Domain-Specific Embedded Languages with Boost.Proto

Or, "How I Learned to Stop Worrying and Love Expression Templates"

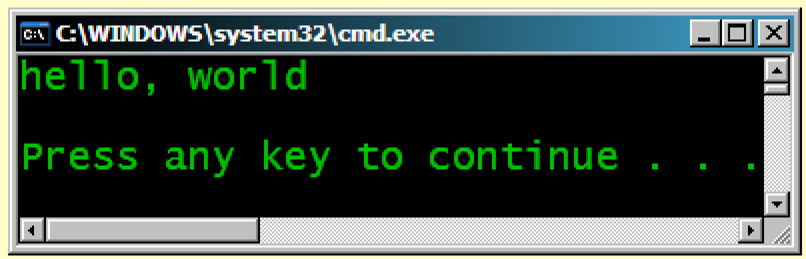
Talk Overview

Goal: Designing declarative, domain-specific libraries using Boost.Proto.

1. Domain Specific Embedded Languages
2. DSEL Design with Boost.Proto
   1. Expression construction
   2. Expression evaluation
   3. Expression introspection
   4. Expression transformation
3. Extending Boost.Proto

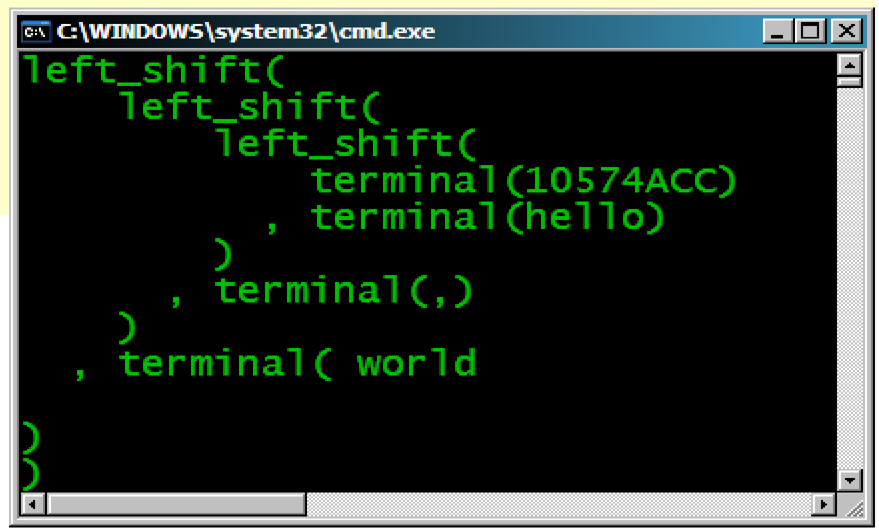
Hello, Boost.Proto!

```
// #includes ...
using namespace boost::proto;
terminal< std::ostream & >::type cout_ = { std::cout };

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

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



Hello, Boost.Proto!

```
// #includes ...
using namespace boost::proto;
terminal< std::ostream & >::type cout_ = { std::cout };

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



Domain-Specific Embedded Languages

Or, "Operator Abuse for Fun and Profit!"

DSEL Example: Boost.Spirit

- Parser Generator
  - similar in purpose to lex / YACC
- DSEL for declaring *grammars*
  - grammars can be recursive
  - DSEL approximates Backus-Naur Form
- Statically embedded language
  - Domain-specific statements are composed from C++ expressions.

Joel de Guzman

Infix Calculator Grammar

- In Extended Backus-Naur Form

```
group ::= '('    expr    ')'
fact  ::= integer        | group;
term  ::= fact     (('*'    fact) | ('/'    fact))*
expr  ::= term     (('+'    term) | ('-'    term))*
```

Infix Calculator Grammar

- In Boost.Spirit

```
spirit::rule<> group, fact, term, expr;

group   = '(' >> expr >> ')';
fact    = spirit::int_p | group;
term    = fact >> *(('*' >> fact) | ('/' >> fact));
expr    = term >> *(('+' >> term) | ('-' >> term));
```

What is an Expression Template?

