type for the binary ->* operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct mem_ptr {
};
```


Struct assign

boost::proto::tag::assign — Tag type for the binary = operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct assign {
};
```


Struct shift_left_assign

boost::proto::tag::shift_left_assign — Tag type for the binary <<= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct shift_left_assign {
};
```


Struct shift_right_assign

boost::proto::tag::shift_right_assign — Tag type for the binary >>= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct shift_right_assign {
};
```


Struct multiplies_assign

boost::proto::tag::multiplies_assign — Tag type for the binary *= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct multiplies_assign {
};
```


Struct divides_assign

boost::proto::tag::divides_assign — Tag type for the binary /= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct divides_assign {
};
```


Struct modulus_assign

boost::proto::tag::modulus_assign — Tag type for the binary = operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct modulus_assign {
};
```


Struct plus_assign

boost::proto::tag::plus_assign — Tag type for the binary += operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct plus_assign {
};
```


Struct minus_assign

boost::proto::tag::minus_assign — Tag type for the binary -= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct minus_assign {
};
```


Struct bitwise_and_assign

boost::proto::tag::bitwise_and_assign — Tag type for the binary &= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct bitwise_and_assign {
};
```


Struct bitwise_or_assign

boost::proto::tag::bitwise_or_assign — Tag type for the binary |= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct bitwise_or_assign {
};
```


Struct bitwise_xor_assign

boost::proto::tag::bitwise_xor_assign — Tag type for the binary ^= operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct bitwise_xor_assign {
};
```


Struct subscript

boost::proto::tag::subscript — Tag type for the binary subscript operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct subscript {
};
```


Struct if_else_

boost::proto::tag::if_else_ — Tag type for the ternary ?: conditional operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct if_else_ {
};
```


Struct function

`boost::proto::tag::function` — Tag type for the n-ary function call operator.

Synopsis

```
// In header: <boost/proto/tags.hpp>

struct function {
};
```

Header `<boost/proto/traits.hpp>`

Contains definitions for various expression traits and utilities like `proto::tag_of<>` and `proto::arity_of<>`; the functions `proto::value()`, `proto::left()` and `proto::right()`; `proto::child()`, `proto::child_c()`, `proto::as_expr()`, `proto::as_child()`, and assorted helpers.

```
namespace boost {
namespace proto {
    template<typename T> struct is_callable;
    template<typename T> struct is_aggregate;

    template<typename T> struct terminal;
    template<typename T, typename U, typename V> struct if_else_;
    template<typename T> struct unary_plus;
    template<typename T> struct negate;
    template<typename T> struct dereference;
    template<typename T> struct complement;
    template<typename T> struct address_of;
    template<typename T> struct logical_not;
    template<typename T> struct pre_inc;
    template<typename T> struct pre_dec;
    template<typename T> struct post_inc;
    template<typename T> struct post_dec;
    template<typename T, typename U> struct shift_left;
    template<typename T, typename U> struct shift_right;
    template<typename T, typename U> struct multiplies;
    template<typename T, typename U> struct divides;
    template<typename T, typename U> struct modulus;
    template<typename T, typename U> struct plus;
    template<typename T, typename U> struct minus;
    template<typename T, typename U> struct less;
    template<typename T, typename U> struct greater;
    template<typename T, typename U> struct less_equal;
    template<typename T, typename U> struct greater_equal;
    template<typename T, typename U> struct equal_to;
    template<typename T, typename U> struct not_equal_to;
    template<typename T, typename U> struct logical_or;
    template<typename T, typename U> struct logical_and;
    template<typename T, typename U> struct bitwise_and;
    template<typename T, typename U> struct bitwise_or;
    template<typename T, typename U> struct bitwise_xor;
    template<typename T, typename U> struct comma;
    template<typename T, typename U> struct mem_ptr;
    template<typename T, typename U> struct assign;
    template<typename T, typename U> struct shift_left_assign;
    template<typename T, typename U> struct shift_right_assign;
    template<typename T, typename U> struct multiplies_assign;
    template<typename T, typename U> struct divides_assign;
```



```

template<typename T, typename U> struct modulus_assign;
template<typename T, typename U> struct plus_assign;
template<typename T, typename U> struct minus_assign;
template<typename T, typename U> struct bitwise_and_assign;
template<typename T, typename U> struct bitwise_or_assign;
template<typename T, typename U> struct bitwise_xor_assign;
template<typename T, typename U> struct subscript;
template<typename... A> struct function;
template<typename Tag, typename T> struct nullary_expr;
template<typename Tag, typename T> struct unary_expr;
template<typename Tag, typename T, typename U> struct binary_expr;
template<typename Tag, typename... A> struct nary_expr;
template<typename T> struct is_expr;
template<typename Expr> struct tag_of;
template<typename Expr> struct arity_of;
template<typename T>
    typename proto::result_of::as_expr< T >::type as_expr(T &);
template<typename T>
    typename proto::result_of::as_expr< T const >::type as_expr(T const &);
template<typename Domain, typename T>
    typename proto::result_of::as_expr< T, Domain >::type as_expr(T &);
template<typename Domain, typename T>
    typename proto::result_of::as_expr< T const, Domain >::type
        as_expr(T const &);
template<typename T>
    typename proto::result_of::as_child< T >::type as_child(T &);
template<typename T>
    typename proto::result_of::as_child< T const >::type as_child(T const &);
template<typename Domain, typename T>
    typename proto::result_of::as_child< T, Domain >::type as_child(T &);
template<typename Domain, typename T>
    typename proto::result_of::as_child< T const, Domain >::type
        as_child(T const &);
template<typename N, typename Expr>
    typename proto::result_of::child< Expr &, N >::type child(Expr &);
template<typename N, typename Expr>
    typename proto::result_of::child< Expr const &, N >::type
        child(Expr const &);
template<typename Expr>
    typename proto::result_of::child< Expr & >::type child(Expr &);
template<typename Expr>
    typename proto::result_of::child< Expr const & >::type
        child(Expr const &);
template<long N, typename Expr>
    typename proto::result_of::child_c< Expr &, N >::type child_c(Expr &);
template<long N, typename Expr>
    typename proto::result_of::child_c< Expr const &, N >::type
        child_c(Expr const &);
template<typename Expr>
    typename proto::result_of::value< Expr & >::type value(Expr &);
template<typename Expr>
    typename proto::result_of::value< Expr const & >::type
        value(Expr const &);
template<typename Expr>
    typename proto::result_of::left< Expr & >::type left(Expr &);
template<typename Expr>
    typename proto::result_of::left< Expr const & >::type left(Expr const &);
template<typename Expr>
    typename proto::result_of::right< Expr & >::type right(Expr &);
template<typename Expr>
    typename proto::result_of::right< Expr const & >::type
        right(Expr const &);
namespace functional {

```



```
template<typename Domain = proto::default_domain> struct as_expr;
template<typename Domain = proto::default_domain> struct as_child;
template<long N> struct child_c;
template<typename N = mpl::long_<0> > struct child;
struct value;
struct left;
struct right;
}
namespace result_of {
    template<typename T, typename Domain = proto::default_domain>
        struct as_expr;
    template<typename T, typename Domain = proto::default_domain>
        struct as_child;
    template<typename Expr, typename N = mpl::long_<0> > struct child;
    template<typename Expr> struct value;
    template<typename Expr> struct left;
    template<typename Expr> struct right;
    template<typename Expr, long N> struct child_c;
}
}
```


Struct template `as_expr`

`boost::proto::functional::as_expr` — A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::as_expr()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Domain = proto::default_domain>
struct as_expr : proto::callable {
    // member classes/structs/unions
    template<typename This, typename T>
    struct result<This(T)> : proto::result_of::as_expr< typename remove_reference< T >::type, Domain >
    {
        // public member functions
    };

    template<typename T>
    typename proto::result_of::as_expr< T, Domain >::type
    operator()(T &) const;

    template<typename T>
    typename proto::result_of::as_expr< T const, Domain >::type
    operator()(T const &) const;
};
```

Description

`as_expr` public member functions

1.

```
template<typename T>
    typename proto::result_of::as_expr< T, Domain >::type
    operator()(T & t) const;
```

Wrap an object in a Proto terminal if it isn't a Proto expression already.

Parameters: `t` The object to wrap.

Returns: `proto::as_expr<Domain>(t)`

2.

```
template<typename T>
    typename proto::result_of::as_expr< T const, Domain >::type
    operator()(T const & t) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(T)>

boost::proto::functional::as_expr::result<This(T)>

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename This, typename T>
struct result<This(T)> :
    proto::result_of::as_expr< typename remove_reference< T >::type, Domain >
{
};
```


Struct template `as_child`

`boost::proto::functional::as_child` — A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::as_child()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Domain = proto::default_domain>
struct as_child : proto::callable {
    // member classes/structs/unions
    template<typename This, typename T>
    struct result_of::as_child< typename remove_reference< T >::type, Domain > {
        // ...
    };

    // public member functions
    template<typename T>
    typename proto::result_of::as_child< T, Domain >::type
    operator()(T &) const;
    template<typename T>
    typename proto::result_of::as_child< T const, Domain >::type
    operator()(T const &) const;
};
```

Description

`as_child` public member functions

1.

```
template<typename T>
    typename proto::result_of::as_child< T, Domain >::type
    operator()(T & t) const;
```

Wrap an object in a Proto terminal if it isn't a Proto expression already.

Parameters: `t` The object to wrap.

Returns: `proto::as_child<Domain>(t)`

2.

```
template<typename T>
    typename proto::result_of::as_child< T const, Domain >::type
    operator()(T const & t) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(T)>

boost::proto::functional::as_child::result<This(T)>

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename This, typename T>
struct result<This(T)> :
    proto::result_of::as_child< typename remove_reference< T >::type, Domain >
{
};
```


Struct template `child_c`

`boost::proto::functional::child_c` — A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::child_c()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<long N>
struct child_c : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : proto::result_of::child_c< Expr, N > {
    };

    // public member functions
    template<typename Expr>
    typename proto::result_of::child_c< Expr &, N >::type
    operator()(Expr &) const;
    template<typename Expr>
    typename proto::result_of::child_c< Expr const &, N >::type
    operator()(Expr const &) const;
};
```

Description

`child_c` public member functions

1.

```
template<typename Expr>
    typename proto::result_of::child_c< Expr &, N >::type
    operator()(Expr & expr) const;
```

Return the N^{th} child of the given expression.

Parameters: `expr` The expression node.

Requires: `proto::is_expr<Expr>::value` is true

`N < Expr::proto_arity::value`

Returns: `proto::child_c<N>(expr)`

Throws: Will not throw.

2.

```
template<typename Expr>
    typename proto::result_of::child_c< Expr const &, N >::type
    operator()(Expr const & expr) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(Expr)>

boost::proto::functional::child_c::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : proto::result_of::child_c< Expr, N > {
};
```


Struct template child

`boost::proto::functional::child` — A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::child()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename N = mpl::long_<0> >
struct child : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result_of<This(Expr)> : proto::result_of::child< Expr, N > {
    };

    // public member functions
    template<typename Expr>
        typename proto::result_of::child< Expr &, N >::type
        operator()(Expr &) const;
    template<typename Expr>
        typename proto::result_of::child< Expr const &, N >::type
        operator()(Expr const &) const;
};
```

Description

A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::child()` function. `N` is required to be an MPL Integral Constant.

child public member functions

```
1. template<typename Expr>
    typename proto::result_of::child< Expr &, N >::type
    operator()(Expr & expr) const;
```

Return the N^{th} child of the given expression.

Parameters: `expr` The expression node.

Requires: `proto::is_expr<Expr>::value` is true

`N::value < Expr::proto_arity::value`

Returns: `proto::child<N>(expr)`

Throws: Will not throw.

```
2. template<typename Expr>
    typename proto::result_of::child< Expr const &, N >::type
    operator()(Expr const & expr) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(Expr)>

boost::proto::functional::child::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : proto::result_of::child< Expr, N > {
};
```


Struct value

`boost::proto::functional::value` — A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::value()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

struct value : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : proto::result_of::value< Expr > {
    };

    // public member functions
    template<typename Expr>
    typename proto::result_of::value< Expr & >::type operator()(Expr &) const;
    template<typename Expr>
    typename proto::result_of::value< Expr const & >::type
    operator()(Expr const &) const;
};
```

Description

`value` public member functions

1.

```
template<typename Expr>
    typename proto::result_of::value< Expr & >::type
    operator()(Expr & expr) const;
```

Return the value of the given terminal expression.

Parameters: `expr` The terminal expression node.

Requires: `proto::is_expr<Expr>::value` is true

`0 == Expr::proto_arity::value`

Returns: `proto::value(expr)`

Throws: Will not throw.

2.

```
template<typename Expr>
    typename proto::result_of::value< Expr const & >::type
    operator()(Expr const & expr) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(Expr)>

boost::proto::functional::value::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : proto::result_of::value< Expr > {
};
```


Struct left

`boost::proto::functional::left` — A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::left()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

struct left : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : proto::result_of::left< Expr > {
    };

    // public member functions
    template<typename Expr>
    typename proto::result_of::left< Expr & >::type operator()(Expr &) const;
    template<typename Expr>
    typename proto::result_of::left< Expr const & >::type
    operator()(Expr const &) const;
};
```

Description

left public member functions

1.

```
template<typename Expr>
    typename proto::result_of::left< Expr & >::type
    operator()(Expr & expr) const;
```

Return the left child of the given binary expression.

Parameters: `expr` The expression node.

Requires: `proto::is_expr<Expr>::value` is true

`2 == Expr::proto_arity::value`

Returns: `proto::left(expr)`

Throws: Will not throw.

2.

```
template<typename Expr>
    typename proto::result_of::left< Expr const & >::type
    operator()(Expr const & expr) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(Expr)>

boost::proto::functional::left::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : proto::result_of::left< Expr > {
};
```


Struct right

`boost::proto::functional::right` — A callable [PolymorphicFunctionObject](#) that is equivalent to the `proto::right()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

struct right : proto::callable {
    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> : proto::result_of::right< Expr > {
    };

    // public member functions
    template<typename Expr>
    typename proto::result_of::right< Expr & >::type operator()(Expr &) const;
    template<typename Expr>
    typename proto::result_of::right< Expr const & >::type
    operator()(Expr const &) const;
};
```

Description

`right` public member functions

1.

```
template<typename Expr>
    typename proto::result_of::right< Expr & >::type
    operator()(Expr & expr) const;
```

Return the right child of the given binary expression.

Parameters: `expr` The expression node.

Requires: `proto::is_expr<Expr>::value` is true

`2 == Expr::proto_arity::value`

Returns: `proto::right(expr)`

Throws: Will not throw.

2.

```
template<typename Expr>
    typename proto::result_of::right< Expr const & >::type
    operator()(Expr const & expr) const;
```


Struct template result<This(Expr)>

boost::proto::functional::right::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> : proto::result_of::right< Expr > {
};
```


Struct template `as_expr`

`boost::proto::result_of::as_expr` — A metafunction that computes the return type of the `proto::as_expr()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename Domain = proto::default_domain>
struct as_expr {
    // types
    typedef typename Domain::template as_expr< T >::result_type type;
};
```

Description

The `proto::result_of::as_expr<>` metafunction turns types into Proto expression types, if they are not already, in a domain-specific way. It is intended for use to compute the type of a local variable that can hold the result of the `proto::as_expr()` function.

See `proto::domain::as_expr<>` for a complete description of the default behavior.

Struct template `as_child`

`boost::proto::result_of::as_child` — A metafunction that computes the return type of the `proto::as_child()` function.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename Domain = proto::default_domain>
struct as_child {
    // types
    typedef typename Domain::template as_child< T >::result_type type;
};
```

Description

The `proto::result_of::as_child<>` metafunction turns types into Proto expression types, if they are not already, in a domain-specific way. It is used by Proto to compute the type of an object to store as a child in another expression node.

See `proto::domain::as_child<>` for a complete description of the default behavior.

Struct template child

`boost::proto::result_of::child` — A metafunction that returns the type of the N^{th} child of a Proto expression, where N is an MPL Integral Constant.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename N = mpl::long_<0> >
struct child : proto::result_of::child_c<Expr, N::value> {
};
```

Description

`proto::result_of::child<Expr, N>` is equivalent to `proto::result_of::child_c<Expr, N::value>`.

Struct template value

`boost::proto::result_of::value` — A metafunction that returns the type of the value of a terminal Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
struct value {
    // types
    typedef typename Expr::proto_child0 value_type;
    typedef see-below type;
};
```

Description

`value` public types

1. `typedef typename Expr::proto_child0 value_type;`

The raw type of the value as it is stored within `Expr`. This may be a value or a reference.

