Boost.Proto by Doing

or, “Proto is useful for lots of everyday things. Really.”

Talk Overview

- Basic Example: Boost.Assign
  - Front Ends, Back Ends
  - Expression Extension
  - Simple Grammars and Transforms
- (Intermediate Example: Future Groups)
- Advanced Example: Boost.Phoenix
  - The Expression Problem
  - Domains and Sub-Domains
  - Extensible Grammars and Transforms
- Improving Diagnostics

Example 1

`map_list_of()` from Boost.Assign

map_list_of

```
#include <map>
#include <cassert>
#include <boost/assign/list_of.hpp> // for 'map_list_of()'
using namespace boost::assign; // bring 'map_list_of()' into scope

int main()
{
    std::map<int,int> next = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
    assert( next.size() == 5 );
    assert( next[ 1 ] == 2 );
    assert( next[ 5 ] == 6 );
}
```

What is map_list_of?

Proto Front Ends

Plant a seed, grow a tree

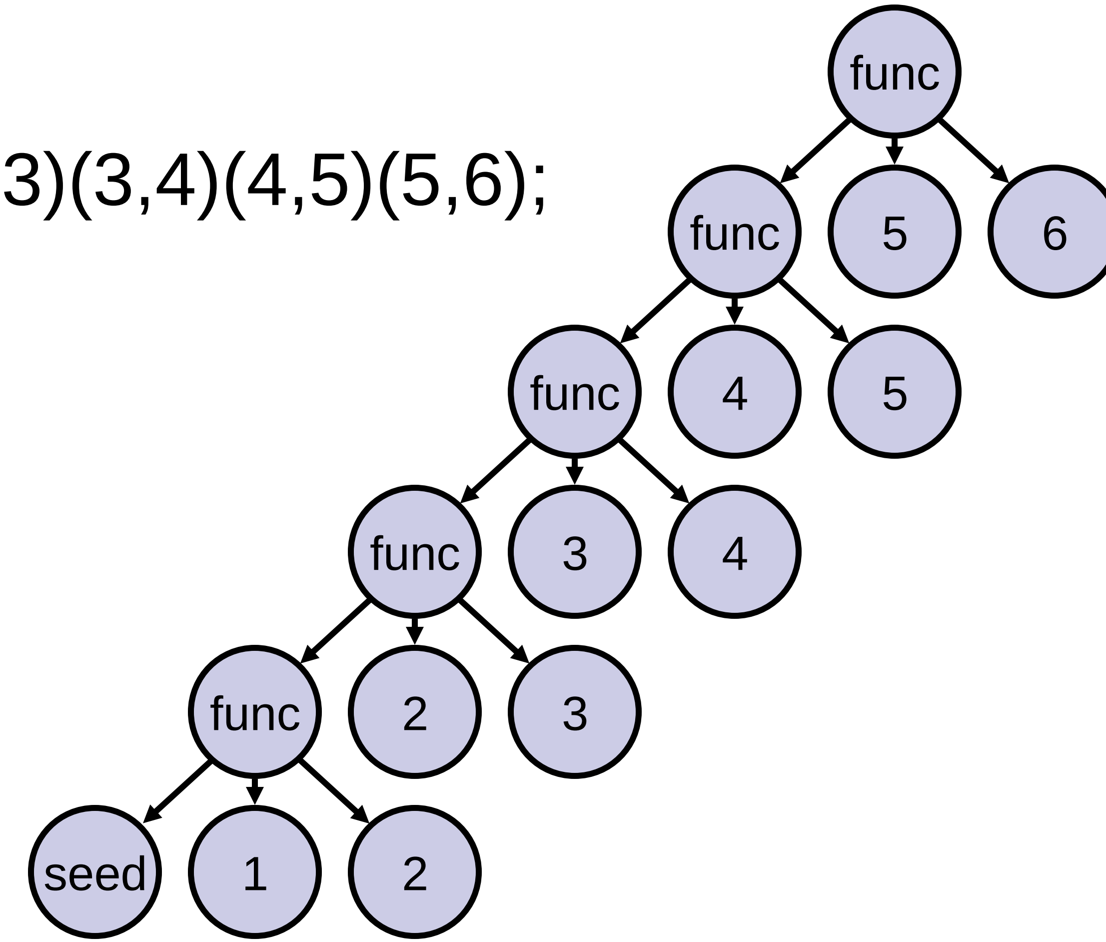
Define a "Seed" Terminal

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

struct map_list_of_ {};
proto::terminal<map_list_of_>::type const map_list_of = {{}};

int main()
{
    map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
}
```

Compiles and runs!
(And does nothing.)

Just another bloody tree…

map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);

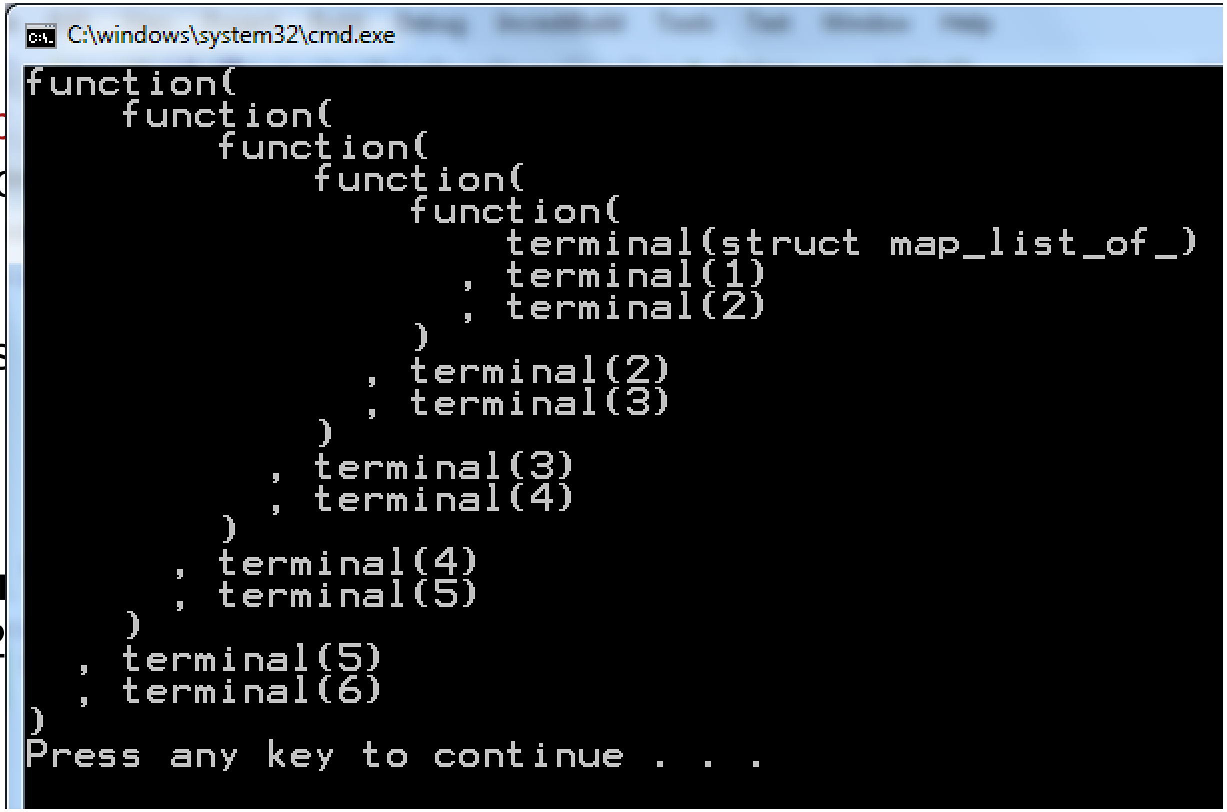


Pretty-print trees with display_expr

```
#include <iostream>
#include <boost/proto/proto.hpp>
namespace proto = boost::proto;

struct map_list_of_ {};
proto::terminal<map_list_of_>::type const map_list_of = {{}};

int main()
{
    proto::display_expr(
        map_list_of(1,2)(2,3)(3,4)(4,5)(5,6)
    );
}
```

Pretty-print trees with display_expr