- Tree representation of an expression
- Built using templates and operator overloading

```
template< class Derived > struct Op { ... };
```

```
template< class Tag, class L, class R >
struct BinOp : Op< BinOp<Tag, L, R> >
  { L const & left; R const & right; ... };
```

`BinOp<>` is a container for two sub-expressions.

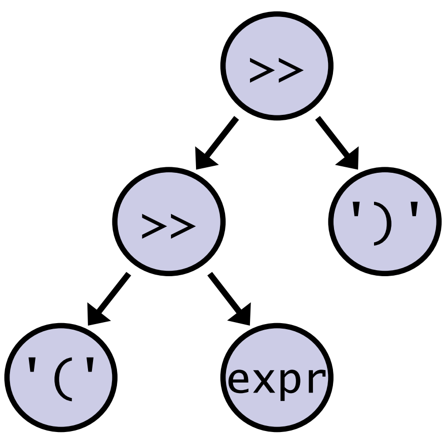
```
template< class L, class R >
BinOp< RShift, L, R > const
operator >> ( Op<L> const & l, Op<R> const & r )
  { ... }
```

Builds the tree representation of "`expr1 >> expr2`".

What is an Expression Template?

- DSELs like Spirit interpret expression trees in a domain-specific way.

```
rule<> group = '(' >> expr >> ')';
```




```
BinOp<
    RShift,
    BinOp<
        RShift,
        Term<char>,
        Term<rule<> >
    >,
    Term<char>
>
```

What is an Expression Template?

- DSELs like Spirit interpret expression trees in a domain-specific way.

```
rule<> group = '(' >> expr >> ')'
```

```
struct rule
{
  template< class Derived >
  rule & operator= ( Op< Derived > const & op )
  {
    ... magic happens here ...
    return *this;
  }
  ...
};
```

Somehow interpret op as a grammar rule. *This is hard!*

DSELs in C++, the Sad Truth

- Very difficult to design
- Code is hard to read and maintain
- Terrible compiler error messages
- So why not a DSEL for defining DSELs?!
  - ☑ Expression construction
  - ☑ Expression evaluation
  - ☑ Expression introspection
  - ☑ Expression transformation

DSEL Design with Boost.Proto

Or, "It's the Grammar, stupid."

A Calculator DSEL

- Expression construction

```
using namespace boost::proto;

template<class I> struct arg { typedef I arity; };
terminal< arg< mpl::int_<1> > >::type _1 = {{}};
terminal< arg< mpl::int_<2> > >::type _2 = {{}};

int main()
{
    (_1 + _2);
    return 0;
}
```

```
expr< tag::plus, args2< ref_< expr<
tag::terminal, args1< arg< mpl::int_<1>
> >, 1 > const >, ref_< expr<
tag::terminal, args1< arg< mpl::int_<2>
> >, 1> const > >, 2>
```

Calculator, Reloaded

- Expression evaluation

```
// To be defined ...
struct calculator_context // ...

int main()
{
    // Placeholders have values 1. and 2. respectively:
    calculator_context ctx(1., 2.);

    // Evaluate a calculator expression with the
    // calculator_context
    std::cout << eval(_1 + _2, ctx) << std::endl;
}
```

Calculator, Reloaded

- Expression evaluation, continued

```
// This describes how to evaluate calculator expressions
struct calculator_context
  : callable_context< calculator_context const >
{
    double d[2];
    typedef double result_type;
    calculator_context(double d1, double d2) { d[0] = d1; d[1] = d2; }

    template<typename T>
    double operator()(tag::terminal, arg<T>) const
    {
        return d[T()-1]; // handle argument placeholders
    }
};
```

The Context Concept

- All context types must look like this:

```
struct Context
{
   template<typename Expr> struct eval
   {
      typedef unspecified result_type;
      result_type operator()(Expr &expr, Context &context) const;
   };
};
```

