Author's Corner: Boost.Proto

Or, "A Young Person's Guide to Domain-Specific Embedded Languages"

Why Boost.Proto?

- The Year: 2005
- The Problem: Xpressive development stall
- The Solution: Work on a different problem!

Boost Proto,

Born April 20, 2005

Proto Design Goals

- Out with the bad ...
  - ☒ Expression template spaghetti code
  - ☒ Low-level, verbose template hackery
- In with the good ...
  - ☑ Clean separation of ET data structure and algorithms
  - ☑ Concise high-level description of target domain

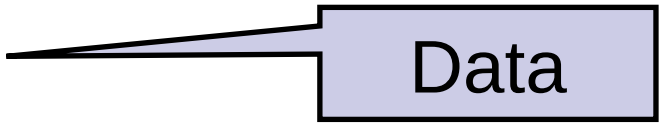
Proto Terms and Concepts

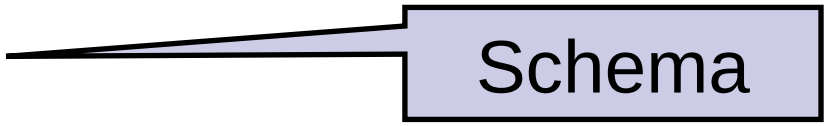
- Domain-Specific Embedded Language
  - *Language:*
    - symbols, grammar, semantics
  - *Domain-Specific:*
    - simplified grammar, increased expressiveness in narrow domain
  - *Embedded:*
    - implemented as a library in a host language

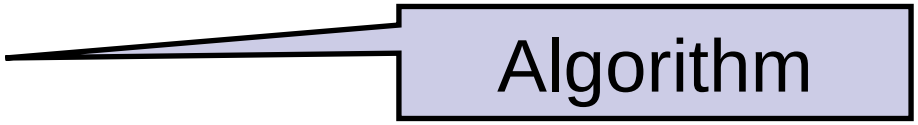
Example: Boost.Spirit, Boost.Lambda, Blitz++, etc.

Proto Terms and Concepts

- Expression
  - Symbols in a tree
- Grammar
  - Describes valid trees within a domain
- Transform
  - Operates on expression trees







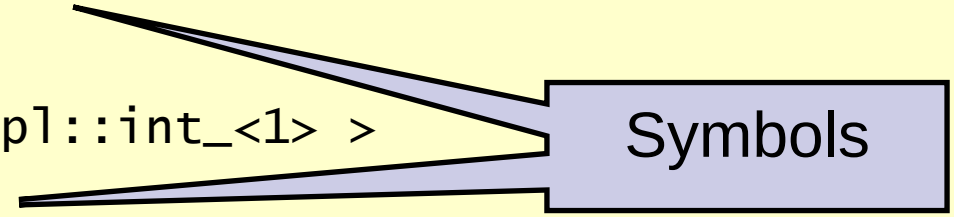
Example: Mini-Lambda Library

```
// #includes ...

// A type for numbered placeholders
template<class Int>
struct placeholder : Int
{};

// Numbered placeholders as Proto terminals
proto::terminal<
    placeholder< mpl::int_<0> >
>::type _1;

proto::terminal<
    placeholder< mpl::int_<1> >
>::type _2;
```



Example: Mini-Lambda Library

```
// A TR1-style function object that calls fusion::at()
struct at : proto::callable
{
    template<class Sig> struct result;

    template<class This, class Cont, class Int>
    struct result<This(Cont, Int)>
      : fusion::result_of::at<
            typename boost::remove_reference<Cont>::type
          , typename boost::remove_reference<Int>::type   >
    {};

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

Example: Mini-Lambda Library

```
// A Proto transform that evaluates a lambda expression
struct LambdaEval
  : proto::or_<
        proto::when<
            proto::terminal<placeholder<proto::_> >
          , at(proto::_data, proto::_value)
        >
      , proto::when<
            proto::_
          , proto::_default<LambdaEval>
        >
    >
{};
```

Algorithm

Example: Mini-Lambda Library

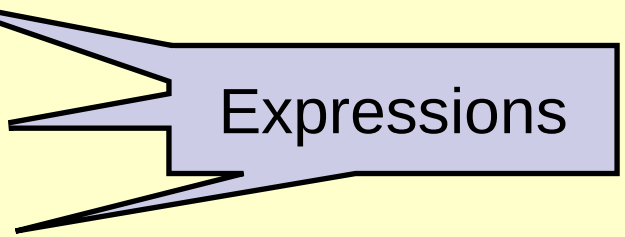
```
// Use the LambdaEval transform to evaluate
// lambda expressions.