2. `typedef see-below type;`

If `Expr` is not a reference type, `type` is computed as follows:

- `T const(&)[N]` becomes `T[N]`
- `T[N]` becomes `T[N]`
- `T(&)[N]` becomes `T[N]`
- `R(&)(A...)` becomes `R(&)(A...)`
- `T const &` becomes `T`
- `T &` becomes `T`
- `T` becomes `T`

If `Expr` is a non-const reference type, `type` is computed as follows:

- `T const(&)[N]` becomes `T const(&)[N]`
- `T[N]` becomes `T(&)[N]`
- `T(&)[N]` becomes `T(&)[N]`
- `R(&)(A...)` becomes `R(&)(A...)`
- `T const &` becomes `T const &`
- `T &` becomes `T &`
- `T` becomes `T &`

If `Expr` is a const reference type, `type` is computed as follows:

- `T const(&)[N]` becomes `T const(&)[N]`

- `T[N]` becomes `T const(&)[N]`
- `T(&)[N]` becomes `T(&)[N]`
- `R(&)(A...)` becomes `R(&)(A...)`
- `T const &` becomes `T const &`
- `T &` becomes `T &`
- `T` becomes `T const &`

Struct template left

`boost::proto::result_of::left` — A metafunction that returns the type of the left child of a binary Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
struct left : proto::result_of::child_c< Expr, 0 > {
};
```

Description

`proto::result_of::left<Expr>` is equivalent to `proto::result_of::child_c<Expr, 0>`.

Struct template right

`boost::proto::result_of::right` — A metafunction that returns the type of the right child of a binary Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
struct right : proto::result_of::child_c< Expr, 1 > {
};
```

Description

`proto::result_of::right<Expr>` is equivalent to `proto::result_of::child_c<Expr, 1>`.

Struct template `child_c`

`boost::proto::result_of::child_c` — A metafunction that returns the type of the N^{th} child of a Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, long N>
struct child_c {
    // types
    typedef typename Expr::proto_child0 value_type;
    typedef see-below type;
};
```

Description

A metafunction that returns the type of the N^{th} child of a Proto expression. N must be 0 or less than `Expr::proto_arity::value`.

`child_c` public types

1. `typedef typename Expr::proto_child0 value_type;`

The raw type of the N^{th} child as it is stored within `Expr`. This may be a value or a reference.

2. `typedef see-below type;`

If `Expr` is not a reference type, `type` is computed as follows:

- `T const &` becomes `T`
- `T &` becomes `T`
- `T` becomes `T`

If `Expr` is a non-const reference type, `type` is computed as follows:

- `T const &` becomes `T const &`
- `T &` becomes `T &`
- `T` becomes `T &`

If `Expr` is a const reference type, `type` is computed as follows:

- `T const &` becomes `T const &`
- `T &` becomes `T &`
- `T` becomes `T const &`

Struct template `is_callable`

`boost::proto::is_callable` — Boolean metafunction which tells whether a type is a callable [PolymorphicFunctionObject](#) or not.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct is_callable : mpl::bool_<true-or-false> {
};
```

Description

`proto::is_callable<>` is used by the `proto::when<>` transform to determine whether a function type $R(A_1, \dots, A_n)$ is a [CallableTransform](#) or an [ObjectTransform](#). The former are evaluated using `proto::call<>` and the later with `proto::make<>`. If `proto::is_callable<R>::value` is true, the function type is a [CallableTransform](#); otherwise, it is an [ObjectTransform](#).

Unless specialized for a type `T`, `proto::is_callable<T>::value` is computed as follows:

- If `T` is a template type $X<Y_0, \dots, Y_n>$, where all Y_x are types for x in $[0, n]$, `proto::is_callable<T>::value` is `boost::is_same<Y_n, proto::callable>::value`.
- If `T` is derived from `proto::callable`, `proto::is_callable<T>::value` is true.
- Otherwise, `proto::is_callable<T>::value` is false.

Struct template `is_aggregate`

`boost::proto::is_aggregate` — A Boolean metafunction that indicates whether a type requires aggregate initialization.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct is_aggregate : mpl::bool_<true-or-false> {
};
```

Description

`proto::is_aggregate<>` is used by the `proto::make<>` transform to determine how to construct an object of some type `T`, given some initialization arguments a_0, \dots, a_n . If `proto::is_aggregate<T>::value` is `true`, then an object of type `T` will be initialized as `T t = {a0, ..., an};`. Otherwise, it will be initialized as `T t(a0, ..., an).`

Note: `proto::expr<>` and `proto::basic_expr<>` are aggregates.

Struct template terminal

`boost::proto::terminal` — A metafunction for generating terminal expression types, a grammar element for matching terminal expressions, and a [PrimitiveTransform](#) that returns the current expression unchanged.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct terminal : proto::transform< terminal<T> > {
    // types
    typedef proto::expr< proto::tag::terminal, proto::term< T > >      type;
    typedef proto::basic_expr< proto::tag::terminal, proto::term< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef Expr result_type;

        // public member functions
        Expr operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

Struct template impl

boost::proto::terminal::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef Expr result_type;

    // public member functions
    Expr operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1. Expr operator()(typename impl::expr_param expr, typename impl::state_param,
 typename impl::data_param) const;

Parameters: expr The current expression
Requires: proto::matches<Expr, proto::terminal<T> >::value is true.
Returns: expr
Throws: Will not throw.

Struct template `if_else_`

`boost::proto::if_else_` — A metafunction for generating ternary conditional expression types, a grammar element for matching ternary conditional expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U, typename V>
struct if_else_ : proto::transform< if_else_<T, U, V> > {
    // types
    typedef proto::expr< proto::tag::if_else_, proto::list3< T, U, V > >      type;
    typedef proto::basic_expr< proto::tag::if_else_, proto::list3< T, U, V > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<if_else_>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::if_else_::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<if_else_::template impl<Expr, State, Data>
    {
    };
```


Struct template unary_plus

boost::proto::unary_plus — A metafunction for generating unary plus expression types, a grammar element for matching unary plus expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct unary_plus : proto::transform< unary_plus<T> > {
    // types
    typedef proto::expr< proto::tag::unary_plus, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::unary_plus, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<unary_plus>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::unary_plus::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<unary_plus>::template impl<Expr, State, Data>
{
};
```


Struct template negate

boost::proto::negate — A metafunction for generating unary minus expression types, a grammar element for matching unary minus expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct negate : proto::transform< negate<T> > {
    // types
    typedef proto::expr< proto::tag::negate, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::negate, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<negate>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::negate::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<negate>::template impl<Expr, State, Data> {
};
```


Struct template dereference

`boost::proto::dereference` — A metafunction for generating dereference expression types, a grammar element for matching dereference expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct dereference : proto::transform< dereference<T> > {
    // types
    typedef proto::expr< proto::tag::dereference, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::dereference, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<dereference>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::dereference::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<dereference>::template impl<Expr, State, Data>
{
};
```


Struct template complement

`boost::proto::complement` — A metafunction for generating complement expression types, a grammar element for matching complement expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct complement : proto::transform< complement<T> > {
    // types
    typedef proto::expr< proto::tag::complement, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::complement, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<complement>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::complement::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<complement>::template impl<Expr, State, Data>
{
};
```


Struct template `address_of`

`boost::proto::address_of` — A metafunction for generating `address_of` expression types, a grammar element for matching `address_of` expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct address_of : proto::transform< address_of<T> > {
    // types
    typedef proto::expr< proto::tag::address_of, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::address_of, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<address_of>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::address_of::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<address_of>::template impl<Expr, State, Data>
{
};
```


Struct template `logical_not`

`boost::proto::logical_not` — A metafunction for generating `logical_not` expression types, a grammar element for matching `logical_not` expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct logical_not : proto::transform< logical_not<T> > {
    // types
    typedef proto::expr< proto::tag::logical_not, proto::list1< T > >      type;
    typedef proto::basic_expr< proto::tag::logical_not, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<logical_not>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::logical_not::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<logical_not>::template impl<Expr, State, Data>
{
};
```


Struct template `pre_inc`

`boost::proto::pre_inc` — A metafunction for generating pre-increment expression types, a grammar element for matching pre-increment expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct pre_inc : proto::transform< pre_inc<T> > {
    // types
    typedef proto::expr< proto::tag::pre_inc, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::pre_inc, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<pre_inc>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::pre_inc::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<pre_inc>::template impl<Expr, State, Data> {
};
```


Struct template `pre_dec`

`boost::proto::pre_dec` — A metafunction for generating pre-decrement expression types, a grammar element for matching pre-decrement expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct pre_dec : proto::transform< pre_dec<T> > {
    // types
    typedef proto::expr< proto::tag::pre_dec, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::pre_dec, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<pre_dec>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::pre_dec::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<pre_dec>::template impl<Expr, State, Data> {
};
```


Struct template `post_inc`

`boost::proto::post_inc` — A metafunction for generating post-increment expression types, a grammar element for matching post-increment expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct post_inc : proto::transform< post_inc<T> > {
    // types
    typedef proto::expr< proto::tag::post_inc, proto::list1< T > > type;
    typedef proto::basic_expr< proto::tag::post_inc, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<post_inc>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::post_inc::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<post_inc>::template impl<Expr, State, Data>
{
};
```


Struct template `post_dec`

`boost::proto::post_dec` — A metafunction for generating post-decrement expression types, a grammar element for matching post-decrement expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct post_dec : proto::transform< post_dec<T> > {
    // types
    typedef proto::expr< proto::tag::post_dec, proto::list1< T > >
type;
    typedef
        proto::basic_expr< proto::tag::post_dec, proto::list1< T > >
        proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<post_dec>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::post_dec::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<post_dec>::template impl<Expr, State, Data>
{
};
```


Struct template `shift_left`

`boost::proto::shift_left` — A metafunction for generating left-shift expression types, a grammar element for matching left-shift expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct shift_left : proto::transform< shift_left<T, U> > {
    // types
    typedef proto::expr< proto::tag::shift_left, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::shift_left, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<shift_left>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::shift_left::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<shift_left>::template impl<Expr, State, Data>
{
};
```


Struct template `shift_right`

`boost::proto::shift_right` — A metafunction for generating right-shift expression types, a grammar element for matching right-shift expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct shift_right : proto::transform< shift_right<T, U> > {
    // types
    typedef proto::expr< proto::tag::shift_right, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::shift_right, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<shift_right>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::shift_right::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<shift_right>::template impl<Expr, State, Data>
{
};
```


Struct template multiplies

boost::proto::multiplies — A metafunction for generating multiplies expression types, a grammar element for matching multiplies expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct multiplies : proto::transform< multiplies<T, U> > {
    // types
    typedef proto::expr< proto::tag::multiplies, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::multiplies, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<multiplies>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::multiplies::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<multiplies>::template impl<Expr, State, Data>
{
};
```


Struct template divides

`boost::proto::divides` — A metafunction for generating divides expression types, a grammar element for matching divides expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct divides : proto::transform< divides<T, U> > {
    // types
    typedef proto::expr< proto::tag::divides, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::divides, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<divides>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::divides::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<divides>::template impl<Expr, State, Data> {
};
```


Struct template modulus

boost::proto::modulus — A metafunction for generating modulus expression types, a grammar element for matching modulus expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct modulus : proto::transform< modulus<T, U> > {
    // types
    typedef proto::expr< proto::tag::modulus, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::modulus, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<modulus>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::modulus::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<modulus>::template impl<Expr, State, Data> {
};
```


Struct template plus

boost::proto::plus — A metafunction for generating binary plus expression types, a grammar element for matching binary plus expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct plus : proto::transform< plus<T, U> > {
    // types
    typedef proto::expr< proto::tag::plus, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::plus, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::pass_through<plus>::template impl<Expr, State, Data> {
    };
};
```

Description

Struct template impl

boost::proto::plus::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<plus>::template impl<Expr, State, Data> {
};
```


Struct template minus

`boost::proto::minus` — A metafunction for generating binary minus expression types, a grammar element for matching binary minus expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct minus : proto::transform< minus<T, U> > {
    // types
    typedef proto::expr< proto::tag::minus, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::minus, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::pass_through<minus>::template impl<Expr, State, Data> {
    };
};
```

Description

Struct template impl

boost::proto::minus::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<minus>::template impl<Expr, State, Data> {
};
```


Struct template less

boost::proto::less — A metafunction for generating less expression types, a grammar element for matching less expressions, and a [PrimitiveTransform](#) that dispatches to the [proto::pass_through<>](#) transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct less : proto::transform< less<T, U> > {
    // types
    typedef proto::expr< proto::tag::less, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::less, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::pass_through<less>::template impl<Expr, State, Data> {
    };
};
```

Description

Struct template impl

boost::proto::less::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<less>::template impl<Expr, State, Data> {
};
```


Struct template greater

`boost::proto::greater` — A metafunction for generating greater expression types, a grammar element for matching greater expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct greater : proto::transform< greater<T, U> > {
    // types
    typedef proto::expr< proto::tag::greater, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::greater, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<greater>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::greater::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<greater>::template impl<Expr, State, Data> {
};
```


Struct template less_equal

boost::proto::less_equal — A metafunction for generating less-or-equal expression types, a grammar element for matching less-or-equal expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct less_equal : proto::transform< less_equal<T, U> > {
    // types
    typedef proto::expr< proto::tag::less_equal, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::less_equal, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<less_equal>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::less_equal::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<less_equal>::template impl<Expr, State, Data>
{
};
```


Struct template greater_equal

boost::proto::greater_equal — A metafunction for generating greater-or-equal expression types, a grammar element for matching greater-or-equal expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct greater_equal : proto::transform< greater_equal<T, U> > {
    // types
    typedef proto::expr< proto::tag::greater_equal, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::greater_equal, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<greater_equal>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::greater_equal::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<greater_equal>::template impl<Expr, State, Data>
{
};
```


Struct template equal_to

boost::proto::equal_to — A metafunction for generating equal-to expression types, a grammar element for matching equal-to expressions, and a [PrimitiveTransform](#) that dispatches to the [proto::pass_through<>](#) transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct equal_to : proto::transform< equal_to<T, U> > {
    // types
    typedef proto::expr< proto::tag::equal_to, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::equal_to, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<equal_to>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::equal_to::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<equal_to>::template impl<Expr, State, Data>
{
};
```


Struct template `not_equal_to`

`boost::proto::not_equal_to` — A metafunction for generating not-equal-to expression types, a grammar element for matching not-equal-to expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct not_equal_to : proto::transform< not_equal_to<T, U> > {
    // types
    typedef proto::expr< proto::tag::not_equal_to, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::not_equal_to, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<not_equal_to>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::not_equal_to::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<not_equal_to>::template impl<Expr, State, Data>
{
};
```


Struct template `logical_or`

`boost::proto::logical_or` — A metafunction for generating logical-or expression types, a grammar element for matching logical-or expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct logical_or : proto::transform< logical_or<T, U> > {
    // types
    typedef proto::expr< proto::tag::logical_or, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::logical_or, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<logical_or>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::logical_or::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<logical_or>::template impl<Expr, State, Data>
{
};
```


Struct template `logical_and`

`boost::proto::logical_and` — A metafunction for generating logical-and expression types, a grammar element for matching logical-and expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct logical_and : proto::transform< logical_and<T, U> > {
    // types
    typedef proto::expr< proto::tag::logical_and, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::logical_and, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<logical_and>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::logical_and::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<logical_and>::template impl<Expr, State, Data>
{
};
```


Struct template `bitwise_and`

`boost::proto::bitwise_and` — A metafunction for generating bitwise-and expression types, a grammar element for matching bitwise-and expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct bitwise_and : proto::transform< bitwise_and<T, U> > {
    // types
    typedef proto::expr< proto::tag::bitwise_and, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::bitwise_and, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<bitwise_and>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::bitwise_and::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<bitwise_and>::template impl<Expr, State, Data>
{
};
```


Struct template bitwise_or

boost::proto::bitwise_or — A metafunction for generating bitwise-or expression types, a grammar element for matching bitwise-or expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct bitwise_or : proto::transform< bitwise_or<T, U> > {
    // types
    typedef proto::expr< proto::tag::bitwise_or, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::bitwise_or, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<bitwise_or>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::bitwise_or::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<bitwise_or>::template impl<Expr, State, Data>
{
};
```


Struct template bitwise_xor

boost::proto::bitwise_xor — A metafunction for generating bitwise-xor expression types, a grammar element for matching bitwise-xor expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct bitwise_xor : proto::transform< bitwise_xor<T, U> > {
    // types
    typedef proto::expr< proto::tag::bitwise_xor, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::bitwise_xor, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<bitwise_xor>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::bitwise_xor::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<bitwise_xor>::template impl<Expr, State, Data>
{
};
```


Struct template comma

boost::proto::comma — A metafunction for generating comma expression types, a grammar element for matching comma expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct comma : proto::transform< comma<T, U> > {
    // types
    typedef proto::expr< proto::tag::comma, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::comma, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::pass_through<comma>::template impl<Expr, State, Data> {
    };
};
```