- Helpers for building contexts
  - default_context / default_eval
  - callable_context / callable_eval

default_context, detail.

```
struct default_context {
    template<class Expr> struct eval
      : default_eval<Expr, default_context const> {};
};

template<class Expr, class Ctx> //, class T = typename Expr::tag_type
struct default_eval< Expr, Ctx, tag::plus >
{
    static Expr &sexpr;
    static Ctx &sctx;
    typedef
        BOOST_TYPEOF(eval(left(sexpr),sctx) + eval(right(sexpr),sctx))
    result_type;
    result_type operator()(Expr &expr, Ctx &ctx) const
    {
        return eval(left(expr),ctx) + eval(right(expr),ctx);
    }
};
```

callable_context, detail.

```
template<class Derived>
struct callable_context {
    template<class Expr> struct eval
      : callable_eval<Expr, Derived> {};
};

template<class Expr, class Derived> //, int I = Expr::arity::value
struct callable_eval< Expr, Derived, 2 >
{
    typedef typename Derived::result_type result_type;
    result_type operator()(Expr &expr, Derived &ctx) const
    {
        return ctx(typename Expr::tag_type()
                  , arg_c<0>(expr)
                  , arg_c<1>(expr));
    }
};
```

Expression Introspection

- Query the properties of expression trees at compile time
- Utilities for defining *patterns* (aka, *grammars*)
- `proto::matches<>` for matching an expression to a pattern.

"It's the Grammar, stupid."

An Input/Output DSEL

```
// Terminals for lazy input/output expressions:
terminal< std::istream & >::type cin_ = { std::cin };
terminal< std::ostream & >::type cout_ = { std::cout };

template<class IO>
void do_io( IO const & io )
{
    eval( io, default_context() );
}

int main()
{
    int i = 0;
    do_io( cin_ >> i );  // reads an int
    do_io( cout_ << i ); // writes an int
}
```

What if `do_io()` needs to know whether `IO` represents an input or an output operation?

Answer: Input/Output Patterns

```
// A pattern to match Input expressions
struct Input
  : shift_right< terminal< std::istream & >, _ >
{};

// A pattern to match Output expressions
struct Output
  : shift_left< terminal< std::ostream & >, _ >
{};

template<class IO>
void do_io(IO const &io)
{
    if( matches<IO, Input>::value  ) std::cout << "Input!\n";
    if( matches<IO, Output>::value ) std::cout << "Output!\n";
    // ...
}
```

Matches "cin_ >> i"

Matches "cout_ << i"

Evaluated at compile time!

Whoops! A Problem ...

```
// A pattern to match Output expressions
struct Output
  : shift_left< terminal< std::ostream & >, _ >
{};
```

- Why doesn't this expression match the `Output` pattern?

```
( cout_ << 1 ) << 2;
```

```
shift_left<
    shift_left<
        terminal< std::ostream & >::type
      , terminal< int const & >::type
    >::type
  , terminal< int const & >::type
>::type
```

Solution: A Recursive Pattern

```
// A pattern to match Output expressions
struct Output
  : or_<
        shift_left< terminal< std::ostream & >, _ >
      , shift_left< Output, _ >
    >
{};
```

- `or_<>` alternates tried in order
- Short-circuit evaluation
- Notice recursion on `Output` pattern!

From Patterns to Grammars

- A Proto Grammar for our Calculator:

```
struct Calculator;

struct Double      : terminal< double > {};
struct Placeholder : terminal< arg<_> > {};
struct Plus        : plus< Calculator, Calculator > {};
struct Minus       : minus< Calculator, Calculator > {};
struct Multiplies  : multiplies< calculator, Calculator > {};
struct Divides     : divides< Calculator, Calculator > {};

struct Calculator
  : or_<
        Double, Placeholder, Plus, Minus, Multiplies, Divides
    >
{};
```

What are grammars good for?