```
#include <iostream>
#include <boost/proto/p
namespace proto = boo

struct map_list_of_ {};
proto::terminal<map_lis

int main()
{
    proto::display_expr
        map_list_of(1,2)(2
    );
}
```

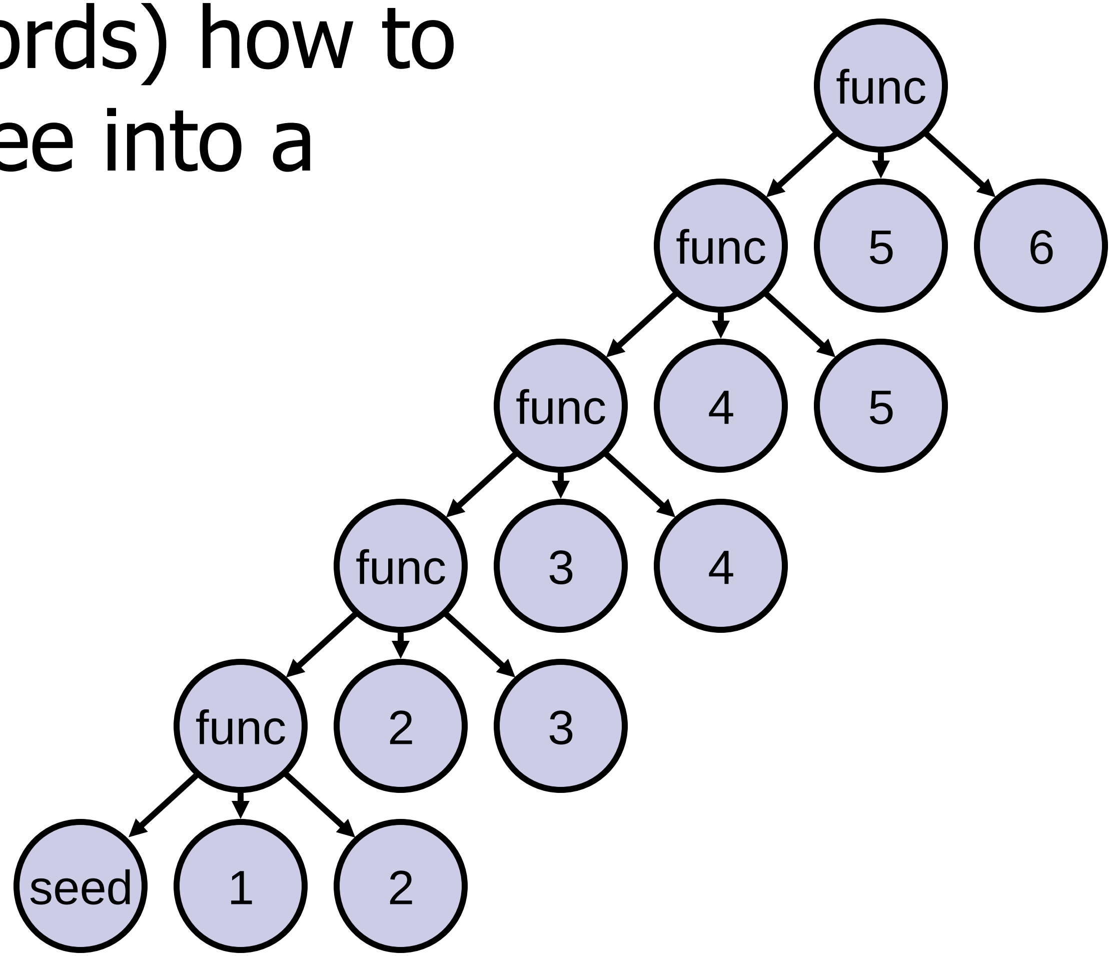


Back Ends

Tree-walking, take 1

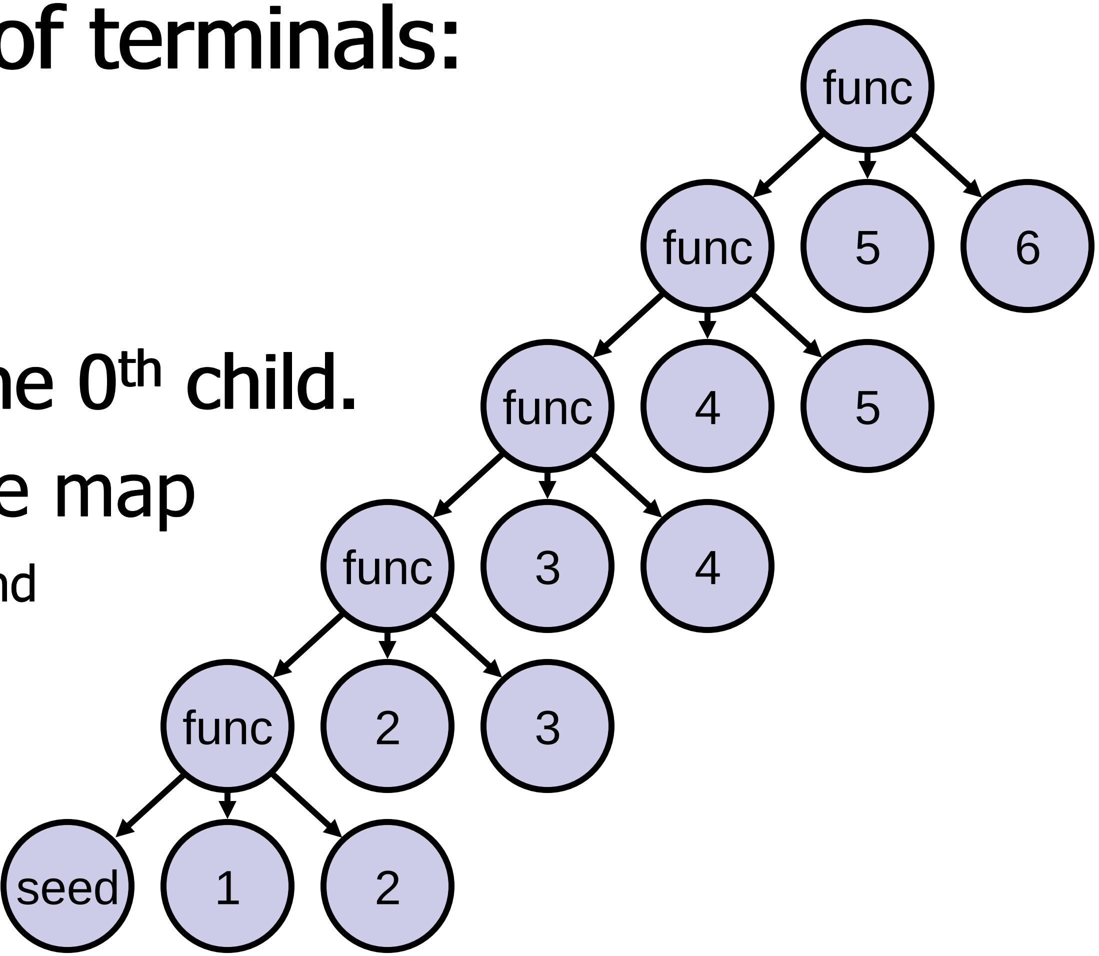
Populate a map from a tree …

- Describe (in words) how to turn the this tree into a std::map.



Populate a map from a tree …

- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0th child.
  2. Insert into the map the 1st and 2nd children.



Populate a map from a tree …

```
template<typename Map>
void fill_map( proto::terminal<map_list_of_>::type, Map& ) // end recursion
{}

template<class Fun, class Map>
void fill_map( Fun const& f, Map& m )
{
   fill_map( proto::child_c<0>(f), m ); // recurse on 0th child
   m[ proto::value( proto::child_c<1>(f) ) ] = proto::value( proto::child_c<2>(f) );
}

int main()
{
   std::map<int, int> m;
   fill_map( map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), m );
}
```

Populate a map from a tree …

```
template<typename Map>
void fill_map( proto::terminal<map_list_of_>::type, Map& ) // end recursion
{}

template<class Fun, class Map>
void fill_map( Fun const& f, Map& m )
{
   fill_map( proto::child_c<0>(f), m ); // recurse on 0th child
   m[ proto::value( proto::child_c<1>(f) ) ] = proto::value( proto::child_c<2>(f) );
}

int main()
{
   std::map<int, int> m;
   fill_map( map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), m );
}
```

proto::child_c<N>() extracts the Nth child.

proto::value() extracts the value from a terminal.

It really works!