Description

Struct template impl

boost::proto::comma::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<comma>::template impl<Expr, State, Data> {
};
```


Struct template mem_ptr

boost::proto::mem_ptr

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct mem_ptr : proto::transform< mem_ptr<T, U> > {
    // types
    typedef proto::expr< proto::tag::mem_ptr, proto::list2< T, U > >    type;
    typedef proto::basic_expr< proto::tag::mem_ptr, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<mem_ptr>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::mem_ptr::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<mem_ptr>::template impl<Expr, State, Data> {
};
```


Struct template assign

`boost::proto::assign` — A metafunction for generating assignment expression types, a grammar element for matching assignment expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct assign : proto::transform< assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::assign, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::pass_through<assign>::template impl<Expr, State, Data> {
};
```


Struct template `shift_left_assign`

`boost::proto::shift_left_assign` — A metafunction for generating left-shift-assign expression types, a grammar element for matching left-shift-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct shift_left_assign : proto::transform< shift_left_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::shift_left_assign, proto::list2< T, U > >      type;      ↵

    typedef proto::basic_expr< proto::tag::shift_left_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<shift_left_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::shift_left_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<shift_left_assign>::template impl<Expr, State, Data>
{
};
```


Struct template `shift_right_assign`

`boost::proto::shift_right_assign` — A metafunction for generating right-shift-assign expression types, a grammar element for matching right-shift-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct shift_right_assign : proto::transform< shift_right_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::shift_right_assign, proto::list2< T, U > >      type;      ↵

    typedef proto::basic_expr< proto::tag::shift_right_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::pass_through<shift_right_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::shift_right_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<shift_right_assign>::template impl<Expr, State, Data>
{
};
```


Struct template multiplies_assign

boost::proto::multiplies_assign — A metafunction for generating multiplies-assign expression types, a grammar element for matching multiplies-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct multiplies_assign : proto::transform< multiplies_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::multiplies_assign, proto::list2< T, U > > type;      ↵

    typedef proto::basic_expr< proto::tag::multiplies_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<multiplies_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::multiplies_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<multiplies_assign>::template impl<Expr, State, Data>
{
};
```


Struct template divides_assign

boost::proto::divides_assign — A metafunction for generating divides-assign expression types, a grammar element for matching divides-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct divides_assign : proto::transform< divides_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::divides_assign, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::divides_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<divides_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::divides_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<divides_assign>::template impl<Expr, State, Data>
{
};
```


Struct template modulus_assign

boost::proto::modulus_assign — A metafunction for generating modulus-assign expression types, a grammar element for matching modulus-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct modulus_assign : proto::transform< modulus_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::modulus_assign, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::modulus_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<modulus_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::modulus_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<modulus_assign>::template impl<Expr, State, Data>
{
};
```


Struct template plus_assign

boost::proto::plus_assign — A metafunction for generating plus-assign expression types, a grammar element for matching plus-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct plus_assign : proto::transform< plus_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::plus_assign, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::plus_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<plus_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::plus_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<plus_assign>::template impl<Expr, State, Data>
{
};
```


Struct template `minus_assign`

`boost::proto::minus_assign` — A metafunction for generating minus-assign expression types, a grammar element for matching minus-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct minus_assign : proto::transform< minus_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::minus_assign, proto::list2< T, U > >      type;
    typedef proto::basic_expr< proto::tag::minus_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<minus_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::minus_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<minus_assign>::template impl<Expr, State, Data>
{
};
```


Struct template `bitwise_and_assign`

`boost::proto::bitwise_and_assign` — A metafunction for generating bitwise-and-assign expression types, a grammar element for matching bitwise-and-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct bitwise_and_assign : proto::transform< bitwise_and_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::bitwise_and_assign, proto::list2< T, U > > type;      ↵

    typedef proto::basic_expr< proto::tag::bitwise_and_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::pass_through<bitwise_and_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::bitwise_and_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<bitwise_and_assign>::template impl<Expr, State, Data>
{
};
```


Struct template bitwise_or_assign

boost::proto::bitwise_or_assign — A metafunction for generating bitwise-or-assign expression types, a grammar element for matching bitwise-or-assign expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct bitwise_or_assign : proto::transform< bitwise_or_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::bitwise_or_assign, proto::list2< T, U > > type;

    typedef proto::basic_expr< proto::tag::bitwise_or_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<bitwise_or_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::bitwise_or_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<bitwise_or_assign>::template impl<Expr, State, Data>
{
};
```


Struct template bitwise_xor_assign

boost::proto::bitwise_xor_assign — A metafunction for generating bitwise-xor-assign expression types, a grammar element for matching bitwise-xor-assign expressions, and a [PrimitiveTransform](#) that dispatches to the [proto::pass_through<>](#) transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct bitwise_xor_assign : proto::transform< bitwise_xor_assign<T, U> > {
    // types
    typedef proto::expr< proto::tag::bitwise_xor_assign, proto::list2< T, U > > type;      ↵

    typedef proto::basic_expr< proto::tag::bitwise_xor_assign, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::pass_through<bitwise_xor_assign>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::bitwise_xor_assign::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<bitwise_xor_assign>::template impl<Expr, State, Data>
{
};
```


Struct template subscript

`boost::proto::subscript` — A metafunction for generating subscript expression types, a grammar element for matching subscript expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T, typename U>
struct subscript : proto::transform< subscript<T, U> > {
    // types
    typedef proto::expr< proto::tag::subscript, proto::list2< T, U > > type;
    typedef proto::basic_expr< proto::tag::subscript, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<subscript>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::subscript::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<subscript>::template impl<Expr, State, Data>
{
};
```


Struct template function

`boost::proto::function` — A metafunction for generating function-call expression types, a grammar element for matching function-call expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename... A>
struct function : proto::transform< function<A...> > {
    // types
    typedef proto::expr< proto::tag::function, proto::listN< A... > > type;
    typedef proto::basic_expr< proto::tag::function, proto::listN< A... > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<function>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Struct template impl

boost::proto::function::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<function>::template impl<Expr, State, Data>
{
};
```


Struct template nullary_expr

boost::proto::nullary_expr — A metafunction for generating nullary expression types, a grammar element for matching nullary expressions, and a [PrimitiveTransform](#) that returns the current expression unchanged.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Tag, typename T>
struct nullary_expr : proto::transform< nullary_expr<Tag, T> > {
    // types
    typedef proto::expr< Tag, proto::term< T > >      type;
    typedef proto::basic_expr< Tag, proto::term< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef Expr result_type;

        // public member functions
        Expr operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

Use proto::nullary_expr<proto::_, proto::_> as a grammar element to match any nullary expression.

Struct template impl

boost::proto::nullary_expr::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef Expr result_type;

    // public member functions
    Expr operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1. Expr operator()(typename impl::expr_param expr, typename impl::state_param,
 typename impl::data_param) const;

Parameters: expr The current expression
Requires: proto::matches<Expr, proto::nullary_expr<Tag, T> >::value is true.
Returns: expr
Throws: Will not throw.

Struct template unary_expr

boost::proto::unary_expr — A metafunction for generating unary expression types with a specified tag type, a grammar element for matching unary expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Tag, typename T>
struct unary_expr : proto::transform< unary_expr<Tag, T> > {
    // types
    typedef proto::expr< Tag, proto::list1< T > >      type;
    typedef proto::basic_expr< Tag, proto::list1< T > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<unary_expr>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Use `proto::unary_expr<proto::_, proto::_>` as a grammar element to match any unary expression.

Struct template impl

boost::proto::unary_expr::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<unary_expr>::template impl<Expr, State, Data>
{
};
```


Struct template `binary_expr`

`boost::proto::binary_expr` — A metafunction for generating binary expression types with a specified tag type, a grammar element for matching binary expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Tag, typename T, typename U>
struct binary_expr : proto::transform< binary_expr<Tag, T, U> > {
    // types
    typedef proto::expr< Tag, proto::list2< T, U > >      type;
    typedef proto::basic_expr< Tag, proto::list2< T, U > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<binary_expr>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Use `proto::binary_expr<proto::_ , proto::_ , proto::_>` as a grammar element to match any binary expression.

Struct template impl

boost::proto::binary_expr::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<binary_expr>::template impl<Expr, State, Data>
{
};
```


Struct template `nary_expr`

`boost::proto::nary_expr` — A metafunction for generating n-ary expression types with a specified tag type, a grammar element for matching n-ary expressions, and a [PrimitiveTransform](#) that dispatches to the `proto::pass_through<>` transform.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Tag, typename... A>
struct nary_expr : proto::transform< nary_expr<Tag, A...> > {
    // types
    typedef proto::expr< Tag, proto::listN< A... > >      type;
    typedef proto::basic_expr< Tag, proto::listN< A... > > proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::pass_through<nary_expr>::template impl<Expr, State, Data>
    {
    };
};
```

Description

Use `proto::nary_expr<proto::_ , proto::vararg<proto::_> >` as a grammar element to match any n-ary expression; that is, any non-terminal.

Struct template impl

boost::proto::nary_expr::impl

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::pass_through<nary_expr>::template impl<Expr, State, Data>
{
};
```


Struct template `is_expr`

`boost::proto::is_expr` — A Boolean metafunction that indicates whether a given type `T` is a Proto expression type.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
struct is_expr : mpl::bool_<true-or-false> {
};
```

Description

If `T` is an instantiation of `proto::expr<>` or `proto::basic_expr<>` or is an extension (via `proto::extends<>` or `BOOST_PROTO_EXTENDS()`) of such an instantiation, `proto::is_expr<T>::value` is true. Otherwise, `proto::is_expr<T>::value` is false.

Struct template tag_of

boost::proto::tag_of — A metafunction that returns the tag type of a Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
struct tag_of {
    // types
    typedef typename Expr::proto_tag type;
};
```


Struct template arity_of

boost::proto::arity_of — A metafunction that returns the arity of a Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
struct arity_of : Expr::proto_arity {
};
```


Function `as_expr`

`boost::proto::as_expr` — A function that wraps non-Proto expression types in Proto terminals and leaves Proto expression types alone.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
    typename proto::result_of::as_expr< T >::type as_expr(T & t);
template<typename T>
    typename proto::result_of::as_expr< T const >::type as_expr(T const & t);
template<typename Domain, typename T>
    typename proto::result_of::as_expr< T, Domain >::type as_expr(T & t);
template<typename Domain, typename T>
    typename proto::result_of::as_expr< T const, Domain >::type
    as_expr(T const & t);
```

Description

The `proto::as_expr()` function returns Proto expression objects that are suitable for storage in a local variable. It turns non-Proto objects into Proto terminals. Its behavior is domain-specific. By default, non-Proto types are wrapped by value (if possible) in a new Proto terminal expression, and objects that are already Proto expressions are returned by value.

If `Domain` is not explicitly specified, it is assumed to be `proto::default_domain`.

See `proto::domain::as_expr<>` for a complete description of this function's default behavior.

Returns: `typename Domain::template as_expr< T >()(t)`

Function `as_child`

`boost::proto::as_child` — A function that wraps non-Proto objects in Proto terminals (by reference) and leaves Proto expression types alone.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename T>
    typename proto::result_of::as_child< T >::type as_child(T & t);
template<typename T>
    typename proto::result_of::as_child< T const >::type as_child(T const & t);
template<typename Domain, typename T>
    typename proto::result_of::as_child< T, Domain >::type as_child(T & t);
template<typename Domain, typename T>
    typename proto::result_of::as_child< T const, Domain >::type
    as_child(T const & t);
```

Description

The `proto::as_child()` function returns Proto expression objects that are suitable for storage as child nodes in an expression tree. It turns non-Proto objects into Proto terminals. Its behavior is domain-specific. By default, non-Proto types are held wrapped by reference in a new Proto terminal expression, and objects that are already Proto expressions are simply returned by reference.

If `Domain` is not explicitly specified, it is assumed to be `proto::default_domain`.

See `proto::domain::as_child<>` for a complete description of this function's default behavior.

Returns: `typename Domain::template as_child< T >()(t)`

Function child

`boost::proto::child` — Return the N^{th} child of the specified Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename N, typename Expr>
    typename proto::result_of::child< Expr &, N >::type child(Expr & expr);
template<typename N, typename Expr>
    typename proto::result_of::child< Expr const &, N >::type
        child(Expr const & expr);
template<typename Expr>
    typename proto::result_of::child< Expr & >::type child(Expr & expr);
template<typename Expr>
    typename proto::result_of::child< Expr const & >::type
        child(Expr const & expr);
```

Description

Return the N^{th} child of the specified Proto expression. If N is not specified, as in `proto::child(expr)`, then N is assumed to be `mpl::long_<0>`. The child is returned by reference.

Parameters: `expr` The Proto expression.

Requires: `proto::is_expr<Expr>::value` is true.

N is an MPL Integral Constant.

`N::value < Expr::proto_arity::value`

Returns: A reference to the N^{th} child of `expr`.

Throws: Will not throw.

Function child_c

boost::proto::child_c — Return the N^{th} child of the specified Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<long N, typename Expr>
    typename proto::result_of::child_c< Expr &, N >::type child_c(Expr & expr);
template<long N, typename Expr>
    typename proto::result_of::child_c< Expr const &, N >::type
    child_c(Expr const & expr);
```

Description

Return the N^{th} child of the specified Proto expression. The child is returned by reference.

Requires: [proto::is_expr](#)<Expr>::value is true.

Returns: N < Expr::proto_arity::value

 A reference to the N^{th} child of expr.

Throws: Will not throw.

Function value

`boost::proto::value` — Return the value stored within the specified Proto terminal expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
    typename proto::result_of::value< Expr & >::type value(Expr & expr);
template<typename Expr>
    typename proto::result_of::value< Expr const & >::type
    value(Expr const & expr);
```

Description

Return the the value stored within the specified Proto terminal expression. The value is returned by reference.

Requires: `0 == Expr::proto_arity::value`
Returns: A reference to the terminal's value
Throws: Will not throw.

Function left

boost::proto::left — Return the left child of the specified binary Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
    typename proto::result_of::left< Expr & >::type left(Expr & expr);
template<typename Expr>
    typename proto::result_of::left< Expr const & >::type
    left(Expr const & expr);
```

Description

Return the left child of the specified binary Proto expression. The child is returned by reference.

Requires: [proto::is_expr](#)<Expr>::value is true.

 2 == Expr::proto_arity::value

Returns: A reference to the left child of expr.

Throws: Will not throw.

Function right

boost::proto::right — Return the right child of the specified binary Proto expression.

Synopsis

```
// In header: <boost/proto/traits.hpp>

template<typename Expr>
    typename proto::result_of::right< Expr & >::type right(Expr & expr);
template<typename Expr>
    typename proto::result_of::right< Expr const & >::type
    right(Expr const & expr);
```

Description

Return the right child of the specified binary Proto expression. The child is returned by reference.

Parameters: `expr` The Proto expression.
Requires: `proto::is_expr<Expr>::value` is true.

Returns: `2 == Expr::proto_arity::value`
 A reference to the right child of `expr`.
Throws: Will not throw.

Header **<boost/proto/transform/arg.hpp>**

Contains definition of the childN transforms and friends.

```
namespace boost {
    namespace proto {
        struct _expr;
        struct _state;
        struct _data;
        template<int N> struct _child_c;
        struct _value;
        struct _void;
        struct _byref;
        struct _byval;
    }
}
```


Struct `_expr`

`boost::proto::_expr` — A [PrimitiveTransform](#) that returns the current expression unmodified.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

struct _expr : proto::transform< _expr > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef Expr result_type;

        // public member functions
        Expr operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

Example:

```
proto::terminal<int>::type i = {42};
proto::terminal<int>::type & j = proto::_expr()(i);
assert( boost::addressof(i) == boost::addressof(j) );
```


Struct template impl

boost::proto::_expr::impl

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef Expr result_type;

    // public member functions
    Expr operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1. Expr operator()(typename impl::expr_param expr, typename impl::state_param,
 typename impl::data_param) const;

Returns the current expression.

Parameters: expr The current expression.

Returns: expr

Throws: Will not throw.

Struct `_state`

`boost::proto::_state` — A [PrimitiveTransform](#) that returns the current state unmodified.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

struct _state : proto::transform< _state > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef State result_type;

        // public member functions
        State operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

Example:

```
proto::terminal<int>::type i = {42};
char ch = proto::_state()(i, 'a');
assert( ch == 'a' );
```


Struct template impl

boost::proto::_state::impl

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef State result_type;

    // public member functions
    State operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1. State `operator()(typename impl::expr_param, typename impl::state_param state, typename impl::data_param) const;`

Returns the current state.

Parameters: state The current state.

Returns: state

Throws: Will not throw.