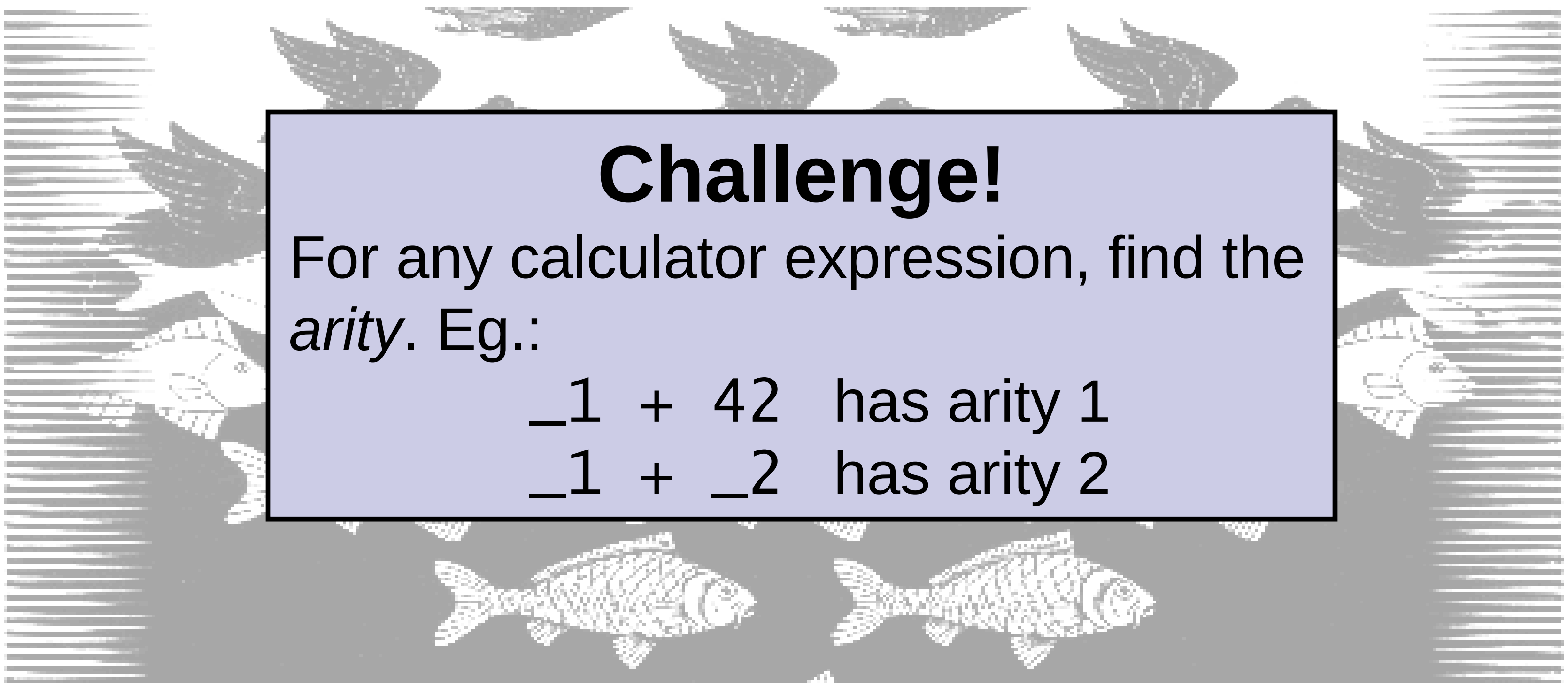
```
// A function that only accepts valid calculator expressions!
template<class Expr>
typename enable_if< matches< Expr, Calculator > >::type
evaluate( Expr const & expr )
{
    // ...
}
```

- `enable_if<>` tricks to prune overload sets
- `mpl::if_<>` for type selections
- Compile-time assertions
- Defining tree transformations
- Controlling Proto's operator overloading

"It's the Grammar, stupid."