```cpp
template<typename Map>
void fill_map(proto::terminal<map_list_of_>::type, Map&)
{}

template<class Fun, class Map>
void fill_map(Fun const &fun, Map& m)
{
    fill_map(proto::child_c<0>(fun), m); // recurse
    m[proto::value(proto::child_c<1>(fun))] = proto::value(proto::child_c<2>(fun));
}

int main()
{
    std::map<int, int> m;
    fill_map( map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), m );
}
```

Autos

Name	Value	Type
m	[5]((1,2),(2,3),(3,4),(4,5),(5,6))	std::map<int,i
map_list_of	{child0={...} }	boost::proto::e

Call Stack

Name
scratch.exe!main() Line 134
scratch.exe!__tmainCRTStartup() Line 586 + 0x
scratch.exe!mainCRTStartup() Line 403

Are we done?

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

struct map_list_of_ {};
terminal<map_list_of_>::type map_list_of;

template<typename Map>
void fill_map( terminal<map_list_of_>::type, Map& )
{}

template<class Fun, class Map>
void fill_map( Fun const& f, Map& m )
{
   fill_map( child_c<0>(f), m );
   m[ value( child_c<1>(f) ) ] = value( child_c<2>(f) );
}

int main()
{
   std::map<int, int> m;
   fill_map( map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), m );
}
```

Not done yet!

- We need to eliminate fill_map from the interface.
- We need to check expression for validity.

Expression Tree Extensibility

Adding members to trees

How to eliminate fill_map?

```
#include <map>
#include <cassert>
#include <boost/assign/list_of.hpp>
using namespace boost::assign;

int main()
{
    std::map<int,int> next = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
    assert( next.size() == 5 );
    assert( next[ 1 ] == 2 );
    assert( next[ 5 ] == 6 );
}
```

This tree must be convertible to a std::map.

Introducing proto::extends

```
// Define an expression wrapper that provides a conversion to a map
template<typename Expr>
struct map_list_of_expr
  : proto::extends< Expr, map_list_of_expr< Expr >, map_list_of_domain >
{
  map_list_of_expr( Expr const & expr = Expr() )
    : proto::extends< Expr, map_list_of_expr< Expr >, map_list_of_domain >( expr )
  {}

  template<class K, class V, class C, class A>
  operator std::map<K,V,C,A>() const
  {
    std::map<K,V,C,A> m;
    fill_map( *this, m );
    return m;
  }
};

map_list_of_expr< proto::terminal< map_list_of_ >::type > const map_list_of;

int main()
{
  std::map<int,int> next0 = map_list_of;          // OK!
  std::map<int,int> next1 = map_list_of(1,2);     // OK!
}
```

A map_list_of_expr< T > is just like a T, except:

- it has a conversion to a std::map.
- operations on it produce other map_list_of_expr trees

Domains and Generators

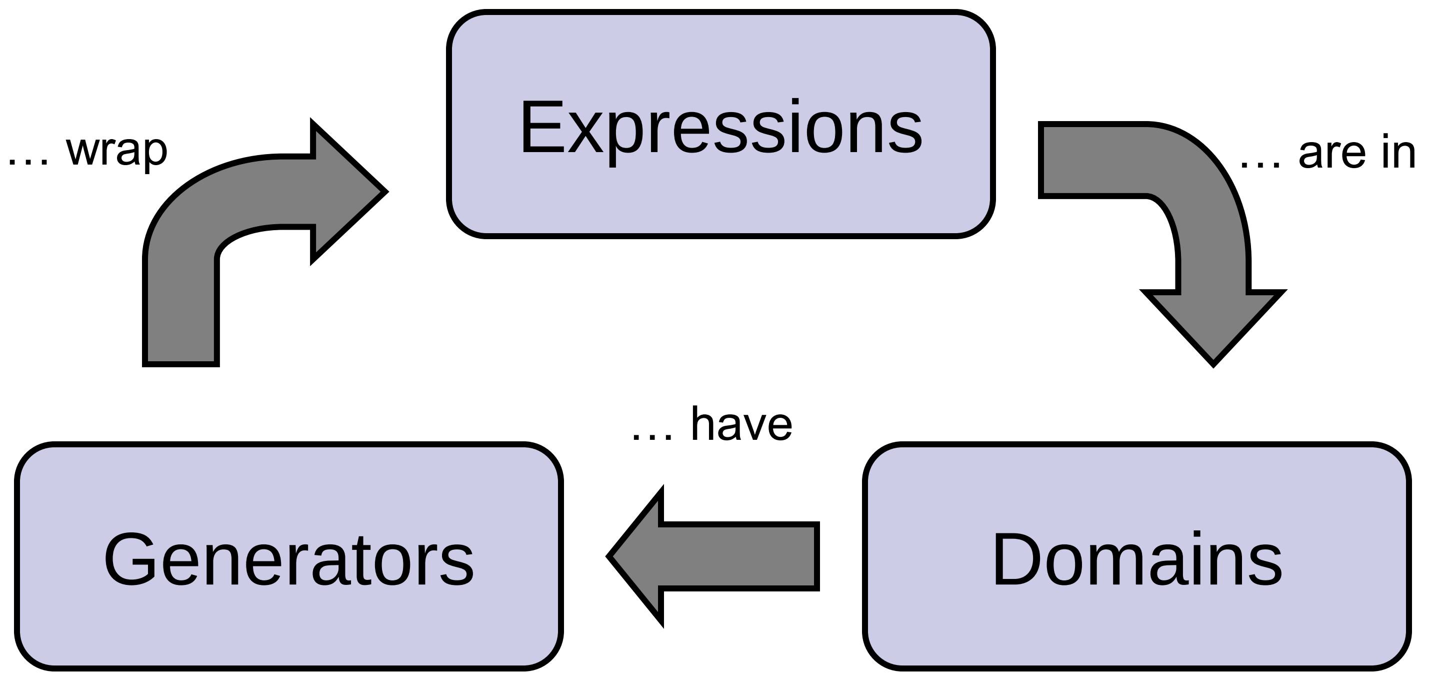
- **Proto "domain":**
  - A type used to associate an expression with a proto "generator"
- **Proto "generator":**
  - A function that does *something* to an expression.

```
template< typename Expr >
struct map_list_of_expr;

struct map_list_of_domain
  : proto::domain< proto::generator< map_list_of_expr > >
{};
```