Struct _data

boost::proto::_data — A [PrimitiveTransform](#) that returns the current data unmodified.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

struct _data : proto::transform< _data > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef Data result_type;

        // public member functions
        Data operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

Example:

```
proto::terminal<int>::type i = {42};
std::string str("hello");
std::string & data = proto::_data()(i, 'a', str);
assert( &str == &data );
```


Struct template impl

boost::proto::_data::impl

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef Data result_type;

    // public member functions
    Data operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1. Data operator()(typename impl::expr_param, typename impl::state_param,
 typename impl::data_param data) const;

Returns the current data.

Parameters: data The current data.

Returns: data

Throws: Will not throw.

Struct template `_child_c`

`boost::proto::_child_c` — A [PrimitiveTransform](#) that returns N-th child of the current expression.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<int N>
struct _child_c : proto::transform< _child_c<N> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef typename proto::result_of::child_c< Expr, N >::type result_type;

        // public member functions
        typename proto::result_of::child_c< Expr, N >::type
        operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
    };
};
```

Description

Example:

```
proto::terminal<int>::type i = {42};
proto::terminal<int>::type & j = proto::_child_c<0>()(i);
assert( boost::addressof(i) == boost::addressof(j) );
```


Struct template impl

boost::proto::child_c::impl

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef typename proto::result_of::child_c< Expr, N >::type result_type;

    // public member functions
    typename proto::result_of::child_c< Expr, N >::type
    operator()(typename impl::expr_param expr, typename impl::state_param,
               typename impl::data_param) const;
};
```

Description

impl public member functions

1.

```
typename proto::result_of::child_c< Expr, N >::type
operator()(typename impl::expr_param expr, typename impl::state_param,
           typename impl::data_param) const;
```

Returns the N-th child of `expr`

Parameters: `expr` The current expression.

Requires: `Expr::proto_arity::value > N`

Returns: `proto::child_c<N>(expr)`

Throws: Will not throw.

Struct _value

boost::proto::_value — A [PrimitiveTransform](#) that returns the value of the current terminal expression.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

struct _value : proto::transform< _value > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef typename proto::result_of::value< Expr >::type result_type;

        // public member functions
        typename proto::result_of::value< Expr >::type
        operator()(typename impl::expr_param, typename impl::state_param,
                   typename impl::data_param) const;
    };
};
```

Description

Example:

```
proto::terminal<int>::type i = {42};
int j = proto::_value()(i);
assert( 42 == j );
```


Struct template impl

boost::proto::_value::impl

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef typename proto::result_of::value< Expr >::type result_type;

    // public member functions
    typename proto::result_of::value< Expr >::type
    operator()(typename impl::expr_param, typename impl::state_param,
               typename impl::data_param) const;
};
```

Description

impl public member functions

1.

```
typename proto::result_of::value< Expr >::type
operator()(typename impl::expr_param expr, typename impl::state_param,
           typename impl::data_param) const;
```

Returns the value of the specified terminal expression.

Parameters: expr The current expression.

Requires: Expr::proto_arity::value == 0.

Returns: proto::value(expr)

Throws: Will not throw.

Struct `_void`

`boost::proto::_void` — A [PrimitiveTransform](#) that does nothing and returns void.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

struct _void : proto::transform< _void > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef void result_type;

        // public member functions
        void operator()(typename impl::expr_param, typename impl::state_param,
                        typename impl::data_param) const;
    };
};
```

Description

Struct template impl

boost::proto::_void::impl

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef void result_type;

    // public member functions
    void operator()(typename impl::expr_param, typename impl::state_param,
                    typename impl::data_param) const;
};
```

Description

impl public member functions

1.

```
void operator()(typename impl::expr_param, typename impl::state_param,
                typename impl::data_param) const;
```

Does nothing.

Throws: Will not throw.

Struct `_byref`

`boost::proto::_byref` — A unary callable [PolymorphicFunctionObject](#) that wraps its argument in a `boost::reference_wrapper<>`.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

struct _byref : proto::callable {
    // member classes/structs/unions
    template<typename This, typename T>
    struct result<This(T &)> {
        // types
        typedef boost::reference_wrapper< T > const type;
    };
    template<typename This, typename T>
    struct result<This(T)> {
        // types
        typedef boost::reference_wrapper< T const > const type;
    };

    // public member functions
    template<typename T>
    boost::reference_wrapper< T > const operator()(T &) const;
    template<typename T>
    boost::reference_wrapper< T const > const operator()(T const &) const;
};
```

Description

Example:

```
proto::terminal<int>::type i = {42};
boost::reference_wrapper<proto::terminal<int>::type> j
    = proto::when<proto::_ , proto::_byref(_)>()(i);
assert( boost::addressof(i) == boost::addressof(j.get()) );
```

`_byref` public member functions

1.

```
template<typename T>
    boost::reference_wrapper< T > const operator()(T & t) const;
```

Wrap the parameter `t` in a `boost::reference_wrapper<>`

Parameters: `t` The object to wrap

Returns: `boost::ref(t)`

Throws: Will not throw.

2.

```
template<typename T>
    boost::reference_wrapper< T const > const operator()(T const & t) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(T &)>

boost::proto::_byref::result<This(T &)>

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename This, typename T>
struct result<This(T &)> {
    // types
    typedef boost::reference_wrapper< T > const type;
};
```


Struct template result<This(T)>

boost::proto::_byref::result<This(T)>

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename This, typename T>
struct result<This(T)> {
    // types
    typedef boost::reference_wrapper< T const > const type;
};
```


Struct `_byval`

`boost::proto::_byval` — A unary callable [PolymorphicFunctionObject](#) that strips references and `boost::reference_wrapper<>` from its argument.

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

struct _byval : proto::callable {
    // member classes/structs/unions
    template<typename This, typename T>
    struct result<This(boost::reference_wrapper< T >)> : result<This(T)> {
    };
    template<typename This, typename T>
    struct result<This(T &)> : result<This(T)> {
    };
    template<typename This, typename T>
    struct result<This(T)> {
        // types
        typedef T type;
    };

    // public member functions
    template<typename T> T operator()(T const &) const;
    template<typename T>
    T operator()(boost::reference_wrapper< T > const &) const;
};
```

Description

Example:

```
proto::terminal<int>::type i = {42};
int j = 67;
int k = proto::when<proto::_ , proto::_byval(proto::_state)>()(i, boost::ref(j));
assert( 67 == k );
```

`_byval` public member functions

1.

```
template<typename T> T operator()(T const & t) const;
```

Parameters: `t` The object to unref
Returns: `t`
Throws: Will not throw.

2.

```
template<typename T>
    T operator()(boost::reference_wrapper< T > const & t) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result<This(boost::reference_wrapper< T >)>

boost::proto::_byval::result<This(boost::reference_wrapper< T >)>

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename This, typename T>
struct result<This(boost::reference_wrapper< T >)> : result<This(T)> {
};
```


Struct template result<This(T &)>

boost::proto::_byval::result<This(T &)>

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename This, typename T>
struct result<This(T &)> : result<This(T)> {
};
```


Struct template result<This(T)>

boost::proto::_byval::result<This(T)>

Synopsis

```
// In header: <boost/proto/transform/arg.hpp>

template<typename This, typename T>
struct result<This(T)> {
    // types
    typedef T type;
};
```

Header <boost/proto/transform/call.hpp>

Contains definition of the call<> transform.

```
namespace boost {
    namespace proto {
        template<typename T> struct call;
    }
}
```


Struct template call

`boost::proto::call` — Make the given [CallableTransform](#) into a [PrimitiveTransform](#).

Synopsis

```
// In header: <boost/proto/transform/call.hpp>

template<typename T>
struct call : proto::transform< call<T> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl<Expr, State, Data> {
        // types
        typedef see-below result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

The purpose of `proto::call<>` is to annotate a transform as callable so that `proto::when<>` knows how to apply it. The template parameter must be either a [PrimitiveTransform](#) or a [CallableTransform](#); that is, a function type for which the return type is a callable [PolymorphicFunctionObject](#).

For the complete description of the behavior of the `proto::call<>` transform, see the documentation for the nested `proto::call::impl<>` class template.

Struct template impl

boost::proto::call::impl

Synopsis

```
// In header: <boost/proto/transform/call.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl<Expr, State, Data> {
    // types
    typedef see-below result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                          typename impl::state_param,
                          typename impl::data_param) const;
};
```

Description

impl public types

1. typedef *see-below* result_type;

boost::call<T>::impl<Expr, State, Data>::result_type is computed as follows:

- If T is of the form `PrimitiveTransform` or `PrimitiveTransform()`, then result_type is:

```
typename boost::result_of<PrimitiveTransform(Expr, State, Data)>::type
```

- If T is of the form `PrimitiveTransform(A0)`, then result_type is:

```
typename boost::result_of<PrimitiveTransform(
    typename boost::result_of<when<_, A0>(Expr, State, Data)>::type,
    State,
    Data
)>::type
```

- If T is of the form `PrimitiveTransform(A0, A1)`, then result_type is:

```
typename boost::result_of<PrimitiveTransform(
    typename boost::result_of<when<_, A0>(Expr, State, Data)>::type,
    typename boost::result_of<when<_, A1>(Expr, State, Data)>::type,
    Data
)>::type
```

- If T is of the form `PrimitiveTransform(A0, A1, A2)`, then result_type is:

```
typename boost::result_of<PrimitiveTransform(
    typename boost::result_of<when<_, A0>(Expr, State, Data)>::type,
    typename boost::result_of<when<_, A1>(Expr, State, Data)>::type,
    typename boost::result_of<when<_, A2>(Expr, State, Data)>::type
)>::type
```


- If T is of the form `PolymorphicFunctionObject`(A_0, \dots, A_n), then `result_type` is:

```
typename boost::result_of<PolymorphicFunctionObject(
    typename boost::result_of<when<_,A0>(Expr, State, Data)>::type,
    ...
    typename boost::result_of<when<_,An>(Expr, State, Data)>::type
>::type
```

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                      typename impl::state_param state,
                      typename impl::data_param data) const;
```

`proto::call<T>::impl<Expr, State, Data>::operator()` behaves as follows:

- If T is of the form `PrimitiveTransform` or `PrimitiveTransform()`, then return

```
PrimitiveTransform()(expr, state, data)
```

- If T is of the form `PrimitiveTransform`(A_0), then return

```
PrimitiveTransform()(
    when<_,A0>()(expr, state, data),
    state,
    sata
)
```

- If T is of the form `PrimitiveTransform`(A_0, A_1), then return:

```
PrimitiveTransform()(
    when<_,A0>()(expr, state, data),
    when<_,A1>()(expr, state, data),
    Data
)
```

- If T is of the form `PrimitiveTransform`(A_0, A_1, A_2), then return

```
PrimitiveTransform()(
    when<_,A0>()(expr, state, data),
    when<_,A1>()(expr, state, data),
    when<_,A2>()(expr, state, data)
)
```

- If T is of the form `PolymorphicFunctionObject`(A_0, \dots, A_n), then return:

```
PolymorphicFunctionObject()(
    when<_,A0>()(expr, state, data),
    ...
    when<_,An>()(expr, state, data)
)
```


Header <[boost/proto/transform/default.hpp](#)>

```
namespace boost {  
  namespace proto {  
    template<typename Grammar = unspecified> struct _default;  
  }  
}
```


Struct template `_default`

`boost::proto::_default` — A [PrimitiveTransform](#) that gives expressions their usual C++ behavior

Synopsis

```
// In header: <boost/proto/transform/default.hpp>

template<typename Grammar = unspecified>
struct _default : proto::transform< _default<Grammar> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl<Expr, State, Data> {
        // types
        typedef typename Expr::tag_type Tag;           // For exposition only
        typedef see-below result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;

        static Expr s_expr;      // For exposition only
        static State s_state;     // For exposition only
        static Data s_data;      // For exposition only
    };
};
```

Description

For the complete description of the behavior of the `proto::_default` transform, see the documentation for the nested `proto::_default::impl<>` class template.

When used without specifying a `Grammar` parameter, `proto::_default` behaves as if the parameter were `proto::_default<>`.

Struct template impl

boost::proto::_default::impl

Synopsis

```
// In header: <boost/proto/transform/default.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl<Expr, State, Data> {
    // types
    typedef typename Expr::tag_type Tag;           // For exposition only
    typedef see-below result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                           typename impl::state_param,
                           typename impl::data_param) const;
    static Expr s_expr; // For exposition only
    static State s_state; // For exposition only
    static Data s_data; // For exposition only
};
```

Description

Let OP be the C++ operator corresponding to `Expr::proto_tag`. (For example, if Tag is `proto::tag::plus`, let OP be +.)

The behavior of this class is specified in terms of the C++0x `decltype` keyword. In systems where this keyword is not available, Proto uses the Boost.Typeof library to approximate the behavior.

impl public types

1. typedef *see-below* result_type;

- If Tag corresponds to a unary prefix operator, then the result type is

```
decltype(
    OP Grammar()(proto::child(s_expr), s_state, s_data)
)
```

- If Tag corresponds to a unary postfix operator, then the result type is

```
decltype(
    Grammar()(proto::child(s_expr), s_state, s_data) OP
)
```

- If Tag corresponds to a binary infix operator, then the result type is

```
decltype(
    Grammar()(proto::left(s_expr), s_state, s_data) OP
    Grammar()(proto::right(s_expr), s_state, s_data)
)
```

- If Tag is `proto::tag::subscript`, then the result type is


```
decltype(
  Grammar()(proto::left(s_expr), s_state, s_data) [
    Grammar()(proto::right(s_expr), s_state, s_data) ]
)
```

- If Tag is `proto::tag::if_else_` , then the result type is

```
decltype(
  Grammar()(proto::child_c<0>(s_expr), s_state, s_data) ?
  Grammar()(proto::child_c<1>(s_expr), s_state, s_data) :
  Grammar()(proto::child_c<2>(s_expr), s_state, s_data)
)
```

- If Tag is `proto::tag::function` , then the result type is

```
decltype(
  Grammar()(proto::child_c<0>(s_expr), s_state, s_data) (
  Grammar()(proto::child_c<1>(s_expr), s_state, s_data),
  ...
  Grammar()(proto::child_c<N>(s_expr), s_state, s_data) )
)
```

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                      typename impl::state_param state,
                      typename impl::data_param data) const;
```

`proto::_default<Grammar>::impl<Expr, State, Data>::operator()`

- If Tag corresponds to a unary prefix operator, then return

```
OP Grammar()(proto::child(expr), state, data)
```

- If Tag corresponds to a unary postfix operator, then return

```
Grammar()(proto::child(expr), state, data) OP
```

- If Tag corresponds to a binary infix operator, then return

```
Grammar()(proto::left(expr), state, data) OP
Grammar()(proto::right(expr), state, data)
```

- If Tag is `proto::tag::subscript` , then return

```
Grammar()(proto::left(expr), state, data) [
  Grammar()(proto::right(expr), state, data) ]
```

- If Tag is `proto::tag::if_else_` , then return


```
Grammar()(proto::child_c<0>(expr), state, data) ?  
Grammar()(proto::child_c<1>(expr), state, data) :  
Grammar()(proto::child_c<2>(expr), state, data)
```

- If Tag is `proto::tag::function` , then return

```
Grammar()(proto::child_c<0>(expr), state, data) (  
Grammar()(proto::child_c<1>(expr), state, data),  
...  
Grammar()(proto::child_c<N>(expr), state, data) )
```

Header <boost/proto/transform/fold.hpp>

Contains definition of the `proto::fold<>` and `proto::reverse_fold<>` transforms.

```
namespace boost {  
  namespace proto {  
    template<typename Sequence, typename State0, typename Fun> struct fold;  
    template<typename Sequence, typename State0, typename Fun>  
      struct reverse_fold;  
  }  
}
```


Struct template fold

`boost::proto::fold` — A [PrimitiveTransform](#) that invokes the `fusion::fold<>` algorithm to accumulate a value.

Synopsis

```
// In header: <boost/proto/transform/fold.hpp>

template<typename Sequence, typename State0, typename Fun>
struct fold : proto::transform< fold<Sequence, State0, Fun> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef when<_, Sequence>                                X;           // For exposition
    public:
        typedef when<_, State0>                                  Y;           // For exposition
    public:
        typedef typename boost::result_of<X(Expr, State, Data)>::type seq;       // A Fusion sequence, for exposition only
        typedef typename boost::result_of<Y(Expr, State, Data)>::type state0;     // An initial state for the fold, for exposition only
        typedef unspecified fun;           // fun(d)(s,e)
    public:
        == when<_,Fun>()(e,s,d)
        typedef typename fusion::result_of::fold<seq, state0, fun>::type result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

For the complete description of the behavior of the `proto::fold<>` transform, see the documentation for the nested `proto::fold::impl<>` class template.