Expression Transformation

- Turns an expression tree into some other object
- Transformations attached to grammar rules
- Transformations provided by Proto or user defined



Challenge!

For any calculator expression, find the *arity*. Eg.:

`_1 + 42` has arity 1

`_1 + _2` has arity 2

Arity Calculation

- Calculator expression arity is calculated as follows:

Expression	**Arity**
double	0
_1	1
_2	2
Left *op* Right	max(arity(Left), arity(Right))

From Grammars to Transforms

```
// Ye olde calculator grammar
struct Calculator;

struct Double       : terminal< double > {};
struct Placeholder : terminal< arg<_> > {};
struct Plus         : plus< Calculator, Calculator > {};
struct Minus        : minus< Calculator, Calculator > {};
struct Multiplies  : multiplies< calculator, Calculator > {};
struct Divides      : divides< Calculator, Calculator > {};



struct Calculator
  : or_<
        Double, Placeholder, Plus, Minus, Multiplies, Divides
    >
{};
```

Double Transform

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

Anything that matches `Double` is transformed into `mpl::int_<0>`

```
struct Calculator
  : or_<
        trans::always< Double, mpl::int_<0> >, ...
    >
{};
```

Placeholder Transform

```
template<class I> struct arg { typedef I arity; };

// A custom transform that gets the arity of a placeholder
template<class Placeholder> struct arg_arity : Placeholder
{
    template<class Expr, class, class> struct apply {
        typedef typename Expr::arg0_type::arity type;
    };
};

struct Placeholder : terminal< arg<_> > {};

struct Calculator
  : or_< ...
      , arg_arity< Placeholder >, ...
    >
{};
```

Expr is the type that matched Placeholder

Anything that matches Placeholder is transformed according to the arg_arity<> transform.

Binary Operator Transform

```
// A custom transform that gets the max arity of a binary operation
template<class BinOp> struct max_arity : BinOp
{
    template<class Expr, class S, class V> struct apply {
        typedef typename BinOp
            ::template apply<Expr, S, V>::type child_arities;
        typedef typename child_arities::arg0_type left_arity;
        typedef typename child_arities::arg1_type right_arity;
        typedef typename mpl::max<left_arity, right_arity>::type type;
    };
};
```

Apply `plus<>`'s transform; recursively transforms left and right children and reassembles into `plus<>`.

```
struct Calculator
  : or_< ...
      , max_arity< plus< Calculator, Calculator > >, ...
    >
{};
```

Anything that matches `plus<...>` is transformed according to the `max_arity<>` transform.

Binary Operator Transform

The type of an expression that matches `plus< Calculator, Calculator >`

```
typedef typename plus< Calculator, Calculator >
    ::template apply<
        expr< tag::plus, args2< Left, Right > >, S, V
    >::type child_arities;
```

The result type of `plus<>` transform

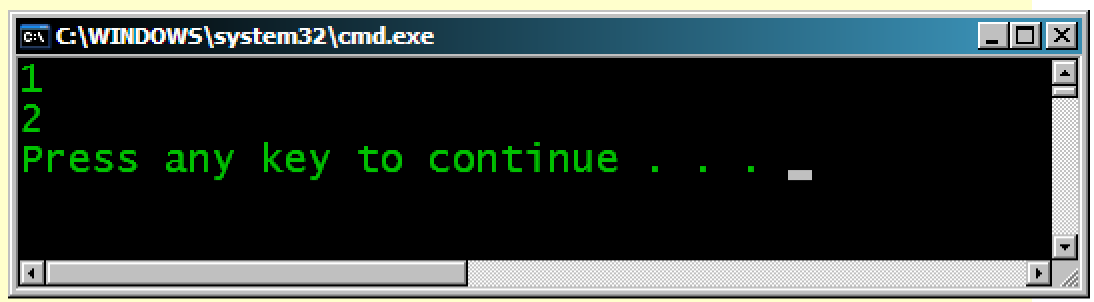
```
typedef expr<
    tag::plus
  , args2<
        typename Calculator::template apply<Left, S, V>::type
      , typename Calculator::template apply<Right, S, V>::type
    >
> child_arities;
```