domain

generator

Expressions, Domains and Generators



Expression Tree Validation

Spotting invalid expressions

IMPORTANT

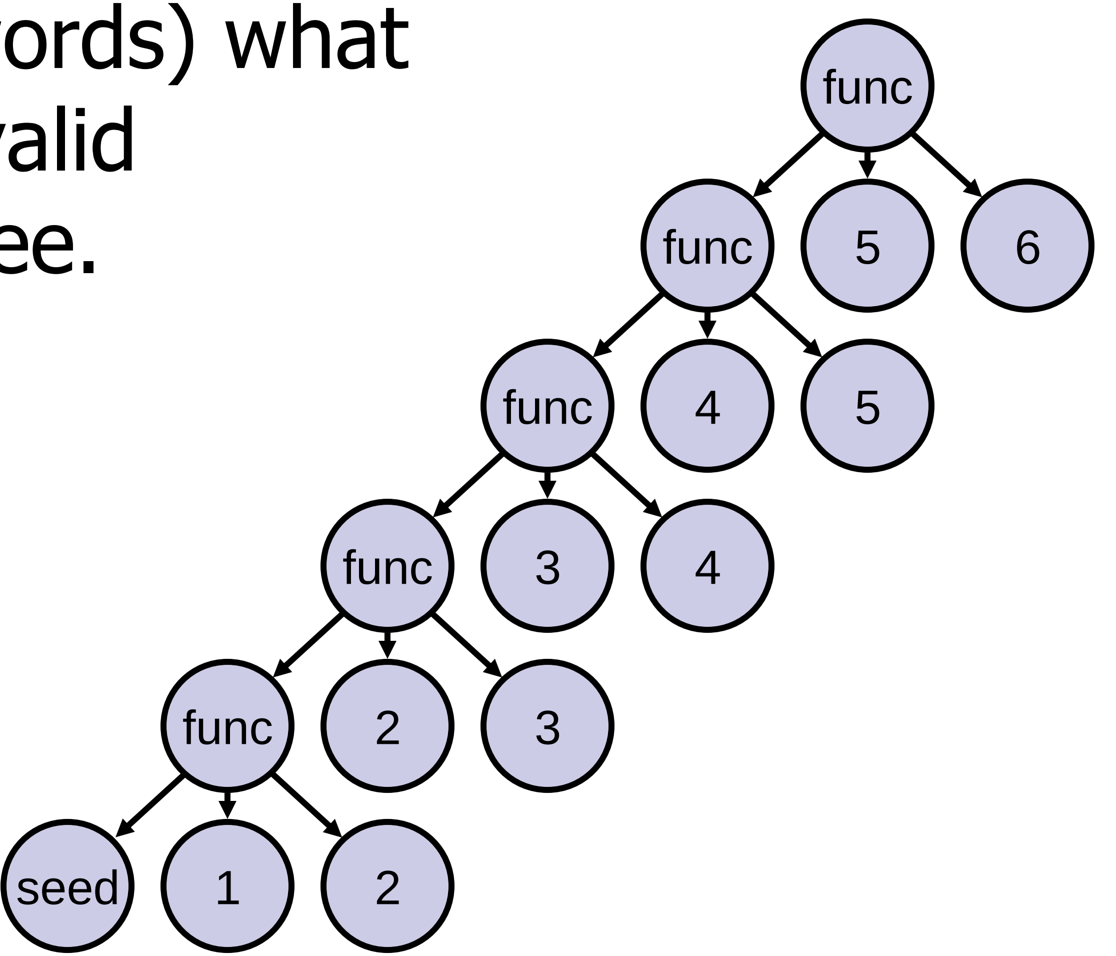
Proto’s Promiscuous Operators

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

struct map_list_of_ {};
proto::terminal<map_list_of_>::type const map_list_of = {{}};

int main()
{
    map_list_of(1,2) * 32 << map_list_of; // WTF???!!!
}
```

This compiles.
Oops.

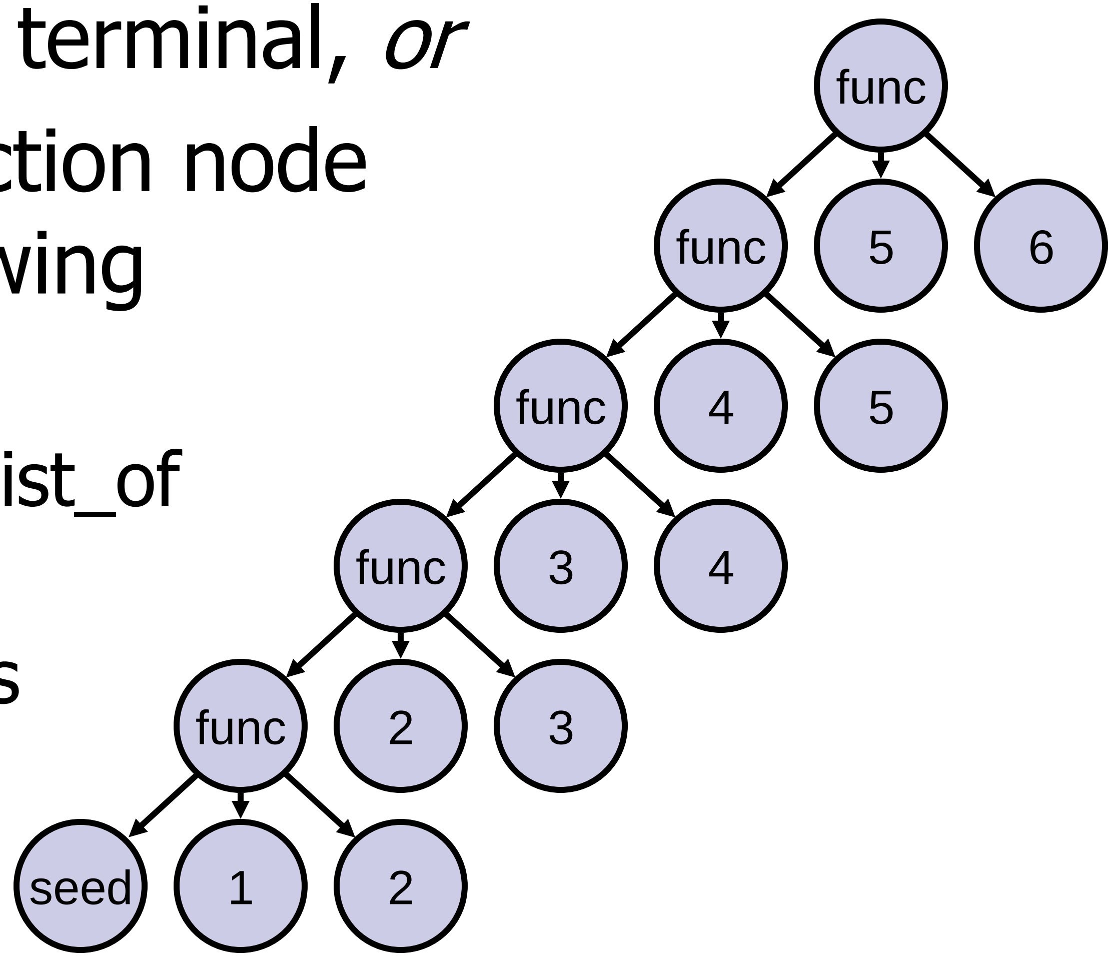
A valid map_list_of tree is …

- Describe (in words) what makes this a valid map_list_of tree.



A valid map_list_of tree is …

- A map_list_of terminal, *or*
- A ternary function node with the following children:
  1. A valid map_list_of tree
  2. Two terminals



A valid map_list_of tree is …

- A map_list_of terminal, *or*
- A ternary function node with the following children:
  1. A valid map_list_of tree
  2. Two terminals

```
using proto::_;

struct MapListOf :
  proto::or_<
      proto::terminal<map_list_of_>
    , proto::function<
        MapListOf
      , proto::terminal<_>
      , proto::terminal<_>
      >
    >
{};
```

Detecting Wild Expressions

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

struct map_list_of_ {};
proto::terminal<map_list_of_>::type const map_list_of = {{}};

struct MapListOf : /* as before */ {};

int main()
{
  BOOST_PROTO_ASSERT_MATCHES(
     map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), MapListOf );

  BOOST_PROTO_ASSERT_MATCHES_NOT(
     map_list_of(1,2) * 32 << map_list_of, MapListOf );
}
```

These are evaluated at compile time

Grammars and Algorithms

A valid map_list_of tree is:

- A map_list_of terminal, or
- A ternary function node with the following children:
  1. A valid map_list_of tree
  2. Two terminals

Populate a map from a tree:

- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0th child.
  2. Insert into the map the 1st and 2nd children.

Writing Proto Transforms

Tree-walking, take 2

(“Buckle your seatbelt Dorothy,
‘cause Kansas is going bye-bye.”)

Populate a map from a tree…

- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0^{th} child.
  2. Insert into the map the 1^{st} and 2^{nd} children.

```
struct MapListOf :
  proto::or_<
      proto::terminal<map_list_of_>
    , proto::function<
          MapListOf
        , proto::terminal<_>
        , proto::terminal<_>
      >
    >
{};
```

Populate a map from a tree…

- For map_list_of terminals:
  1. Do nothing.

```
proto::when<
    proto::terminal<map_list_of_>
  , proto::_void
>
```

Use proto::when to associate a transform with a grammar rule.