Struct template impl

boost::proto::fold::impl

Synopsis

```
// In header: <boost/proto/transform/fold.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef when<_, Sequence> X;           // For exposition
    typedef when<_, State0> Y;             // For exposition
    typedef typename boost::result_of<X(Expr, State, Data)>::type seq; // A Fusion sequence, for exposition only
    typedef typename boost::result_of<Y(Expr, State, Data)>::type state0; // An initial state for the fold, for exposition only
    typedef unspecified fun;               // fun(d)(s,e)
    == when<_,Fun>()(e,s,d)
    typedef typename fusion::result_of::fold<seq, state0, fun>::type result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                           typename impl::state_param,
                           typename impl::data_param) const;
};
```

Description

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                       typename impl::state_param state,
                       typename impl::data_param data) const;
```

Let seq be `when<_, Sequence>()(expr, state, data)`, let state0 be `when<_, State0>()(expr, state, data)`, and let `fun(data)` be an object such that `fun(data)(state, expr)` is equivalent to `when<_, Fun>()(expr, state, data)`. Then, this function returns `fusion::fold(seq, state0, fun(data))`.

Parameters:

data	An arbitrary data
expr	The current expression
state	The current state

Struct template `reverse_fold`

`boost::proto::reverse_fold` — A [PrimitiveTransform](#) that is the same as the `proto::fold<>` transform, except that it folds back-to-front instead of front-to-back. It uses the `proto::_reverse` callable [PolymorphicFunctionObject](#) to create a `fusion::reverse_view<>` of the sequence before invoking `fusion::fold<>`.

Synopsis

```
// In header: <boost/proto/transform/fold.hpp>

template<typename Sequence, typename State0, typename Fun>
struct reverse_fold : proto::fold< proto::_reverse(Sequence), State0, Fun > {
};
```

Header `<boost/proto/transform/fold_tree.hpp>`

Contains definition of the `proto::fold_tree<>` and `proto::reverse_fold_tree<>` transforms.

```
namespace boost {
  namespace proto {
    template<typename Sequence, typename State0, typename Fun> struct fold_tree;
    template<typename Sequence, typename State0, typename Fun>
      struct reverse_fold_tree;
  }
}
```


Struct template fold_tree

boost::proto::fold_tree — A [PrimitiveTransform](#) that recursively applies the `proto::fold<>` transform to sub-trees that all share a common tag type.

Synopsis

```
// In header: <boost/proto/transform/fold_tree.hpp>

template<typename Sequence, typename State0, typename Fun>
struct fold_tree : proto::transform< fold_tree<Sequence, State0, Fun> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::fold<Sequence, State0, recurse_if_<typename Expr::proto_tag, Fun> >
        ::template impl<Expr, State, Data>
    {
    };
};
```

Description

`proto::fold_tree<>` is useful for flattening trees into lists; for example, you might use `proto::fold_tree<>` to flatten an expression tree like `a | b | c` into a Fusion list like `cons(c, cons(b, cons(a)))`.

`proto::fold_tree<>` is easily understood in terms of a `recurse_if_<>` helper, defined as follows:

```
template<typename Tag, typename Fun>
struct recurse_if_ :
    proto::if_<
        // If the current node has type type "Tag" ...
        boost::is_same<proto::tag_of<proto::_>, Tag>(),
        // ... recurse, otherwise ...
        proto::fold<proto::_, proto::_state, recurse_if_<Tag, Fun> >,
        // ... apply the Fun transform.
        Fun
    >
{
};
```

With `recurse_if_<>` as defined above, `proto::fold_tree<Sequence, State0, Fun>()(expr, state, data)` is equivalent to:

```
proto::fold<
    Sequence,
    State0,
    recurse_if_<typename Expr::proto_tag, Fun>
>()(expr, state, data).
```

It has the effect of folding a tree front-to-back, recursing into child nodes that share a tag type with the parent node.

Struct template impl

boost::proto::fold_tree::impl

Synopsis

```
// In header: <boost/proto/transform/fold_tree.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::fold<Sequence, State0, recurse_if_<typename Expr::proto_tag, Fun> >
        ::template impl<Expr, State, Data>
{
};
```


Struct template reverse_fold_tree

boost::proto::reverse_fold_tree — A [PrimitiveTransform](#) that recursively applies the `proto::reverse_fold<>` transform to subtrees that all share a common tag type.

Synopsis

```
// In header: <boost/proto/transform/fold_tree.hpp>

template<typename Sequence, typename State0, typename Fun>
struct reverse_fold_tree :
    proto::transform< reverse_fold_tree<Sequence, State0, Fun> >
{
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl :
        proto::reverse_fold<Sequence, State0, recurse_if_<typename Expr::proto_tag, Fun> >
        ::template impl<Expr, State, Data>
    {
    };
};
```

Description

`proto::reverse_fold_tree<>` is useful for flattening trees into lists; for example, you might use `proto::reverse_fold_tree<>` to flatten an expression tree like `a | b | c` into a Fusion list like `cons(a, cons(b, cons(c)))`.

`proto::reverse_fold_tree<>` is easily understood in terms of a `recurse_if_<>` helper, defined as follows:

```
template<typename Tag, typename Fun>
struct recurse_if_ :
    proto::if_<
        // If the current node has type type "Tag" ...
        boost::is_same<proto::tag_of<proto::_>, Tag>(),
        // ... recurse, otherwise ...
        proto::reverse_fold<proto::_ , proto::_state, recurse_if_<Tag, Fun> >,
        // ... apply the Fun transform.
        Fun
    >
{
};
```

With `recurse_if_<>` as defined above, `proto::reverse_fold_tree<Sequence, State0, Fun>()(expr, state, data)` is equivalent to:

```
proto::reverse_fold<
    Sequence,
    State0,
    recurse_if_<typename Expr::proto_tag, Fun>
>()(expr, state, data).
```

It has the effect of folding a tree back-to-front, recursing into child nodes that share a tag type with the parent node.

Struct template impl

boost::proto::reverse_fold_tree::impl

Synopsis

```
// In header: <boost/proto/transform/fold_tree.hpp>

template<typename Expr, typename State, typename Data>
struct impl :
    proto::reverse_fold<Sequence, State0, recurse_if_<typename Expr::proto_tag, Fun> >
    ::template impl<Expr, State, Data>
{
};
```

Header <boost/proto/transform/impl.hpp>

Contains definition of transform<> and transform_impl<> helpers.

```
namespace boost {
    namespace proto {
        template<typename PrimitiveTransform> struct transform;
        template<typename Expr, typename State, typename Data>
            struct transform_impl;
    }
}
```


Struct template transform

boost::proto::transform — Inherit from this to make your type a [PrimitiveTransform](#).

Synopsis

```
// In header: <boost/proto/transform/impl.hpp>

template<typename PrimitiveTransform>
struct transform {
    // types
    typedef PrimitiveTransform transform_type;

    // member classes/structs/unions
    template<typename This, typename Expr>
    struct result<This(Expr)> {
        // types
        typedef typename PrimitiveTransform::template impl< Expr, unspecified, unspecified >::result_type type;
    };
    template<typename This, typename Expr, typename State>
    struct result<This(Expr, State)> {
        // types
        typedef typename PrimitiveTransform::template impl< Expr, State, unspecified >::result_type type;
    };
    template<typename This, typename Expr, typename State, typename Data>
    struct result<This(Expr, State, Data)> {
        // types
        typedef typename PrimitiveTransform::template impl< Expr, State, Data >::result_type type;
    };

    // public member functions
    template<typename Expr>
    typename PrimitiveTransform::template impl<Expr &, unspecified, unspecified>::result_type
    operator()(Expr &) const;
    template<typename Expr, typename State>
    typename PrimitiveTransform::template impl<Expr &, State &, unspecified>::result_type
    operator()(Expr &, State &) const;
    template<typename Expr, typename State>
    typename PrimitiveTransform::template impl<Expr &, State const &, unspecified>::result_type
    operator()(Expr &, State const &) const;
    template<typename Expr, typename State, typename Data>
    typename PrimitiveTransform::template impl<Expr &, State &, Data &>::result_type
    operator()(Expr &, State &, Data &) const;
    template<typename Expr, typename State, typename Data>
    typename PrimitiveTransform::template impl<Expr &, State const &, Data &>::result_type
    operator()(Expr &, State const &, Data &) const;
};
```

Description

transform public member functions

1.

```
template<typename Expr>
typename PrimitiveTransform::template impl<Expr &, unspecified, unspecified>::result_type
operator()(Expr & expr) const;
```

Returns: `typename PrimitiveTransform::template impl<Expr &, unspecified, unspecified>()(expr, unspecified, unspecified)`

2.

```
template<typename Expr, typename State>
    typename PrimitiveTransform::template impl<Expr &, State &, unspecified>::result_type
    operator()(Expr & expr, State & state) const;
```

Returns: `typename PrimitiveTransform::template impl<Expr &, State &, unspecified>()(expr, state, unspecified)`

3.

```
template<typename Expr, typename State>
    typename PrimitiveTransform::template impl<Expr &, State const &, unspecified>::result_type
    operator()(Expr & expr, State const & state) const;
```

Returns: `typename PrimitiveTransform::template impl<Expr &, State const &, unspecified>()(expr, state, unspecified)`

4.

```
template<typename Expr, typename State, typename Data>
    typename PrimitiveTransform::template impl<Expr &, State &, Data &>::result_type
    operator()(Expr & expr, State & state, Data & data) const;
```

Returns: `typename PrimitiveTransform::template impl<Expr &, State &, Data &>()(expr, state, data)`

5.

```
template<typename Expr, typename State, typename Data>
    typename PrimitiveTransform::template impl<Expr &, State const &, Data &>::result_type
    operator()(Expr & expr, State const & state, Data & data) const;
```

Returns: `typename PrimitiveTransform::template impl<Expr &, State const &, Data &>()(expr, state, data)`

Struct template result<This(Expr)>

boost::proto::transform::result<This(Expr)>

Synopsis

```
// In header: <boost/proto/transform/impl.hpp>

template<typename This, typename Expr>
struct result<This(Expr)> {
    // types
    typedef typename PrimitiveTransform::template impl< Expr, unspecified, unspecified >::result_type type;
};
```


Struct template result<This(Expr, State)>

boost::proto::transform::result<This(Expr, State)>

Synopsis

```
// In header: <boost/proto/transform/impl.hpp>

template<typename This, typename Expr, typename State>
struct result<This(Expr, State)> {
    // types
    typedef typename PrimitiveTransform::template impl< Expr, State, unspecified >::result_type type;
};
```


Struct template result<This(Expr, State, Data)>

boost::proto::transform::result<This(Expr, State, Data)>

Synopsis

```
// In header: <boost/proto/transform/impl.hpp>

template<typename This, typename Expr, typename State, typename Data>
struct result<This(Expr, State, Data)> {
    // types
    typedef typename PrimitiveTransform::template impl< Expr, State, Data >::result_type type;
};
```


Struct template transform_impl

boost::proto::transform_impl

Synopsis

```
// In header: <boost/proto/transform/impl.hpp>

template<typename Expr, typename State, typename Data>
struct transform_impl {
    // types
    typedef typename boost::remove_reference<Expr const>::type    expr;
    typedef typename boost::add_reference<Expr const>::type       expr_param;
    typedef typename boost::remove_reference<State const>::type   state;
    typedef typename boost::add_reference<State const>::type       state_param;
    typedef typename boost::remove_reference<Data const>::type     data;
    typedef typename boost::add_reference<Data const>::type        data_param;
};
```

Header <boost/proto/transform/lazy.hpp>

Contains definition of the `proto::lazy<>` transform.

```
namespace boost {
    namespace proto {
        template<typename T> struct lazy;
    }
}
```


Struct template lazy

`boost::proto::lazy` — A [PrimitiveTransform](#) that uses `proto::make<>` to build a [CallableTransform](#), and then uses `proto::call<>` to apply it.

Synopsis

```
// In header: <boost/proto/transform/lazy.hpp>

template<typename T>
struct lazy : proto::transform< lazy<T> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl<Expr, State, Data> {
        // types
        typedef see-below result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

`proto::lazy<>` is useful as a higher-order transform, when the transform to be applied depends on the current state of the transformation. The invocation of the `proto::make<>` transform evaluates any nested transforms, and the resulting type is treated as a [CallableTransform](#), which is evaluated with `proto::call<>`.

For the full description of the behavior of the `proto::lazy<>` transform, see the documentation for the nested `proto::lazy::impl<>` class template.

Struct template impl

boost::proto::lazy::impl

Synopsis

```
// In header: <boost/proto/transform/lazy.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl<Expr, State, Data> {
    // types
    typedef see-below result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                          typename impl::state_param,
                          typename impl::data_param) const;
};
```

Description

impl public types

1. typedef *see-below* result_type;

`proto::lazy<T>::impl<Expr, State, Data>::result_type` is calculated as follows:

- If T is of the form $O(A_0, \dots, A_n)$, then let O' be `boost::result_of<proto::make<O>(Expr, State, Data)>::type` and let T' be $O'(A_0, \dots, A_n)$.
- Otherwise, let T' be `boost::result_of<proto::make<T>(Expr, State, Data)>::type`.

The result type is `boost::result_of<proto::call<T'>(Expr, State, Data)>::type`.

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                      typename impl::state_param state,
                      typename impl::data_param data) const;
```

`proto::lazy<T>::impl<Expr, State, Data>::operator()` behaves as follows:

- If T is of the form $O(A_0, \dots, A_n)$, then let O' be `boost::result_of<proto::make<O>(Expr, State, Data)>::type` and let T' be $O'(A_0, \dots, A_n)$.
- Otherwise, let T' be `boost::result_of<proto::make<T>(Expr, State, Data)>::type`.

Returns: `proto::call<T'>()(expr, state, data)`

Header <boost/proto/transform/make.hpp>

Contains definition of the `proto::make<>` and `proto::protect<>` transforms.


```
namespace boost {  
  namespace proto {  
    template<typename T> struct noinvoke;  
    template<typename PrimitiveTransform> struct protect;  
    template<typename T> struct make;  
  }  
}
```


Struct template noinvoke

`boost::proto::noinvoke` — A type annotation in an [ObjectTransform](#) which instructs Proto not to look for a nested `::type` within `T` after type substitution.

Synopsis

```
// In header: <boost/proto/transform/make.hpp>

template<typename T>
struct noinvoke {
};
```

Description

[ObjectTransforms](#) are evaluated by `proto::make<>`, which finds all nested transforms and replaces them with the result of their applications. If any substitutions are performed, the result is first assumed to be a metafunction to be applied; that is, Proto checks to see if the result has a nested `::type` typedef. If it does, that becomes the result. The purpose of `proto::noinvoke<>` is to prevent Proto from looking for a nested `::type` typedef in these situations.

Example:

```
struct Test
: proto::when<
    , proto::noinvoke<
        // This remove_pointer invocation is blocked by noinvoke
        boost::remove_pointer<
            // This add_pointer invocation is *not* blocked by noinvoke
            boost::add_pointer<_>
        >
    >()
>{};

void test_noinvoke()
{
    typedef proto::terminal<int>::type Int;

    BOOST_MPL_ASSERT((
        boost::is_same<
            boost::result_of<Test(Int)>::type
            , boost::remove_pointer<Int *>
        >
    ));

    Int i = {42};
    boost::remove_pointer<Int *> t = Test()(i);
}
```


Struct template protect

`boost::proto::protect` — A [PrimitiveTransform](#) which prevents another [PrimitiveTransform](#) from being applied in an [ObjectTransform](#).

Synopsis

```
// In header: <boost/proto/transform/make.hpp>

template<typename PrimitiveTransform>
struct protect : proto::transform< protect<PrimitiveTransform> > {
    // member classes/structs/unions
    template<typename , typename , typename >
    struct impl {
        // types
        typedef PrimitiveTransform result_type;
    };
};
```

Description

When building higher order transforms with `proto::make<>` or `proto::lazy<>`, you sometimes would like to build types that are parameterized with Proto transforms. In such lambda-style transforms, Proto will unhelpfully find all nested transforms and apply them, even if you don't want them to be applied. Consider the following transform, which will replace the `proto::_` in `Bar<proto::_>()` with `proto::terminal<int>::type`:

```
template<typename T>
struct Bar
{};

struct Foo :
    proto::when<proto::_, Bar<proto::_>() >
{};

proto::terminal<int>::type i = {0};

int main() {
    Foo()(i);
    std::cout << typeid(Foo()(i)).name() << std::endl;
}
```

If you actually wanted to default-construct an object of type `Bar<proto::_>`, you would have to protect the `_` to prevent it from being applied. You can use `proto::protect<>` as follows:

```
// OK: replace anything with Bar<_>()
struct Foo :
    proto::when<proto::_, Bar<proto::protect<proto::_> >() >
{};
```


Struct template impl

boost::proto::protect::impl

Synopsis

```
// In header: <boost/proto/transform/make.hpp>

template<typename , typename , typename >
struct impl {
    // types
    typedef PrimitiveTransform result_type;
};
```


Struct template make

`boost::proto::make` — A [PrimitiveTransform](#) that computes a type by evaluating any nested transforms and then constructs an object of that type.

Synopsis

```
// In header: <boost/proto/transform/make.hpp>

template<typename T>
struct make : proto::transform< make<T> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef see-below result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

The purpose of `proto::make<>` is to annotate a transform as an [ObjectTransform](#) so that `proto::when<>` knows how to apply it.