The Complete Transform

```
struct Calculator;
struct Double      : terminal< double > {};
struct Placeholder : terminal< arg<_> > {};
struct Plus        : plus< Calculator, Calculator > {};
struct Minus       : minus< Calculator, Calculator > {};
struct Multiplies  : multiplies< calculator, Calculator > {};
struct Divides     : divides< Calculator, Calculator > {};

struct Calculator
  : or_<
        trans::always< Double, mpl::int_<0> >
      , arg_arity< Placeholder >
      , max_arity< Plus >
      , max_arity< Minus >
      , max_arity< Multiplies >
      , max_arity< Divides >
    >
{};
```

It works!

```
// Defined as before
struct Calculator ... ;

// Display the arity of a calculator expression
template<class Expr>
void print_arity(Expr const & expr)
{
    BOOST_MPL_ASSERT((matches<Expr, Calculator>));
    typedef typename Calculator::apply<Expr, int, int>::type arity;
    std::cout << arity::value << std::endl;
}

int main()
{
    print_arity(_1 + 42);
    print_arity(_1 + _2);
}
```



Extending Proto

Giving Proto expressions extra smarts.

Calculator, Reloaded

- Expression evaluation flashback

```
// To be defined ...
struct calculator_context // ...

int main()
{
    // Placeholders have values 1. and 2. respectively:
    calculator_context ctx(1., 2.);

    // Evaluate a calculator expression with the calculator_context
    std::cout << eval(_1 + _2, ctx) << std::endl;
}
```

Q: Is `eval()` always needed to make proto expressions do something?

Calculator, Reloaded

- A Calculator expression as an STL functor?

```
int main()
{
    double data[] = { 1., 2., 3., 4. };

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

Whoops! "`_1 * _1`" doesn't have an `operator()` that takes and returns a double.

Calculator, Revolutions

- Expression wrappers to add behaviors!

```
struct calc_domain;

template< class Expr >
struct calc : extends< Expr, calc< Expr >, calc_domain >
{
  calc( Expr const &expr = Expr() )
   : extends< Expr, calc< Expr >, calc_domain >( expr )
  {}

  typedef double result_type;
  result_type operator()( double d1 = 0.0, double d2 = 0.0 ) const
  {
    calculator_context ctx( d1, d2 );
    return eval( *this, ctx );
  }
};
```

The "domain" of our wrapped expressions

`calc<Expr>` *extends* `Expr`

Define `operator()` to evaluate expr with `calculator_context`

Calculator, Revolutions

- Wrap our terminals ...

```
// OK, _1 and _2 are calc<> expressions in the calc_domain
calc< terminal< arg< mpl::int_<1> > >::type > _1;
calc< terminal< arg< mpl::int_<2> > >::type > _2;
```

- Hook Proto's expression generator ...

```
struct calc_domain
  : domain< generate< calc > >
{};
```

In `calc_domain`, wrap all expression types in our `calc<>` wrapper.

Calculator, Revolutions

- It works!

```
int main()
{
    double data[] = { 1., 2., 3., 4. };

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

Watch 1

Name	Value	Type
data	0x0012ff44	double [4]
[0]	1.0000000000000000	double
[1]	4.0000000000000000	double
[2]	9.0000000000000000	double
[3]	16.000000000000000	double

Summing up ...

- Proto makes C++ DSEL design easy and fun!
- Good for expression template ...
  - ... construction
  - ... evaluation
  - ... introspection
  - ... transformation
- Powerful extensibility mechanisms
- Used by xpressive, Spirit-2, Karma, etc ...
- Available in Boost CVS and the File Vault

Questions?