Populate a map from a tree…

- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0^{th} child.
  2. Insert into the map the 1^{st} and 2^{nd} children.

```
struct MapListOf :
  proto::or_<
      proto::terminal<map_list_of_>
    , proto::function<
        MapListOf
      , proto::terminal<_>
      , proto::terminal<_>
      >
    >
{};
```

Populate a map from a tree…

Use proto::and_ to specify a sequence of transforms.

- Otherwise:
  1. Recurse on the 0th child.
  2. Insert into the map the 1st and 2nd children.

```
proto::when<
    proto::function<
        MapListOf
      , proto::terminal<_>
      , proto::terminal<_>
    >
  , proto::and_<
        MapListOf(proto::_child0(_))
      , /* … */
    >
>
```

Use function types to compose transforms.

Composite Transforms

- Use function *types* to represent function *invocations*.

```
MapListOf( proto::_child0( _ ) )
```

Return the current node

Return the 0^{th} child

Invoke the MapListOf transform

Composite Transforms

MapListOf(proto::_child0)

All transforms operate on the current node by default.

Populate a map from a tree…

Define your own actions.

- Otherwise:
  1. Recurse on the 0^{th} child.
  2. Insert into the map the 1^{st} and 2^{nd} children.

```
proto::when<
    proto::function<
        MapListOf
      , proto::terminal<_>
      , proto::terminal<_>
    >
  , proto::and_<
        MapListOf(proto::_child0)
      , map_insert(
            proto::_state
          , proto::_value(proto::_child1)
          , proto::_value(proto::_child2)
        )
    >
>
```

Populate a map from a tree…

Define your own actions.

```
// A simple TR1-style function type that
// inserts a (key, value) pair into a map.
struct map_insert : proto::callable
{
    typedef void result_type;

    template<class M, class K, class V>
    void operator()(M & m, K k, V v) const
    {
        m[ k ] = v;
    }
};
```

```
proto::when<
    proto::function<
        MapListOf
      , proto::terminal<_>
      , proto::terminal<_>
    >
  , proto::and_<
        MapListOf(proto::_child0)
      , map_insert(
            proto::_state
          , proto::_value(proto::_child1)
          , proto::_value(proto::_child2)
        )
    >
>
```

Populate a map from a tree…

Pass extra "state" to your transforms, like, say, a std::map.

- Otherwise:
  1. Recurse on the 0^{th} child.
  2. Insert into the map the 1^{st} and 2^{nd} children.

```
proto::when<
    proto::function<
        MapListOf
      , proto::terminal<_>
      , proto::terminal<_>
    >
  , proto::and_<
        MapListOf(proto::_child0)
      , map_insert(
            proto::_state
          , proto::_value(proto::_child1)
          , proto::_value(proto::_child2)
        )
    >
```

Putting the Pieces Together

```
// Match valid map_list_of expressions and populate a map
struct MapListOf
  : or_<
        when< terminal<map_list_of_>, _void >
      , when<
            function< MapListOf, terminal<_>, terminal<_> >
          , and_<
                MapListOf(_child0)
              , map_insert(_state, _value(_child1), _value(_child2))
            >
        >
    >
{};
```

Using Grammars and Transforms

```
// Match valid map_list_of expressions and populate a map
struct MapListOf  : /* as before */ {};

int main()
{
    // Use MapListOf as a grammar:
    BOOST_PROTO_ASSERT_MATCHES(
        map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), MapListOf );

    // Use MapListOf as a function:
    std::map< int, int > next;
    MapListOf()( map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), next );
    assert( next.size() == 5 );
    assert( next[ 1 ] == 2 );
    assert( next[ 5 ] == 6 );
}
```

The transform's initial state

A Working Expression Extension

```
// Define a domain-specific expression wrapper that provides a conversion to a map
template<typename Expr>
struct map_list_of_expr
  : proto::extends< Expr, map_list_of_expr< Expr >, map_list_of_domain >
{
  map_list_of_expr( Expr const & expr = Expr() )
    : proto::extends< Expr, map_list_of_expr< Expr >, map_list_of_domain >( expr )
  {}

  template<class K, class V, class C, class A>
  operator std::map<K,V,C,A>() const
  {
    BOOST_PROTO_ASSERT_MATCHES(*this, MapListOf);
    std::map<K,V,C,A> m;
    MapListOf()( *this, m );
    return m;
  }
};
```

```
map_list_of_expr< proto::terminal< map_list_of_ >::type > map_list_of;

int main()
{
  std::map<int,int> next0 = map_list_of;                              // OK!
  std::map<int,int> next1 = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);     // OK!
}
```

Proto's Promiscuous Operators

```
/* … as before … */

struct map_list_of_domain
  : proto::domain< proto::generator<map_list_of_expr> >
{};

int main()
{
    map_list_of(1,2) * 32 << map_list_of; // WTF???!!!
}
```

Yeah, but this *still* compiles.

Proto's Promiscuous Operators

```
/* … as before … */

struct map_list_of_domain
  : proto::domain< proto::generator<map_list_of_expr>, MapListOf >
{};

int main()
{
    map_list_of(1,2) * 32 << map_list_of; // WTF???!!!
}
```

```
1>c:\boost\org\trunk\libs\proto\scratch\main.cpp(59) : error
C2893: Failed to specialize function template 'const
detail::as_expr_if<boost::proto::tag::multiplies,const
Left,const Right>::type boost::proto::exprns_::operator
*(const Left &,const Right &)'
```

The Complete Solution

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

struct map_insert : callable
{
    typedef void result_type;

    template<class M, class K, class V>
    void operator()(M & m, K k, V v) const
    {
        m[ k ] = v;
    }
};

struct map_list_of_
{};

struct MapListOf
  : or_<
        when<terminal<map_list_of_>, _void>
      , when<
            function<MapListOf, terminal<_>, terminal<_> >
          , and_<
                MapListOf(_child0)
              , map_insert(_state, _value(_child1), _value(_child2))
            >
        >
    >
{};
```

```
template<typename Expr>
struct map_list_of_expr;

struct map_list_of_domain
  : domain<pod_generator<map_list_of_expr>, MapListOf>
{};

template<typename Expr>
struct map_list_of_expr
{
    BOOST_PROTO_EXTENDS(Expr, map_list_of_expr<Expr>,
        map_list_of_domain)

    template< class K, class V, class C, class A>
    operator std::map<K, V, C, A> () const
    {
        BOOST_PROTO_ASSERT_MATCHES(*this, MapListOf);
        std::map<K, V, C, A> map;
        MapListOf()(*this, map);
        return map;
    }
};

map_list_of_expr<terminal<map_list_of_>::type> const map_list_of = {{}};

int main()
{
    std::map<int, int> next = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
}
```

map_list_of: Take-Away

- Proto is useful even for small problems
- It makes your code:
  - short
  - declarative
  - efficient

Example 2: Future Groups

Gettin' Jiggy with Proto Transforms

What are Future Groups?

- future<int> is the (deferred) result of an asynchronous call.
- Converting a future<int> to an int blocks for the call to finish.
- Could be implemented with threads, thread pools, processes, cloud computing, ponies, whatever. The point is, you don't care.

What are Future Groups?

- A future *group* is an expression involving multiple futures.
- It may block until all or some of the results are available.
- Think of Win32's WaitForMultipleObjects API, or Linux sem_wait/sem_trywait

Future Group Syntax

x || y

Wait for either x *or* y to finish. x and y must have the same type. Result is that type.

x && y

Wait for both x *and* y to finish. x and y can have different types. Result is a tuple.

Future Groups: Example

```
int main()
{
    future<A> a0, a1;
    future<B> b0, b1;
    future<C> c;

    /* ... initialize the futures with asynchronous calls. */

    A                        t0 = a0.get();
    fusion::vector<A, B, C>  t1 = (a0 && b0 && c).get();
    fusion::vector<A, C>     t2 = ((a0 || a1) && c).get();
    fusion::vector<A, B, C>  t3 = ((a0 && b0 || a1 && b1) && c).get();
}
```

Future Groups: Strategy

- Make future<X> a Proto terminal.
- Define the grammar of valid future group expressions.
- Define transforms that compute the result of a future group expression.

Future Group Grammar