For the full description of the behavior of the `proto::make<>` transform, see the documentation for the nested `proto::make::impl<>` class template.

Struct template impl

boost::proto::make::impl

Synopsis

```
// In header: <boost/proto/transform/make.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef see-below result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                          typename impl::state_param,
                          typename impl::data_param) const;
};
```

Description

impl public types

1. typedef *see-below* result_type;

`proto::make<T>::impl<Expr, State, Data>::result_type` is computed as follows:

If `T` is an [ObjectTransform](#) of the form `Object(A0, ... An)`, then let `O` be the return type `Object`. Otherwise, let `O` be `T`. The `result_type` typedef is then computed as follows:

- If `O` is a [Transform](#), then let the result type be `boost::result_of<proto::when<_, O>(Expr, State, Data)>::type`. Note that a substitution took place.
- If `O` is a template like `proto::noinvoke<S<X0, ... Xn> >`, then the result type is calculated as follows:
 - For each `i` in `[0, n]`, let `Xi'` be `boost::result_of<proto::make<Xi>(Expr, State, Data)>::type` (which evaluates this procedure recursively). Note that a substitution took place. (In this case, Proto merely assumes that a substitution took place for the sake of compile-time efficiency. There would be no reason to use `proto::noinvoke<>` otherwise.)
 - The result type is `S<X0', ... Xn'> .`
- If `O` is a template like `S<X0, ... Xn>`, then the result type is calculated as follows:
 - For each `i` in `[0, n]`, let `Xi'` be `boost::result_of<proto::make<Xi>(Expr, State, Data)>::type` (which evaluates this procedure recursively). Note whether any substitutions took place during this operation.
 - If any substitutions took place in the above step and `S<X0', ... Xn'>` has a nested type typedef, the result type is `S<X0', ... Xn'>::type .`
 - Otherwise, the result type is `S<X0', ... Xn'> .`
- Otherwise, the result type is `O`, and note that no substitution took place.

Note that `proto::when<>` is implemented in terms of `proto::call<>` and `proto::make<>`, so the above procedure is evaluated recursively.

impl public member functions

```
1. result_type operator()(typename impl::expr_param expr,
                        typename impl::state_param state,
                        typename impl::data_param data) const;
```

`proto::make<T>::impl<Expr, State, Data>::operator()` behaves as follows:

- If T is of the form $O(A_0, \dots, A_n)$, then:
 - If `proto::is_aggregate<result_type>::value` is true, then construct and return an object that as follows:

```
result_type that = {
    proto::when<_, A0>()(expr, state, data),
    ...
    proto::when<_, An>()(expr, state, data)
};
```

- Otherwise, construct and return an object that as follows:

```
result_type that(
    proto::when<_, A0>()(expr, state, data),
    ...
    proto::when<_, An>()(expr, state, data)
);
```

- Otherwise, construct and return an object that as follows:

```
result_type that = result_type();
```

Header `<boost/proto/transform/pass_through.hpp>`

Definition of the `proto::pass_through<>` transform, which is the default transform of all of the expression generator metafunctions such as `proto::unary_plus<>`, `proto::plus<>` and `proto::nary_expr<>`.

```
namespace boost {
    namespace proto {
        template<typename Grammar> struct pass_through;
    }
}
```


Struct template pass_through

`boost::proto::pass_through` — A [PrimitiveTransform](#) that transforms the child expressions of an expression node according to the corresponding children of a Grammar.

Synopsis

```
// In header: <boost/proto/transform/pass_through.hpp>

template<typename Grammar>
struct pass_through : proto::transform< pass_through<Grammar> > {
    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl<Expr, State, Data> {
        // types
        typedef typename proto::result_of::child_c<Grammar, N>::type      GN;           // For each ↓
        N in [0,Expr arity), for exposition only
        typedef typename proto::result_of::child_c<Expr, N>::type        EN;           // For each ↓
        N in [0,Expr arity), for exposition only
        typedef typename boost::result_of<GN(EN,State,Data)>::type        RN;           // For each ↓
        N in [0,Expr arity), for exposition only
        typedef typename Expr::proto_tag                                T;           // For ex↓
        position only
        typedef typename Expr::proto_domain                            D;           // For ex↓
        position only
        typedef typename D::proto_generator                            G;           // For ex↓
        position only
        typedef proto::listN<R0,...RN>                                  A;           // For ex↓
        position only
        typedef proto::expr<T, A>                                        E;           // For ex↓
        position only
        typedef proto::basic_expr<T, A>                                BE;          // For ex↓
        position only
        typedef typename mpl::if_<proto::wants_basic_expr<G>, BE, E>::type expr_type; // For ex↓
        position only
        typedef typename boost::result_of<D(expr_type)>::type            result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

Given a Grammar such as `proto::plus<T0, T1>`, an expression type that matches the grammar such as `proto::plus<E0, E1>::type`, a state `S` and a data `D`, the result of applying the `proto::pass_through<proto::plus<T0, T1> >` transform is:

```
proto::plus<
    boost::result_of<T0(E0, S, D)>::type,
    boost::result_of<T1(E1, S, D)>::type
>::type
```

The above demonstrates how child transforms and child expressions are applied pairwise, and how the results are reassembled into a new expression node with the same tag type as the original.

The explicit use of `proto::pass_through<>` is not usually needed, since the expression generator metafunctions such as `proto::plus<>` have `proto::pass_through<>` as their default transform. So, for instance, these are equivalent:

- `proto::when< proto::plus<X, Y>, proto::pass_through< proto::plus<X, Y> > >`
- `proto::when< proto::plus<X, Y>, proto::plus<X, Y> >`
- `proto::when< proto::plus<X, Y> > // because of proto::when<class X, class Y=X>`
- `proto::plus<X, Y> // because plus<> is both a grammar and a transform`

For example, consider the following transform that promotes all float terminals in an expression to double.

```
// This transform finds all float terminals in an expression and promotes
// them to doubles.
struct Promote :
    proto::or_<
        proto::when<proto::terminal<float>, proto::terminal<double>::type(proto::_value) >,
            // terminal<>'s default transform is a no-op:
            proto::terminal<proto::_>,
            // nary_expr<> has a pass_through<> transform:
            proto::nary_expr<proto::_ , proto::vararg<Promote> >
        >
    {};
```


Struct template impl

boost::proto::pass_through::impl

Synopsis

```
// In header: <boost/proto/transform/pass_through.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl<Expr, State, Data> {
    // types
    typedef typename proto::result_of::child_c<Grammar, N>::type      GN;           // For each ↓
    N in [0,Expr arity), for exposition only
    typedef typename proto::result_of::child_c<Expr, N>::type        EN;           // For each ↓
    N in [0,Expr arity), for exposition only
    typedef typename boost::result_of<GN(EN,State,Data)>::type        RN;           // For each ↓
    N in [0,Expr arity), for exposition only
    typedef typename Expr::proto_tag                                T;           // For expos↓
    ition only
    typedef typename Expr::proto_domain                            D;           // For expos↓
    ition only
    typedef typename D::proto_generator                            G;           // For expos↓
    ition only
    typedef proto::listN<R0,...RN>                                  A;           // For expos↓
    ition only
    typedef proto::expr<T, A>                                       E;           // For expos↓
    ition only
    typedef proto::basic_expr<T, A>                                  BE;          // For expos↓
    ition only
    typedef typename mpl::if_<proto::wants_basic_expr<G>, BE, E>::type expr_type;  // For expos↓
    ition only
    typedef typename boost::result_of<D(expr_type)>::type            result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                           typename impl::state_param,
                           typename impl::data_param) const;
};
```

Description

impl public member functions

1. `result_type operator()(typename impl::expr_param expr,
 typename impl::state_param state,
 typename impl::data_param data) const;`

Requires: `proto::matches<Expr, Grammar>::value` is true.

Returns:

```
D()(expr_type::make(
    G0()(proto::child_c<0>(expr), state, data),
    ...
    GN()(proto::child_c<N>(expr), state, data)
))
```

Header <boost/proto/transform/when.hpp>

Definition of the `proto::when<>` and `proto::otherwise<>` transforms.


```
namespace boost {  
  namespace proto {  
    template<typename Grammar, typename PrimitiveTransform = Grammar>  
      struct when;  
  
    template<typename Grammar, typename Fun> struct when<Grammar, Fun *>;  
    template<typename Grammar, typename R, typename... A>  
      struct when<Grammar, R(A...)>;  
  
    template<typename Fun> struct otherwise;  
  }  
}
```


Struct template when

`boost::proto::when` — A grammar element and a [PrimitiveTransform](#) that associates a transform with the grammar.

Synopsis

```
// In header: <boost/proto/transform/when.hpp>

template<typename Grammar, typename PrimitiveTransform = Grammar>
struct when : PrimitiveTransform {
    // types
    typedef typename Grammar::proto_grammar proto_grammar;
};
```

Description

Use `proto::when<>` to override a grammar's default transform with a custom transform. It is for used when composing larger transforms by associating smaller transforms with individual rules in your grammar, as in the following transform which counts the number of terminals in an expression.

```
// Count the terminals in an expression tree.
// Must be invoked with initial state == mpl::int_<0>().
struct CountLeaves :
    proto::or_<
        proto::when<proto::terminal<proto::_>, mpl::next<proto::_state>()>,
        proto::otherwise<proto::fold<proto::_ , proto::_state, CountLeaves> >
    >
{};
```

In `proto::when<G, T>`, when `T` is a class type it is a [PrimitiveTransform](#) and the following equivalencies hold:

- `boost::result_of<proto::when<G,T>(E,S,V)>::type` is the same as `boost::result_of<T(E,S,V)>::type`.
- `proto::when<G,T>()(e,s,d)` is the same as `T()(e,s,d)`.

Struct template when<Grammar, Fun *>

boost::proto::when<Grammar, Fun *> — A specialization that treats function pointer [Transforms](#) as if they were function type [Transforms](#).

Synopsis

```
// In header: <boost/proto/transform/when.hpp>

template<typename Grammar, typename Fun>
struct when<Grammar, Fun *> : proto::when< Grammar, Fun > {
};
```

Description

This specialization requires that Fun is actually a function type.

This specialization is required for nested transforms such as `proto::when<G, T0(T1(_))>`. In C++, functions that are used as parameters to other functions automatically decay to function pointer types. In other words, the type `T0(T1(_))` is indistinguishable from `T0(T1(*) (_))`. This specialization is required to handle these nested function pointer type transforms properly.

Struct template when<Grammar, R(A...)>

boost::proto::when<Grammar, R(A...)> — A grammar element and a [PrimitiveTransform](#) that associates a transform with the grammar.

Synopsis

```
// In header: <boost/proto/transform/when.hpp>

template<typename Grammar, typename R, typename... A>
struct when<Grammar, R(A...)> : proto::transform< when<Grammar, R(A...)> > {
    // types
    typedef typename Grammar::proto_grammar proto_grammar;

    // member classes/structs/unions
    template<typename Expr, typename State, typename Data>
    struct impl : proto::transform_impl< Expr, State, Data > {
        // types
        typedef proto::call<R(A...)> call_;           // For ex...
position only
        typedef proto::make<R(A...)> make_;           // For ex...
position only
        typedef typename mpl::if_<proto::is_callable<R>, call_, make_>::type which;           // For ex...
position only
        typedef typename boost::result_of<which(Expr, State, Data)>::type result_type;

        // public member functions
        result_type operator()(typename impl::expr_param,
                               typename impl::state_param,
                               typename impl::data_param) const;
    };
};
```

Description

Use `proto::when<>` to override a grammar's default transform with a custom transform. It is for use when composing larger transforms by associating smaller transforms with individual rules in your grammar.

The `when<G, R(A...)>` form accepts either a [CallableTransform](#) or an [ObjectTransform](#) as its second parameter. `proto::when<>` uses `proto::is_callable<R>::value` to distinguish between the two, and uses `proto::call<>` to evaluate [CallableTransforms](#) and `proto::make<>` to evaluate [ObjectTransforms](#).

Struct template impl

boost::proto::when<Grammar, R(A...)>::impl

Synopsis

```
// In header: <boost/proto/transform/when.hpp>

template<typename Expr, typename State, typename Data>
struct impl : proto::transform_impl< Expr, State, Data > {
    // types
    typedef proto::call<R(A...)>                                call_;      // For exposi
ition only
    typedef proto::make<R(A...)>                                make_;      // For exposi
ition only
    typedef typename mpl::if_<proto::is_callable<R>, call_, make_>::type which;  // For exposi
ition only
    typedef typename boost::result_of<which(Expr, State, Data)>::type result_type;

    // public member functions
    result_type operator()(typename impl::expr_param,
                           typename impl::state_param,
                           typename impl::data_param) const;
};
```

Description

impl public member functions

1.

```
result_type operator()(typename impl::expr_param expr,
                       typename impl::state_param state,
                       typename impl::data_param data) const;
```

Evaluate `R(A...)` as a transform either with `proto::call<>` or with `proto::make<>` depending on whether `proto::is_callable<R>::value` is true or false.

Parameters:

<code>data</code>	An arbitrary data
<code>expr</code>	The current expression
<code>state</code>	The current state

Requires: `proto::matches<Expr, Grammar>::value` is true.

Returns: `which()(expr, state, data)`

Struct template otherwise

`boost::proto::otherwise` — Syntactic sugar for `proto::when< proto::_ , Fun >`, for use in grammars to handle all the cases not yet handled.

Synopsis

```
// In header: <boost/proto/transform/when.hpp>

template<typename Fun>
struct otherwise : proto::when< proto::_ , Fun > {
};
```

Description

Use `proto::otherwise<T>` in your grammars as a synonym for `proto::when< proto::_ , Fun >` as in the following transform which counts the number of terminals in an expression.

```
// Count the terminals in an expression tree.
// Must be invoked with initial state == mpl::int_<0>().
struct CountLeaves :
    proto::or_<
        proto::when<proto::terminal<proto::_> , mpl::next<proto::_state>()> ,
        proto::otherwise<proto::fold<proto::_ , proto::_state , CountLeaves> >
    >
{};
```

Header <boost/proto/context/callable.hpp>

Definition of `proto::context::callable_context<>`, an evaluation context for `proto::eval()` that fans out each node and calls the derived context type with the expressions constituents. If the derived context doesn't have an overload that handles this node, fall back to some other context.

```
namespace boost {
    namespace proto {
        namespace context {
            template<typename Expr, typename Context> struct callable_eval;
            template<typename Context,
                    typename DefaultCtx = proto::context::default_context>
                struct callable_context;
        }
    }
}
```


Struct template callable_eval

`boost::proto::context::callable_eval` — A BinaryFunction that accepts a Proto expression and a callable context and calls the context with the expression tag and children as arguments, effectively fanning the expression out.

Synopsis

```
// In header: <boost/proto/context/callable.hpp>

template<typename Expr, typename Context>
struct callable_eval {
    // types
    typedef typename boost::result_of<
        Context(
            typename Expr::proto_tag,
            typename proto::result_of::child_c<0>::type,
            ...
            typename proto::result_of::child_c<N>::type,
        )>::type
        result_type;

    // public member functions
    result_type operator()(Expr &, Context &) const;
};
```

Description

`proto::context::callable_eval<>` requires that Context is a [PolymorphicFunctionObject](#) that can be invoked with Expr's tag and children as expressions, as follows:

```
context(typename Expr::proto_tag(), proto::child_c<0>(expr), ... proto::child_c<N>(expr))
```

callable_eval public member functions

1. `result_type operator()(Expr & expr, Context & context) const;`

Parameters: context The callable evaluation context
 expr The current expression

Returns: context(typename Expr::proto_tag(), proto::child_c<0>(expr), ...
 proto::child_c<N>(expr))

Struct template callable_context

`boost::proto::context::callable_context` — An evaluation context adaptor that makes authoring a context a simple matter of writing function overloads, rather than writing template specializations.

Synopsis

```
// In header: <boost/proto/context/callable.hpp>

template<typename Context,
        typename DefaultCtx = proto::context::default_context>
struct callable_context {
    // member classes/structs/unions
    template<typename Expr, typename ThisContext = Context>
    struct eval : see-below {
    };
};
```

Description

`proto::callable_context<>` is a base class that implements the context protocol by passing fanned-out expression nodes to the derived context, making it easy to customize the handling of expression types by writing function overloads. Only those expression types needing special handling require explicit handling. All others are dispatched to a user-specified default context, `DefaultCtx`.

`proto::callable_context<>` is defined simply as:

```
template<typename Context, typename DefaultCtx = default_context>
struct callable_context {
    template<typename Expr, typename ThisContext = Context>
    struct eval :
        mpl::if_<
            is_expr_handled_<Expr, Context>, // For exposition
            proto::context::callable_eval<Expr, ThisContext>,
            typename DefaultCtx::template eval<Expr, Context>
        >::type
    {
    };
};
```

The Boolean metafunction `is_expr_handled_<>` uses metaprogramming tricks to determine whether `Context` has an overloaded function call operator that accepts the fanned-out constituents of an expression of type `Expr`. If so, the handling of the expression is dispatched to `proto::context::callable_eval<>`. If not, it is dispatched to the user-specified `DefaultCtx`.

Example:


```
// An evaluation context that increments all
// integer terminals in-place.
struct increment_ints :
    proto::context::callable_context<
        increment_ints const           // derived context
        proto::context::null_context const // fall-back context
    >
{
    typedef void result_type;

    // Handle int terminals here:
    void operator()(proto::tag::terminal, int &i) const
    {
        ++i;
    }
};
```

With `increment_ints`, we can do the following:

```
proto::literal<int> i = 0, j = 10;
proto::eval( i - j * 3.14, increment_ints() );

assert( i.get() == 1 && j.get() == 11 );
```


Struct template eval

boost::proto::context::callable_context::eval

Synopsis

```
// In header: <boost/proto/context/callable.hpp>

template<typename Expr, typename ThisContext = Context>
struct eval : see-below {
};
```

Description

A BinaryFunction that accepts an Expr and a Context, and either fans out the expression and passes it to the context, or else hands off the expression to DefaultCtx.

If Context is a [PolymorphicFunctionObject](#) such that it can be invoked with the tag and children of Expr, as `ctx(typename Expr::proto_tag(), child_c<0>(expr),... child_c<N>(expr))`, then `eval<Expr, ThisContext>` inherits from `proto::context::callable_eval<Expr, ThisContext>`. Otherwise, `eval<Expr, ThisContext>` inherits from `DefaultCtx::eval<Expr, Context>`.

Header <[boost/proto/context/default.hpp](#)>

```
namespace boost {
  namespace proto {
    namespace context {
      template<typename Expr, typename Context> struct default_eval;
      struct default_context;
    }
  }
}
```


Struct template `default_eval`

`boost::proto::context::default_eval` — A BinaryFunction that accepts a Proto expression and a context, evaluates each child expression with the context, and combines the result using the standard C++ meaning for the operator represented by the current expression node.

Synopsis

```
// In header: <boost/proto/context/default.hpp>

template<typename Expr, typename Context>
struct default_eval {
    // types
    typedef typename Expr::tag_type Tag;           // For exposition only
    typedef see-below result_type;

    // public member functions
    result_type operator()(Expr &, Context &) const;
    static Expr & s_expr; // For exposition only
    static Context & s_context; // For exposition only
};
```

Description

Let `OP` be the C++ operator corresponding to `Expr::proto_tag`. (For example, if `Tag` is `proto::tag::plus`, let `OP` be `+`.)

The behavior of this class is specified in terms of the C++0x `decltype` keyword. In systems where this keyword is not available, Proto uses the Boost.Typeof library to approximate the behavior.

`default_eval` public types

1. `typedef see-below result_type;`

- If `Tag` corresponds to a unary prefix operator, then the result type is

```
decltype(
    OP proto::eval(proto::child(s_expr), s_context)
)
```

- If `Tag` corresponds to a unary postfix operator, then the result type is

```
decltype(
    proto::eval(proto::child(s_expr), s_context) OP
)
```

- If `Tag` corresponds to a binary infix operator, then the result type is

```
decltype(
    proto::eval(proto::left(s_expr), s_context) OP
    proto::eval(proto::right(s_expr), s_context)
)
```

- If `Tag` is `proto::tag::subscript`, then the result type is


```
decltype(
    proto::eval(proto::left(s_expr), s_context) [
        proto::eval(proto::right(s_expr), s_context) ]
)
```

- If Tag is `proto::tag::if_else_`, then the result type is

```
decltype(
    proto::eval(proto::child_c<0>(s_expr), s_context) ?
    proto::eval(proto::child_c<1>(s_expr), s_context) :
    proto::eval(proto::child_c<2>(s_expr), s_context)
)
```

- If Tag is `proto::tag::function`, then the result type is

```
decltype(
    proto::eval(proto::child_c<0>(s_expr), s_context) (
        proto::eval(proto::child_c<1>(s_expr), s_context),
        ...
        proto::eval(proto::child_c<N>(s_expr), s_context) )
)
```

default_eval public member functions

1. `result_type operator()(Expr & expr, Context & context) const;`

- If Tag corresponds to a unary prefix operator, then return

```
OP proto::eval(proto::child(expr), context)
```

- If Tag corresponds to a unary postfix operator, then return

```
proto::eval(proto::child(expr), context) OP
```

- If Tag corresponds to a binary infix operator, then return

```
proto::eval(proto::left(expr), context) OP
proto::eval(proto::right(expr), context)
```

- If Tag is `proto::tag::subscript`, then return

```
proto::eval(proto::left(expr), context) [
    proto::eval(proto::right(expr), context) ]
```

- If Tag is `proto::tag::if_else_`, then return

```
proto::eval(proto::child_c<0>(expr), context) ?
proto::eval(proto::child_c<1>(expr), context) :
proto::eval(proto::child_c<2>(expr), context)
```


- If Tag is `proto::tag::function` , then return

```
proto::eval(proto::child_c<0>(expr), context) (  
proto::eval(proto::child_c<1>(expr), context),  
...  
proto::eval(proto::child_c<N>(expr), context) )
```

Parameters:	context	The evaluation context
	expr	The current expression

Struct default_context

boost::proto::context::default_context — An evaluation context that gives the operators their normal C++ semantics.

Synopsis

```
// In header: <boost/proto/context/default.hpp>

struct default_context {
    // member classes/structs/unions
    template<typename Expr, typename ThisContext = default_context const>
    struct eval : proto::context::default_eval< Expr, ThisContext > {
    };
};
```

Description

An evaluation context that gives the operators their normal C++ semantics.

Struct template eval

boost::proto::context::default_context::eval

Synopsis

```
// In header: <boost/proto/context/default.hpp>

template<typename Expr, typename ThisContext = default_context const>
struct eval : proto::context::default_eval< Expr, ThisContext > {
};
```

Header <boost/proto/context/null.hpp>

Definition of `proto::context::null_context<>`, an evaluation context for `proto::eval()` that simply evaluates each child expression, doesn't combine the results at all, and returns void.

```
namespace boost {
  namespace proto {
    namespace context {
      template<typename Expr, typename Context> struct null_eval;
      struct null_context;
    }
  }
}
```


Struct template null_eval

boost::proto::context::null_eval

Synopsis

```
// In header: <boost/proto/context/null.hpp>

template<typename Expr, typename Context>
struct null_eval {
    // types
    typedef void result_type;

    // public member functions
    void operator()(Expr &, Context &) const;
};
```

Description

null_eval public member functions

1. `void operator()(Expr & expr, Context & context) const;`

For N in $[0, \text{Expr arity})$, evaluate:

```
proto::eval(proto::child_c<N>(expr), context)
```


Struct `null_context`

`boost::proto::context::null_context` — An evaluation context for `proto::eval()` that simply evaluates each child expression, doesn't combine the results at all, and returns void.

Synopsis

```
// In header: <boost/proto/context/null.hpp>

struct null_context {
    // member classes/structs/unions
    template<typename Expr, typename ThisContext = null_context const>
    struct eval : proto::context::null_eval< Expr, ThisContext > {
    };
};
```

Description

Struct template eval

boost::proto::context::null_context::eval

Synopsis

```
// In header: <boost/proto/context/null.hpp>

template<typename Expr, typename ThisContext = null_context const>
struct eval : proto::context::null_eval< Expr, ThisContext > {
};
```


Concept CallableTransform

CallableTransform

Description

A CallableTransform is a function type or a function pointer type where the return type Fn is a PolymorphicFunctionObject and the arguments are Transforms. `is_callable< Fn >::value` must be true. The CallableTransform, when applied, has the effect of invoking the polymorphic function object Fn, passing as arguments the result(s) of applying transform(s) Tn.

Associated types

- **result_type**

```
boost::result_of<Fn(Transform<Tn, Expr, State, Data>::result_type...)>::type
```

The result of applying the CallableTransform.

Notation

Fn A type playing the role of polymorphic-function-object-type in the [CallableTransform](#) concept.

Tn A type playing the role of transform-type in the [CallableTransform](#) concept.

Expr A type playing the role of expression-type in the [CallableTransform](#) concept.

State A type playing the role of state-type in the [CallableTransform](#) concept.

Data A type playing the role of data-type in the [CallableTransform](#) concept.

fn Object of type Fn

expr Object of type Expr

state Object of type State

data Object of type Data

Valid expressions

Name	Expression	Type	Semantics
Apply Transform	<code>when< _, Fn(Tn...)>()(expr, state, data)</code>	<code>result_type</code>	Applies the transform.

Models

- `boost::proto::_child(boost::proto::_left)`

Concept Domain

Domain

Description

A Domain creates an association between expressions and a so-called generator, which is a function that maps an expression in the default domain to an equivalent expression in this Domain. It also associates an expression with a grammar, to which all expressions within this Domain must conform.

Associated types

- **proto_grammar**

```
Domain::proto_grammar
```

The grammar to which every expression in this Domain must conform.

- **proto_generator**

```
Domain::proto_generator
```

A Unary Polymorphic Function that accepts expressions in the default domain and emits expressions in this Domain.

- **proto_super_domain**

```
Domain::proto_super_domain
```

The Domain that is a super-domain of this domain, if any such domain exists. If not, it is some unspecified type.

- **result_type**

```
boost::result_of<Domain(Expr)>::type
```

The type of the result of applying `proto_generator` to the specified expression type. The result is required to model [Expr](#). The domain type associated with `result_type` (`result_type::proto_domain`) is required to be the same type as this Domain.

- **as_expr_result_type**

```
Domain::as_expr<Object>::result_type
```

The result of converting some type to a Proto expression type in this domain. This is used, for instance, when calculating the type of a variable to hold a Proto expression. `as_expr_result_type` models [Expr](#).

- **as_child_result_type**

```
Domain::as_child<Object>::result_type
```

The result of converting some type to a Proto expression type in this domain. This is used, for instance, to compute the type of an object suitable for storage as a child in an expression tree. `as_child_result_type` models [Expr](#).

Notation

Domain A type playing the role of domain-type in the [Domain](#) concept.

- Expr A type playing the role of expression-type in the [Domain](#) concept.
- Object A type playing the role of object-type in the [Domain](#) concept.
- d Object of type Domain
- e Object of type Expr
- o Object of type Object

Valid expressions

Name	Expression	Type	Semantics
Apply Generator	d(e)	result_type	The result of applying <code>proto_generator</code> to the specified expression.
As Expression	Domain::as_expr<Object>(o)	as_expr_result_type	The result of converting some object to a Proto expression in this domain. It returns a Proto expression object that is suitable for storage in a variable. It should return a new object, which may be a copy of the object passed in.
As Child	Domain::as_child<Object>(o)	as_child_result_type	The result of converting some object to a Proto expression in this domain. It returns an object suitable for storage as a child in an expression tree, which may simply be a reference to the object passed in.

Models

- `boost::proto::default_domain`

Concept Expr

Expr

Description

An Expr represents a tagged node in an expression tree. The children of the Expr must themselves satisfy the Expr concept. The Expr has an arity representing the number of children. If the number of children is zero, the Expr also has a value. An Expr also has an associated [Domain](#).

Associated types

- **proto_tag**

```
Expr::proto_tag
```

The tag type of the Expr.

- **proto_args**

```
Expr::proto_args
```

A typelist representing either the types of the child nodes, or, if the arity of the Expr is 0, of the value of the terminal.

- **proto_arity**

```
Expr::proto_arity
```

The arity (number of child nodes) of the Expr. proto_arity is an MPL Integral Constant.

- **proto_grammar**

```
Expr::proto_grammar
```

A typedef for an instantiation of `proto::basic_expr<>` that is equivalent to Expr. Expression types are equivalent if they have the same proto_tag, proto_args, and proto_arity.

- **proto_base_expr**

```
Expr::proto_base_expr
```

A typedef for an instantiation of `proto::expr<>` or `proto::basic_expr<>` that is equivalent to Expr. Expression types are equivalent if they have the same proto_tag, proto_args, and proto_arity.

- **proto_derived_expr**

```
Expr::proto_derived_expr
```

A typedef for Expr.

- **proto_domain**

```
Expr::proto_domain
```


The Domain of the Expr. `proto_domain` models [Domain](#).

- **proto_childN**

`Expr::proto_childN`

The type of the Nth child of Expr. Requires `0 == N::value || N::value < proto_arity::value`

Notation

Expr A type playing the role of expression-type in the [Expr](#) concept.

Tag A type playing the role of tag-type in the [Expr](#) concept.

Domain A type playing the role of domain-type in the [Expr](#) concept.

N A type playing the role of mpl-integral-constant-type in the [Expr](#) concept.

e Object of type Expr

Valid expressions

Name	Expression	Type	Semantics
Get N-th Child	<code>boost::proto::child< N >(e)</code>	<code>proto_childN</code>	Extracts the Nth child from this Expr. Requires <code>N::value < proto_arity::value</code> .
Get Terminal Value	<code>boost::proto::value(e)</code>	<code>proto_child0</code>	Extracts the value from a terminal Expr. Requires <code>0 == proto_arity::value</code> .
Get Base	<code>e.proto_base()</code>	<code>proto_base_expr</code>	Returns an object of type <code>proto::expr<></code> or <code>proto::basic_expr<></code> that is equivalent to e.

Models

- `boost::proto::literal< int >`

Concept ObjectTransform

ObjectTransform

Description

An ObjectTransform is a function type or a function pointer type where the return type Obj is a an object type and the arguments are Transforms. `is_callable< Obj >::value` must be false. The ObjectTransform, when applied, has the effect of constructing an object of type Obj' (see below), passing as construction parameters the result(s) of applying transform(s) Tn.

The type Obj may be a template specialization representing a compile-time lambda expression. For instance, if Obj is `std::pair< proto::_value, int >`, the result type of the ObjectTransform is computed by replacing the type `proto::_value` with the result of applying the `proto::_value` transform. For given types Obj, Expr, State and Data, we can say that the type Obj' represents the type Obj after all nested transforms have been replaced with the results of applying the transforms with Expr, State and Data as transform arguments.

If the type Obj is not a template specialization representing a compile-time lambda expression, then the result type Obj' is the same as Obj.

Notation

Obj	A type playing the role of object-type in the ObjectTransform concept.
Tn	A type playing the role of transform-type in the ObjectTransform concept.
Expr	A type playing the role of expression-type in the ObjectTransform concept.
State	A type playing the role of state-type in the ObjectTransform concept.
Data	A type playing the role of data-type in the ObjectTransform concept.
expr	Object of type Expr
state	Object of type State
data	Object of type Data

Valid expressions

Name	Expression	Type	Semantics
Apply Transform	<code>when< _, Obj(Tn...)>()(expr, state, data)</code>	Obj'	Applies the transform.

Models

- `std::pair< boost::proto::_value, int >(boost::proto::_value, int())`

Concept PolymorphicFunctionObject

PolymorphicFunctionObject

Description

A type that can be called and that follows the TR1 ResultOf protocol for return type calculation.

Associated types

- **result_type**

```
result_of<Fn(A0,...An)>::type
```

The result of calling the Polymorphic Function Object.

Notation

Fn A type playing the role of polymorphic-function-object-type in the [PolymorphicFunctionObject](#) concept.

fn Object of type Fn

a0,...an Object of type A0,...An

Valid expressions

Name	Expression	Type	Semantics
Function Call	fn(a0,...an)	result_type	Calls the function object.

Models

- `std::plus<int>`

Concept PrimitiveTransform

PrimitiveTransform

Description

A PrimitiveTransform is a class type that has a nested class template called `impl<>` that takes three template parameters representing an expression type, a state type and a data type. Specializations of the nested `impl` template are ternary monomorphic function objects that accept expression, state, and data parameters. A PrimitiveTransform is also a [PolymorphicFunctionObject](#) implemented in terms of the nested `impl<>` template.

Associated types

- **result_type**