```
// Define the grammar of future group expressions
struct FutureGroup
  : or_<
        terminal<_>
        // (a && b)
      , logical_and<FutureGroup, FutureGroup>
        // (a || b)
      , logical_or<FutureGroup, FutureGroup>
    >
{};
```

Future Group Transforms

- Convert terminals into fusion::single_view
- Convert && into fusion::joint_view
- Convert || into either left or right (but ensure their types are the same)

Future Group Grammar

```
// terminals become a single-element Fusion sequence
when<
    terminal<_>
  , fusion::single_view<_value>(_value)
>
```

Reuse function syntax to mean “construct a single_view” (because single_view is not “callable”)

Future Group Grammar

```
// (a && b) becomes a concatenation of the sequence
// from 'a' and the one from 'b':
when<
    logical_and<FutureGroup, FutureGroup>
  , fusion::joint_view<
        add_const<FutureGroup(_left)>
      , add_const<FutureGroup(_right)>
    >(FutureGroup(_left), FutureGroup(_right))
>
```

In object transforms, Proto looks for nested transforms and evaluates them. (add_const is needed to satisfy joint_view constructor.)

Future Group Grammar

```
// (a || b) becomes the sequence for 'a', so long
// as it is the same as the sequence for 'b'.
when<
    logical_or<FutureGroup, FutureGroup>
  , pick_left<
        FutureGroup(_left)
      , FutureGroup(_right)
    >(FutureGroup(_left))
>
```

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

In object transforms, Proto uses a nested ::type typedef if it finds one.

Assembled Future Group Grammar

```
// 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
  : or_<
        // terminals become a single-element Fusion
        // sequence
        when<
            terminal<_>
          , fusion::single_view<_value>(_value)
        >
        // (a && b) becomes a concatenation of the
        // sequence from 'a' and the one from 'b':
      , when<
            logical_and<FutureGroup, FutureGroup>
          , fusion::joint_view<
                add_const<FutureGroup(_left)>
              , add_const<FutureGroup(_right)>
            >(FutureGroup(_left), FutureGroup(_right))
        >
        // (a || b) becomes the sequence for 'a',
        // so long as it is the same as the
        // sequence for 'b'.
      , when<
            logical_or<FutureGroup, FutureGroup>
          , pick_left<
                FutureGroup(_left)
              , FutureGroup(_right)
            >(FutureGroup(_left))
        >
    >
{};
```

Future Group Extension

```
template<class E>
struct future_expr;

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

Flatten joint_views and single_views into a plain Fusion vector

```
// 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
  : extends<E, future_expr<E>, future_dom>
{
    explicit future_expr(E const & e)
      : extends<E, future_expr<E>, future_dom>(e)
    {}

    typename fusion::result_of::as_vector<
        typename boost::result_of<FutureGroup(E const &)>::type
    >::type
    get() const
    {
        return fusion::as_vector(FutureGroup()(*this));
    }
};
```

The future<> type

```
// The future<> type has an even simpler .get()
// member function.
template<class T>
struct future
  : future_expr<typename terminal<T>::type>
{
    future(T const & t = T())
      : future_expr<typename terminal<T>::type>(terminal<T>::type::make(t))
    {}

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

future<T> is just a future_expr of a Proto terminal.

All Proto expression types have a static ::make member function.

DONE

Future Groups: Take-Aways

- We can use Proto object transforms to easily transform Proto trees into other types.
- Proto transforms recognize common idioms like TR1-compliant function objects and ::type metafunction evaluation
- Makes it easy to reuse readily available types as-is (e.g. boost::add_const, fusion::joint_view)

Example 3

Boost.Phoenix

EXPERIMENTAL

The Expression Problem

"The **expression problem** […] refers to a fundamental dilemma of programming: To which degree can your application be structured in such a way that both the data model and the set of … operations over it can be extended without the need to modify existing code, without the need for code repetition and without runtime type errors."[1]

[1] Mads Torgersen, "The Expression Problem Revisited"

http://www.daimi.au.dk/~madst/ecoop04/main.pdf

Solving The Expression Problem

"A solution to the expression problem […] allows both new data types and operations to be subsequently added any number of times

1. without modification of existing source code
2. without replication of non-trivial code
3. without risk of unhandled combinations of data and operations" [1]

Introducing Boost.Phoenix

```
using namespace boost::phoenix;

std::for_each( v.begin(), v.end(),
    if_( _1 > 5 )
    [
        std::cout << _1 << ", "
    ]
);
```

Boost.Phoenix lambda expression

Phoenix 3 Design Goals

- Small extensible core
- Everything else implemented as an extension
- Proto-based ET representation for:
  - Simple interoperability with other Proto-based ET libraries
  - Separation of ET data and algorithm for reusability and maintainability
  - Unification of Boost's placeholders (::_1, boost::lambda::_1, boost::phoenix::_1, etc.)

Extensible Grammars… How?

```
// Match some expressions and do some stuff
struct my_grammar
  : or_<
        when< terminal<_>, do_stuff1(_) >
      , when<
            function< vararg< my_grammar > >
          , do_stuff2(_)
        >
    >
{};
```

C++ structs are closed to extension

Extensiblility With proto::switch_

```
// Match some expressions
// and do some stuff
struct my_grammar
  : or_<
        when< terminal< /*…*/ >, /*…*/ >
      , when< function< /*…*/ >, /*…*/ >
    >
{};
```

```
// Match some expressions
// do some stuff
struct my_grammar
  : switch_<struct my_grammar_cases>
{};

struct my_grammar_cases
{
    template< class Tag >
    struct case_ : not_< _ >
    {};
};

template<>
struct my_grammar_cases::case_< tag::terminal >
  : when< terminal< /*…*/ >, /*…*/ >
{};

template<>
struct my_grammar_cases::case_< tag::function >
  : when< function< /*…*/ >, /*…*/ >
{};
```

- Fast tag dispatching
- Extensible grammar

Building Boost.Phoenix Core

```
////////////////////////////////////
// phoenix expression wrapper
template<class Expr>
struct actor;

////////////////////////////////////
struct phoenix_domain
  : domain<pod_generator<actor> >
{};

////////////////////////////////////
// phoenix grammar and expression
// evaluator
struct eval
  : switch_<struct eval_cases>
{};

////////////////////////////////////
struct eval_cases
{
    template<class Tag>
    struct case_
      : _default<eval>
    {};
};
```

Does the “default” C++ action on an expression node.
E.g. “a + b” means binary addition.

Uses Boost.Typeof to deduce return types

Building Boost.Phoenix Core

```
/////////////////////////////////////
// phoenix expression wrapper
template<class Expr>
struct actor;

/////////////////////////////////////
struct phoenix_domain
  : domain<pod_generator<actor> >
{};

/////////////////////////////////////
// phoenix grammar and expression
// evaluator
struct eval
  : switch_<struct eval_cases>
{};

/////////////////////////////////////
struct eval_cases
{
    template<class Tag>
    struct case_
      : _default<eval>
    {};
};
```

```
////////////////////////////////
// Meta-function for evaluating result type of
// applying a phoenix expression with arguments
template<class Sig>
struct actor_result;


template<class Actor, class A>
struct actor_result<Actor(A)>
  : boost::result_of<eval(Actor &, fusion::vector<A> &)>
{};
```

Building Boost.Phoenix Core

```
//////////////////////////////////
// phoenix expression wrapper
template<class Expr>
struct actor;

//////////////////////////////////
struct phoenix_domain
  : domain<pod_generator<actor> >
{};

//////////////////////////////////
// phoenix grammar and expression
// evaluator
struct eval
  : switch_<struct eval_cases>
{};

//////////////////////////////////
struct eval_cases
{
    template<class Tag>
    struct case_
      : _default<eval>
    {};
};
```

```
/////////////////////////////
// Meta-function for evaluating result type of
// applying a phoenix expression with arguments
template<class Sig>
struct actor_result;

#define BOOST_PROTO_LOCAL_MACRO(N, class_A, A, A_const_ref_a, a)  \
template<class Actor, class_A(N)>                                  \
struct actor_result<Actor(A(N))>                                   \
  : boost::result_of<eval(Actor &, fusion::vector<A(N)> &)>        \
{};