```
typename Fn::template impl<Expr, State, Data>::result_type
```

The return type of the overloaded function call operator.

Notation

Fn A type playing the role of primitive-transform-type in the [PrimitiveTransform](#) concept.

Expr A type playing the role of expression-type in the [PrimitiveTransform](#) concept.

State A type playing the role of state-type in the [PrimitiveTransform](#) concept.

Data A type playing the role of data-type in the [PrimitiveTransform](#) concept.

fn Object of type Fn

expr Object of type Expr

state Object of type State

data Object of type Data

Valid expressions

Name	Expression	Type	Semantics
Polymorphic Function Call 1	<code>fn(expr)</code>	<code>result_type</code>	Applies the transform.
Polymorphic Function Call 2	<code>fn(expr, state)</code>	<code>result_type</code>	Applies the transform.
Polymorphic Function Call 3	<code>fn(expr, state, data)</code>	<code>result_type</code>	Applies the transform.
Monomorphic Function Call	<code>typename Fn::template impl< Expr, State, Data >()(expr, state, data)</code>	<code>result_type</code>	Applies the transform.

Models

- `boost::proto::_child_c<0>`

Concept Transform

Transform

Description

A Transform is a PrimitiveTransform, a CallableTransform or an ObjectTransform.

Associated types

- **result_type**

```
boost::result_of<when< _, Tn >(Expr, State, Data)>::type
```

The result of applying the Transform.

Notation

Tn A type playing the role of transform-type in the [Transform](#) concept.

Expr A type playing the role of expression-type in the [Transform](#) concept.

State A type playing the role of state-type in the [Transform](#) concept.

Data A type playing the role of data-type in the [Transform](#) concept.

expr Object of type Expr

state Object of type State

data Object of type Data

Valid expressions

Name	Expression	Type	Semantics
Apply Transform	when< _, Tn >(expr, state, data)	result_type	Applies the transform.

Models

- boost::proto::_child(boost::proto::_left)

Appendices

Appendix A: Release Notes

Boost 1.44

Behavior Change: proto::and_<>

In Boost 1.44, the behavior of [proto::and_<>](#) as a transform changed. Previously, it only applied the transform associated with the last grammar in the set. Now, it applies all the transforms but only returns the result of the last. That makes it behave like C++'s comma operator. For example, a grammar such as:


```
proto::and_< G0, G1, G2 >
```

when evaluated with an expression `e` now behaves like this:

```
(G0()(e), G1()(e), G2()(e))
```

Behavior Change: `proto::as_expr()` and `proto::as_child()`

The functions `proto::as_expr()` and `proto::as_child()` are used to guarantee that an object is a Proto expression by turning it into one if it is not already, using an optionally specified domain. In previous releases, when these functions were passed a Proto expression in a domain different to the one specified, they would apply the specified domain's generator, resulting in a twice-wrapped expression. This behavior was surprising to some users.

The new behavior of these two functions is to always leave Proto expressions alone, regardless of the expressions' domains.

Behavior Change: `proto::(pod_)generator<>` and `proto::basic_expr<>`

Users familiar with Proto's extension mechanism have probably used either `proto::generator<>` or `proto::pod_generator<>` with a wrapper template when defining their domain. In the past, Proto would instantiate your wrapper template with instances of `proto::expr<>`. In Boost 1.44, Proto now instantiates your wrapper template with instances of a new type: `proto::basic_expr<>`.

For instance:

```
// An expression wrapper
template<class Expr>
struct my_expr_wrapper;

// A domain
struct my_domain
: proto::domain< proto::generator< my_expr_wrapper > >
{
};

template<class Expr>
struct my_expr_wrapper
: proto::extends<Expr, my_expr_wrapper<Expr>, my_domain>
{
    // Before 1.44, Expr was an instance of proto::expr<>
    // In 1.44, Expr is an instance of proto::basic_expr<>
};
```

The motivation for this change was to improve compile times. `proto::expr<>` is an expensive type to instantiate because it defines a host of member functions. When defining your own expression wrapper, the instance of `proto::expr<>` sits as a hidden data member function in your wrapper and the members of `proto::expr<>` go unused. Therefore, the cost of those member functions is wasted. In contrast, `proto::basic_expr<>` is a very lightweight type with no member functions at all.

The vast majority of programs should recompile without any source changes. However, if somewhere you are assuming that you will be given instances specifically of `proto::expr<>`, your code will break.

New Feature: Sub-domains

In Boost 1.44, Proto introduces an important new feature called "sub-domains". This gives you a way to specify that one domain is compatible with another such that expressions in one domain can be freely mixed with expressions in another. You can define one domain to be the sub-domain of another by using the third template parameter of `proto::domain<>`.

For instance:


```
// Not shown: define some expression
// generators genA and genB

struct A
: proto::domain< genA, proto::_ >
{};

// Define a domain B that is the sub-domain
// of domain A.
struct B
: proto::domain< genB, proto::_, A >
{};
```

Expressions in domains A and B can have different wrappers (hence, different interfaces), but they can be combined into larger expressions. Without a sub-domain relationship, this would have been an error. The domain of the resulting expression in this case would be A.

The complete description of sub-domains can be found in the reference sections for [proto::domain<>](#) and [proto::deduce_domain](#).

New Feature: Domain-specific `as_expr()` and `as_child()`

Proto has always allowed users to customize expressions post-hoc by specifying a Generator when defining their domain. But it has never allowed users to control how Proto assembles sub-expressions in the first place. As of Boost 1.44, users now have this power.

Users defining their own domain can now specify how [proto::as_expr\(\)](#) and [proto::as_child\(\)](#) work in their domain. They can do this easily by defining nested class templates named `as_expr` and/or `as_child` within their domain class.

For example:

```
struct my_domain
: proto::domain< my_generator >
{
    typedef
        proto::domain< my_generator >
        base_domain;

    // For my_domain, as_child does the same as
    // what as_expr does by default.
    template<class T>
    struct as_child
    : base_domain::as_expr<T>
    {}
};
```

In the above example, `my_domain::as_child<>` simply defers to `proto::domain::as_expr<>`. This has the nice effect of causing all terminals to be captured by value instead of by reference, and to likewise store child expressions by value. The result is that expressions in `my_domain` are safe to store in `auto` variables because they will not have dangling references to intermediate temporary expressions. (Naturally, it also means that expression construction has extra runtime overhead of copying that the compiler may or may not be able to optimize away.)

Boost 1.43

In Boost 1.43, the recommended usage of [proto::extends<>](#) changed slightly. The new usage looks like this:


```
// my_expr is an expression extension of the Expr parameter
template<typename Expr>
struct my_expr
{
    : proto::extends<Expr, my_expr<Expr>, my_domain>
    {
        my_expr(Expr const &expr = Expr())
        : proto::extends<Expr, my_expr, my_domain>(expr)
        {}

        // NEW: use the following macro to bring
        // proto::extends::operator= into scope.
        BOOST_PROTO_EXTENDS_USING_ASSIGN(my_expr)
    };
};
```

The new thing is the use of the `BOOST_PROTO_EXTENDS_USING_ASSIGN()` macro. To allow assignment operators to build expression trees, `proto::extends<>` overloads the assignment operator. However, for the `my_expr` template, the compiler generates a default copy assignment operator that hides the ones in `proto::extends<>`. This is often not desired (although it depends on the syntax you want to allow).

Previously, the recommended usage was to do this:

```
// my_expr is an expression extension of the Expr parameter
template<typename Expr>
struct my_expr
{
    : proto::extends<Expr, my_expr<Expr>, my_domain>
    {
        my_expr(Expr const &expr = Expr())
        : proto::extends<Expr, my_expr, my_domain>(expr)
        {}

        // OLD: don't do it like this anymore.
        using proto::extends<Expr, my_expr, my_domain>::operator=;
    };
};
```

While this works in the majority of cases, it still doesn't suppress the implicit generation of the default assignment operator. As a result, expressions of the form `a = b` could either build an expression template or do a copy assignment depending on whether the types of `a` and `b` happen to be the same. That can lead to subtle bugs, so the behavior was changed.

The `BOOST_PROTO_EXTENDS_USING_ASSIGN()` brings into scope the assignment operators defined in `proto::extends<>` as well as suppresses the generation of the copy assignment operator.

Also note that the `proto::literal<>` class template, which uses `proto::extends<>`, has been changed to use `BOOST_PROTO_EXTENDS_USING_ASSIGN()`. The implications are highlighted in the sample code below:

```
proto::literal<int> a(1), b(2); // two non-const proto literals
proto::literal<int> const c(3); // a const proto literal

a = b; // No-op. Builds an expression tree and discards it.
      // Same behavior in 1.42 and 1.43.

a = c; // CHANGE! In 1.42, this performed copy assignment, causing
      // a's value to change to 3. In 1.43, the behavior is now
      // the same as above: build and discard an expression tree.
```

Appendix B: History

August ??, 2010

Boost 1.44: Proto gets sub-domains and per-domain control of `proto::as_expr()` and `proto::as_child()` to meet the needs of Phoenix3.

August 11, 2008	Proto v4 is merged to Boost trunk with more powerful transform protocol.
April 7, 2008	Proto is accepted into Boost.
March 1, 2008	Proto's Boost review begins.
January 11, 2008	Boost.Proto v3 brings separation of grammars and transforms and a "round" lambda syntax for defining transforms in-place.
April 15, 2007	Boost.Xpressive is ported from Proto compilers to Proto transforms. Support for old Proto compilers is dropped.
April 4, 2007	Preliminary submission of Proto to Boost.
December 11, 2006	The idea for transforms that decorate grammar rules is born in a private email discussion with Joel de Guzman and Hartmut Kaiser. The first transforms are committed to CVS 5 days later on December 16.
November 1, 2006	The idea for <code>proto::matches<></code> and the whole grammar facility is hatched during a discussion with Hartmut Kaiser on the spirit-devel list. The first version of <code>proto::matches<></code> is checked into CVS 3 days later. Message is here .
October 28, 2006	Proto is reborn, this time with a uniform expression types that are POD. Announcement is here .
April 20, 2005	Proto is born as a major refactorization of Boost.Xpressive's meta-programming. Proto offers expression types, operator overloads and "compilers", an early formulation of what later became transforms. Announcement is here .

Appendix C: Rationale

Static Initialization

Proto expression types are PODs (Plain Old Data), and do not have constructors. They are brace-initialized, as follows:

```
terminal<int>::type const _i = {1};
```

The reason is so that expression objects like `_i` above can be *statically initialized*. Why is static initialization important? The terminals of many domain- specific embedded languages are likely to be global const objects, like `_1` and `_2` from the Boost Lambda Library. Were these object to require run-time initialization, it might be possible to use these objects before they are initialized. That would be bad. Statically initialized objects cannot be misused that way.

Why Not Reuse MPL, Fusion, et cetera?

Anyone who has peeked at Proto's source code has probably wondered, "Why all the dirty preprocessor gunk? Couldn't this have been all implemented cleanly on top of libraries like MPL and Fusion?" The answer is that Proto could have been implemented this way, and in fact was at one point. The problem is that template metaprogramming (TMP) makes for longer compile times. As a foundation upon which other TMP-heavy libraries will be built, Proto itself should be as lightweight as possible. That is achieved by preferring preprocessor metaprogramming to template metaprogramming. Expanding a macro is far more efficient than instantiating a template. In some cases, the "clean" version takes 10x longer to compile than the "dirty" version.

The "clean and slow" version of Proto can still be found at <http://svn.boost.org/svn/boost/branches/proto/v3>. Anyone who is interested can download it and verify that it is, in fact, unusably slow to compile. Note that this branch's development was abandoned, and it does not conform exactly with Proto's current interface.

Appendix D: Implementation Notes

Quick-n-Dirty Type Categorization

Much has already been written about dispatching on type traits using SFINAE (Substitution Failure Is Not An Error) techniques in C++. There is a Boost library, `Boost.Enable_if`, to make the technique idiomatic. Proto dispatches on type traits extensively, but it doesn't use `enable_if<>` very often. Rather, it dispatches based on the presence or absence of nested types, often typedefs for void.

Consider the implementation of `is_expr<>`. It could have been written as something like this:

```
template<typename T>
struct is_expr
    : is_base_and_derived<proto::some_expr_base, T>
{
};
```

Rather, it is implemented as this:

```
template<typename T, typename Void = void>
struct is_expr
    : mpl::false_
{
};

template<typename T>
struct is_expr<T, typename T::proto_is_expr_>
    : mpl::true_
{
};
```

This relies on the fact that the specialization will be preferred if `T` has a nested `proto_is_expr_` that is a typedef for `void`. All Proto expression types have such a nested typedef.

Why does Proto do it this way? The reason is because, after running extensive benchmarks while trying to improve compile times, I have found that this approach compiles faster. It requires exactly one template instantiation. The other approach requires at least 2: `is_expr<>` and `is_base_and_derived<>`, plus whatever templates `is_base_and_derived<>` may instantiate.

Detecting the Arity of Function Objects

In several places, Proto needs to know whether or not a function object `Fun` can be called with certain parameters and take a fallback action if not. This happens in `proto::callable_context<>` and in the `proto::call<>` transform. How does Proto know? It involves some tricky metaprogramming. Here's how.

Another way of framing the question is by trying to implement the following `can_be_called<>` Boolean metafunction, which checks to see if a function object `Fun` can be called with parameters of type `A` and `B`:

```
template<typename Fun, typename A, typename B>
struct can_be_called;
```

First, we define the following `dont_care` struct, which has an implicit conversion from anything. And not just any implicit conversion; it has an ellipsis conversion, which is the worst possible conversion for the purposes of overload resolution:

```
struct dont_care
{
    dont_care(...);
};
```

We also need some private type known only to us with an overloaded comma operator (!), and some functions that detect the presence of this type and return types with different sizes, as follows:


```

struct private_type
{
    private_type const &operator,(int) const;
};

typedef char yes_type;      // sizeof(yes_type) == 1
typedef char (&no_type)[2]; // sizeof(no_type) == 2

template<typename T>
no_type is_private_type(T const &);

yes_type is_private_type(private_type const &);

```

Next, we implement a binary function object wrapper with a very strange conversion operator, whose meaning will become clear later.

```

template<typename Fun>
struct funwrap2 : Fun
{
    funwrap2();
    typedef private_type const &(*pointer_to_function)(dont_care, dont_care);
    operator pointer_to_function() const;
};

```

With all of these bits and pieces, we can implement `can_be_called<>` as follows:

```

template<typename Fun, typename A, typename B>
struct can_be_called
{
    static funwrap2<Fun> &fun;
    static A &a;
    static B &b;

    static bool const value = (
        sizeof(no_type) == sizeof(is_private_type( (fun(a,b), 0) ))
    );

    typedef mpl::bool_<value> type;
};

```

The idea is to make it so that `fun(a,b)` will always compile by adding our own binary function overload, but doing it in such a way that we can detect whether our overload was selected or not. And we rig it so that our overload is selected if there is really no better option. What follows is a description of how `can_be_called<>` works.

We wrap `Fun` in a type that has an implicit conversion to a pointer to a binary function. An object `fun` of class type can be invoked as `fun(a, b)` if it has such a conversion operator, but since it involves a user-defined conversion operator, it is less preferred than an overloaded `operator()`, which requires no such conversion.

The function pointer can accept any two arguments by virtue of the `dont_care` type. The conversion sequence for each argument is guaranteed to be the worst possible conversion sequence: an implicit conversion through an ellipsis, and a user-defined conversion to `dont_care`. In total, it means that `funwrap2<Fun>()(a, b)` will always compile, but it will select our overload only if there really is no better option.

If there is a better option --- for example if `Fun` has an overloaded function call operator such as `void operator()(A a, B b)` --- then `fun(a, b)` will resolve to that one instead. The question now is how to detect which function got picked by overload resolution.

Notice how `fun(a, b)` appears in `can_be_called<>: (fun(a, b), 0)`. Why do we use the comma operator there? The reason is because we are using this expression as the argument to a function. If the return type of `fun(a, b)` is `void`, it cannot legally be used as an argument to a function. The comma operator sidesteps the issue.

This should also make plain the purpose of the overloaded comma operator in `private_type`. The return type of the pointer to function is `private_type`. If overload resolution selects our overload, then the type of `(fun(a, b), 0)` is `private_type`. Otherwise, it is `int`. That fact is used to dispatch to either overload of `is_private_type()`, which encodes its answer in the size of its return type.

That's how it works with binary functions. Now repeat the above process for functions up to some predefined function arity, and you're done.

Appendix E: Acknowledgements

I'd like to thank Joel de Guzman and Hartmut Kaiser for being willing to take a chance on using Proto for their work on Spirit-2 and Karma when Proto was little more than a vision. Their requirements and feedback have been indispensable.

Thanks to Daniel James for providing a patch to remove the dependence on deprecated configuration macros for C++0x features.

Thanks to Dave Abrahams for an especially detailed review, and for making a VM with msvc-7.1 available so I could track down portability issues on that compiler.

Many thanks to Daniel Wallin who first implemented the code used to find the common domain among a set, accounting for super- and sub-domains. Thanks also to Jeremiah Willcock, John Bytheway and Krishna Achuthan who offered alternate solutions to this tricky programming problem.

Thanks also to the developers of [PETE](#). I found many good ideas there.