#define  BOOST_PROTO_LOCAL_A          BOOST_PROTO_A
#include BOOST_PROTO_LOCAL_ITERATE()
```

Use Proto macros to automate repetitive coding tasks

Building Boost.Phoenix Core

```
//////////////////////////////
template<class Expr>
struct actor
{
    BOOST_PROTO_EXTENDS(Expr, actor<Expr>, phoenix_domain)



    template<class Sig>
    struct result : actor_result<Sig>
    {};

    template<class A>
    typename boost::result_of<actor<Expr>(A const &)>::type
    operator()(A const & a) const
    {
        BOOST_PROTO_ASSERT_MATCHES(*this, eval);
        fusion::vector<A const &> args(a);
        return eval()(*this, args);
    }

    // ... more overloads
};
```

Also defines operator()

Building Boost.Phoenix Core

```
///////////////////////////////
template<class Expr>
struct actor
{
    BOOST_PROTO_BASIC_EXTENDS(Expr, actor<Expr>, phoenix_domain)
    BOOST_PROTO_EXTENDS_ASSIGN()
    BOOST_PROTO_EXTENDS_SUBSCRIPT()

    template<class Sig>
    struct result : actor_result<Sig>
    {};

    template<class A>
    typename boost::result_of<actor<Expr>(A const &)>::type
    operator()(A const & a) const
    {
        BOOST_PROTO_ASSERT_MATCHES(*this, eval);
        fusion::vector<A const &> args(a);
        return eval()(*this, args);
    }

    // ... more overloads
};
```

```
BOOST_PROTO_EXTENDS(X,Y,Z)
==
BOOST_PROTO_BASIC_EXTENDS(X,Y,Z)
BOOST_PROTO_EXTENDS_ASSIGN()
BOOST_PROTO_EXTENDS_SUBSCRIPT()
BOOST_PROTO_EXTENDS_FUNCTION()
```

DONE

Building Argument Placeholders

```
////////////////////////////////
struct arg_tag {};
```

Aside: terminal<X> == nullary_expr<tag::terminal, X>

```
////////////////////////////////
actor<nullary_expr<arg_tag, mpl::int_<0> >::type> const _1 = {{{}}};
actor<nullary_expr<arg_tag, mpl::int_<1> >::type> const _2 = {{{}}};
actor<nullary_expr<arg_tag, mpl::int_<2> >::type> const _3 = {{{}}};

////////////////////////////////
template<>
struct eval_cases::case_<arg_tag>
  : when<
        nullary_expr<arg_tag, _>
      , at(_state, _value)
    >
{};
```

DONE

```
////////////////////////////////
struct at : callable
{
    template<class Sig>
    struct result;

    template<class This, class Cont, class N>
    struct result<This(Cont &, N &)>
      : fusion::result_of::at<Cont, N>
    {};

    template<class Cont, class N>
    typename fusion::result_of::at<Cont, N>::type
    operator ()(Cont & cont, N) const
    {
        return fusion::at<N>(cont);
    }
};
```

So Far, So Good

```
// evaluate phoenix lambda
int i = ( _1 + 3 ) ( 39 );
assert( i == 42 );


int x[] = { 1, 2, 3, 4 };
int y[] = { 2, 4, 6, 8 };
int z[4];

// Use a Phoenix lambda with std algorithms
std::transform( x, x + 4, y, z, _1 * _2 );
```

So Far, So Good

```cpp
    int x[] = { 1, 2, 3, 4 };
    int y[] = { 2, 4, 6, 8 };
    int z[4];

    // Use a Phoenix lambda with std algorithms
    std::transform( x, x + 4, y, z, _1 * _2 );
}
```

Autos

Name	Value	Type
x	0x0023fa5c	int [4]
y	0x0023fa44	int [4]
z	0x0023fa2c	int [4]
[0]	2	int
[1]	8	int
[2]	18	int
[3]	32	int

Strategy: phoenix::if_(x)[y].else_[z]

- Introduce unique expression tags on which phoenix::eval can dispatch.
- Define if_ and else_ in a new if_else_domain that is a sub-domain of phoenix_domain.
- The generator of the if_else_domain finds complete if_/else_ expressions and makes them children of dummy nodes with our unique expression tags.

Building Phoenix if_/else_

```
//////////////////////////////
struct if_tag {};
struct else_tag {};
struct if_else_tag {};

template<class Expr>
struct if_else_actor;

// Grammar for if_(x)[y]
struct if_stmt
  : subscript<function<terminal<if_tag>, eval>, eval>
{};

// Grammar for if_(x)[y].else_[z]
struct if_else_stmt
  : subscript<
      member<
        unary_expr<if_tag, if_stmt>
        , terminal<else_tag>
      >
    , eval
    >
{};
```

Proto lets us overload operator dot! (Sort of.)

Each complete if_ statement is made a child of a dummy node.

Building Phoenix if_/else_

```
///////////////////////////////
typedef functional::make_expr<if_tag>         make_if;
typedef functional::make_expr<if_else_tag>    make_if_else;
typedef pod_generator<if_else_actor>          make_if_else_actor;

///////////////////////////////
struct if_else_generator
  : or_<
        // if_(this)[that]
        when<
            if_stmt
            // wrap in unary if_tag expr and if_else_actor
          , make_if_else_actor(make_if(_byval(_)))
        >
        // if_(this)[that].else_[other]
      , when<
            if_else_stmt
            // wrap in unary if_else_tag expr and if_else_actor
          , make_if_else_actor(make_if_else(_byval(_)))
        >
      , otherwise<
            make_if_else_actor(_)
        >
    >
{};
```

```
///////////////////////////////
struct if_else_domain
  : domain<if_else_generator, _, phoenix_domain>
{};
```

Grammars can be used as generators

if_else_domain is a sub-domain of phoenix_domain

Building Phoenix if_/else_

```
//////////////////////////////
template<class Expr>
struct if_else_actor
{
    BOOST_PROTO_BASIC_EXTENDS(Expr, if_else_actor<Expr>, if_else_domain)
    BOOST_PROTO_EXTENDS_ASSIGN()
    BOOST_PROTO_EXTENDS_SUBSCRIPT()

    // Declare a member named else_ that is a
    // terminal<if_else_tag> in if_else_domain.
    BOOST_PROTO_EXTENDS_MEMBERS(
        ((else_tag,   else_))
    )

    // ... nested result template and operator() overloads like just actor<Expr>
};
```

Give these expressions a member named “else_” that is a terminal<else_tag> in the if_else_domain

Evaluating Phoenix if_/else_

```
///////////////////////////////
// This tells Proto how to evaluate if_(x)[y]
// statements with if_eval
template<>
struct eval_cases::case_<if_tag>
  : when<
        unary_expr<if_tag, if_stmt>
      , if_eval(_right(_left(_left)), _right(_left), _state)
    >
{};

///////////////////////////////
// This tells Proto how to evaluate if(x)[y].else_[z]
// statements with if_else_eval
template<>
struct eval_cases::case_<if_else_tag>
  : when<
        unary_expr<if_else_tag, if_else_stmt>
      , if_else_eval(
            _right(_left(_left(_left(_left)))))
          , _right(_left(_left(_left))))
          , _right(_left)
          , _state
        )
    >
{};
```

```
struct if_eval : callable
{
    typedef void result_type;

    template<class I, class T, class S>
    void operator()(I const & i, T const & t, S & s)
    {
        if( eval()(i, s) )
            eval()(t, s);
    }
};
```

```
struct if_else_eval : callable
{
    typedef void result_type;

    template<class I, class T, class E, class S>
    void operator()(I const & i, T const & t, E const & e, S & s)
    {
        if( eval()(i, s) )
            eval()(t, s);
        else
            eval()(e, s);
    }
};
```

Wrapping up if_/else_

```
//////////////////////////////
// Define the if_ "function"
typedef functional::make_expr<tag::function, if_else_domain> make_fun;

template<class E>
typename boost::result_of<make_fun(if_tag, E const &)>::type const
if_(E const & e)
{
    return make_fun()(if_tag(), boost::ref(e));
}
```

DONE

Reasons why phoenix::if_ should be a real function:

1. Parens in if_(x) don't mean "apply the lambda"
2. if_ should be find-able via ADL
3. Function templates compile faster than proto terminals

Whew! It works.

```cpp
#include <iostream>
#include <algorithm>
using namespace boost;
using namespace phoenix;

int main()
{
    int x[] = { 1, -2, 3, -4 };

    // Use a Phoenix lambda with std algorithms
    std::for_each( x, x + 4,
        phoenix::if_( _1 > 0 )
        [
          std::cout << _1 << " is positive\n"
        ]
        .else_
        [
          std::cout << _1 << " is non-positive\n"
        ]
    );
}
```

C:\windows\system32\cmd.exe

```
1 is positive
-2 is non-positive
3 is positive
-4 is non-positive
Press any key to continue . . .
```

The Expression Problem, Solved

- Proto lets us add new data types easily
- Proto lets us add new operations easily
- The compiler enforces type-safety for us
- (Extending Proto grammars requires a recompilation, but not code repetition.)

Problems with Placeholders

- Boost.Bind defines ::_1
- Boost.Lambda defines ::boost::lambda::_1
- Boost.Phoenix defines ::boost::phoenix::_1
- Spirit.Qi, Spirit.Karma, Xpressive all have positional placeholders, too.

One Placeholder To Rule Them All

```
//////////////////////////////////
// phoenix expression wrapper
template<class Expr>
struct actor;

//////////////////////////////////
struct phoenix_domain
  : domain<pod_generator<actor>, _, proto::default_domain >
{};

/////////////////////////////
struct arg_tag {};

/////////////////////////////
actor<nullary_expr<arg_tag, mpl::int_<0> >::type> const _1 = {{{}}};
actor<nullary_expr<arg_tag, mpl::int_<1> >::type> const _2 = {{{}}};
actor<nullary_expr<arg_tag, mpl::int_<2> >::type> const _3 = {{{}}};
```

A sub-domain of Proto's default domain is compatible with all other domains

Phoenix Summary

- Core: ~50 LOC
- With placeholders and if/else: ~160 LOC
- Everything is an extension, even the placeholders
- Data structure and algorithm are separate
- Unified placeholders!

Better Diagnostics

Avoiding cascading
compile-time errors

A Common Scenario:

- You define a grammar with transforms
- You define an evaluator function that
  - … asserts an expression matches the grammar
  - … passes the expression to the grammar’s transform

Problem: If the exception doesn’t match the grammar, you can’t legally apply its transform. Doing so causes cascading errors.

Cascading Errors: Example

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

struct inc : callable
{
    typedef int & result_type;
    int & operator()(int & i) const
    {
        return ++i;
    }
};

// accept any expression with only integer terminals
// and increments all the integers
struct Inc
  : or_<
        when<terminal<int>, inc(_value)>
      , when<
            nary_expr<_, vararg<Inc> >
          , fold<_, _state, Inc>
        >
    >
{};
```

```
template<typename E>
void eval(E const & e)
{
    BOOST_PROTO_ASSERT_MATCHES(e, Inc);
    // Oops, we try to compile this, too. Likely to fail.
    Inc()(e);
}

int main()
{
    literal<int> i(0), j(1), k(2);
    eval(i + j + k); // OK, i==1, j==2, k==3

    literal<float> f(3.14f);
    eval(i + j + f); // OOPS
}
```

Cascading Errors: Example

```
#includ
using n
using n

struct i
{
    typed
    int &
    {
        re
    }
};

// accep
// and i
struct I
  : or_<
        w
      , w

          ,
        >
    >
{};
```

```
1>------ Build started: Project: scratch, Configuration: Debug Win32 ------
1>Compiling...
1>main.cpp
1>c:\boost\org\trunk\boost\proto\transform\fold.hpp(276) : error C2039: 'result_type' : is not a member of
'boost::proto::control::or_<G0,G1>::impl<Expr,State,Data>'
1>        with
1>        [
1>            G0=boost::proto::when<boost::proto::op::terminal<int>,inc (boost::proto::_value)>,
1>
G1=boost::proto::when<boost::proto::op::nary_expr<boost::proto::wildcardns_::_,boost::proto::control::vararg<Inc>>,bo
ost::proto::fold<boost::proto::wildcardns_::_,boost::proto::_state,Inc>>
1>        ]
1>        and
1>        [
1>            Expr=boost::proto::utility::literal<float> &,
1>            State=int &,
1>            Data=int
1>        ]
1>        c:\boost\org\trunk\boost\proto\transform\fold.hpp(232) : see reference to class template instantiation
'boost::proto::detail::fold_impl<State0,Fun,Expr,State,Data>' being compiled
1>        with
1>        [
1>            State0=boost::proto::_state,
1>            Fun=Inc,
1>            Expr=const boost::proto::exprns_::expr<boost::proto::tag::plus,boost::proto::argsns_::list2<const
boost::proto::exprns_::expr<boost::proto::tag::plus,boost::proto::argsns_::list2<boost::proto::utility::literal<int>
&,boost::proto::utility::literal<int> &>,2> &,boost::proto::utility::literal<float> &>,2> &,
….
1>scratch - 8 error(s), 0 warning(s)
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Huh?

Cascading Errors: Example

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

struct inc : callable
{
    typedef int & result_type;
    int & operator()(int & i) const
    {
        return ++i;
    }
```

Use proto::matches to dispatch either to a function that does the work, or a stub that issues an error.

```
    , when<
          nary_expr<_, vararg<Inc> >
        , fold<_, _state, Inc>
      >
  >
{};
```

```
template<typename E>
void eval2(E const & e, mpl::true_)
{
    Inc()(e);
}

template<typename E>
void eval2(E const & e, mpl::false_)
{
    BOOST_MPL_ASSERT_MSG( (false)
        , LIKE_UM_TOTALLY_INVALID_EXPRESSION_DUDE
        , (E) );
}

template<typename E>
void eval(E const & e)
{
    eval2(e, matches<E, Inc>());
}

int main()
{
    literal<int> i(0), j(1);
    literal<float> f(3.14f);
    eval(i + j + f); // OOPS
}
```

Cascading Errors: Example

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

struct
{
    type
    int
    {
        r
    }
  d
  th
    , w
    >
  >
{};
```

```
template<typename E>
void eval2(E const & e, mpl::true_)
{
                                          N_DUDE

    eval(i + j + f); // OOPS
}
```

```
1>------ Build started: Project: scratch, Configuration: Debug Win32 ------
1>Compiling...
1>main.cpp
1>c:\boost\org\trunk\libs\proto\scratch\main.cpp(35) : error C2664: 'boost::mpl::assertion_failed' : cannot convert parameter 1 from 'boost::mpl::failed ************(__thiscall eval2::LIKE_UM_TOTALLY_INVALID_EXPRESSION_DUDE::* ***********)(E)' to 'boost::mpl::assert<false>::type'
1>        with
1>        [
1>            E=...
1>        ]
1>        No constructor could take the source type, or constructor overload resolution was ambiguous
1>        c:\boost\org\trunk\libs\proto\scratch\main.cpp(45) : see reference to function template instantiation 'void eval2<E>(const E &,boost::mpl::false_)' being compiled
1>        with
1>        [
1>            E=…
1>        ]
1>        c:\boost\org\trunk\libs\proto\scratch\main.cpp(54) : see reference to function template instantiation 'void eval<boost::proto::exprns_::expr<Tag,Args,Arity>>(const E &)' being compiled
1>        with
1>        […
1>        ]
1>Build log was saved at "file://c:\boost\org\trunk\libs\proto\scratch\Debug\BuildLog.htm"
1>scratch - 1 error(s), 0 warning(s)
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Cascading Errors: Take-Aways

- Use proto::matches to dispatch to stub functions on error.
- Use MPL assertions to generate errors in the right place (your code, not Proto's) with the right message.
- Consider adding a comment just before the MPL assertion:
  - /* If your compile breaks here, it probably means *<yadda yadda yadda>* */

Proto by Doing: Summary

- Build libraries with rich user interfaces:
  - quickly
  - with less code
  - with stricter type checking
  - with clean separation between data and algo
  - and better diagnostics
- Proto is *not* (that) scary
- Proto is useful for small DSELs as well as big ones

Questions?