int main()
{
    // The values of _1 and _2:
    fusion::tuple<int, int> tup(2,3);

    int j = LambdaEval()( _2 - _1, 0, tup );
    assert( j == (3 - 2) );

    int k = 42;
    LambdaEval()( k *= _1, 0, tup );
    assert( k == (42 * 2) );

    LambdaEval()( _1 += 4, 0, tup );
    assert( 6 == fusion::at_c<0>(tup) );
}
```



Expressions

```
_2 - _1
```

```
expr<
    tag::minus
  , list2<
        expr<tag::terminal, term< placeholder<...> > > &
      , expr<tag::terminal, term< placeholder<...> > > &
    >
>
```

- Tagged tree, built with operator overloads
- Abstract, no domain-specific information here

Grammars

```
struct Arith;
struct Plus  : proto::plus< Arith, Arith > {};
struct Minus : proto::minus< Arith, Arith > {};
struct Mult  : proto::multiplies< Arith, Arith > {};
struct Div   : proto::divides< Arith, Arith > {};
struct Term  : proto::terminal< proto::_ > {};

struct Arith
  : proto::or_<Plus, Minus, Mult, Div, Term>
{};
```

- Evaluate with `proto::matches<MyExpr, Arith>`
- Expression validation, introspection
- Basis for defining transforms
- Also useful for controlling Proto's operator overloading

Grammars: Historical Note

- Invented in Nov. 2006 on spirit-devel list. Given ...

```
struct CharT {};
proto::terminal<CharT>::type char_;
```

- ...write overloaded `parse()` functions that accept:

Expression	Grammar
`char_`	`terminal<CharT>`
`char_('a')`	`function<terminal<CharT>,_>`
`char_('a','z')`	`function<terminal<CharT>,_,_>`

```
template< class Expr >
typename enable_if<
    proto::matches< Expr, function<terminal<CharT>,_,_> >
>::type
parse( Expr const & expr ) { ... }
```

Transforms

```
struct LambdaEval
  : proto::or_<
        proto::when<
            proto::terminal<placeholder<proto::_> >
          , at(proto::_data, proto::_value)
        >
      , proto::when< proto::_, proto::_default<LambdaEval> >
    >
{};
```

- Transforms == grammars decorated with actions
- Ternary function object
  1. Expression
  2. Initial state, used by some Proto transforms (`fold`)
  3. Auxilliary data, used for anything you like

Transforms: Historical Note

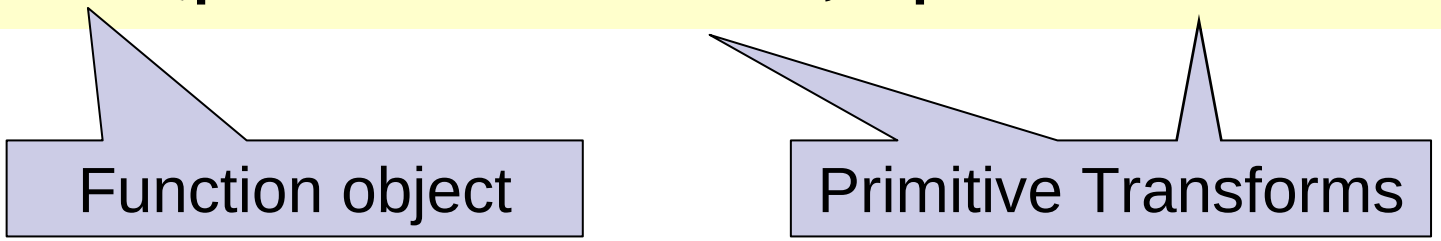
- Invented Dec. 2006 on spirit-devel list
- Modeled on Spirit's semantic actions
  - (which are modeled on YACC's)
- Replaced earlier Proto "compilers"
  - (which sucked)
- Round-lambda syntax came Jan. 2008
  - (modeled on Aleksey's MPL round lambdas)

Round Lambda Notation

- Common semantic actions
  - Create and initialize an object: Obj(x)
  - Call a function: Fun(x)
- A Few Observations ...
  - "Fun" can be the type of a function object
  - If "x" is a type, so are Obj(x) and Fun(x)
- Function type == composite transform
- Object transforms vs. Callable transforms
  - proto::is_callable<> trait

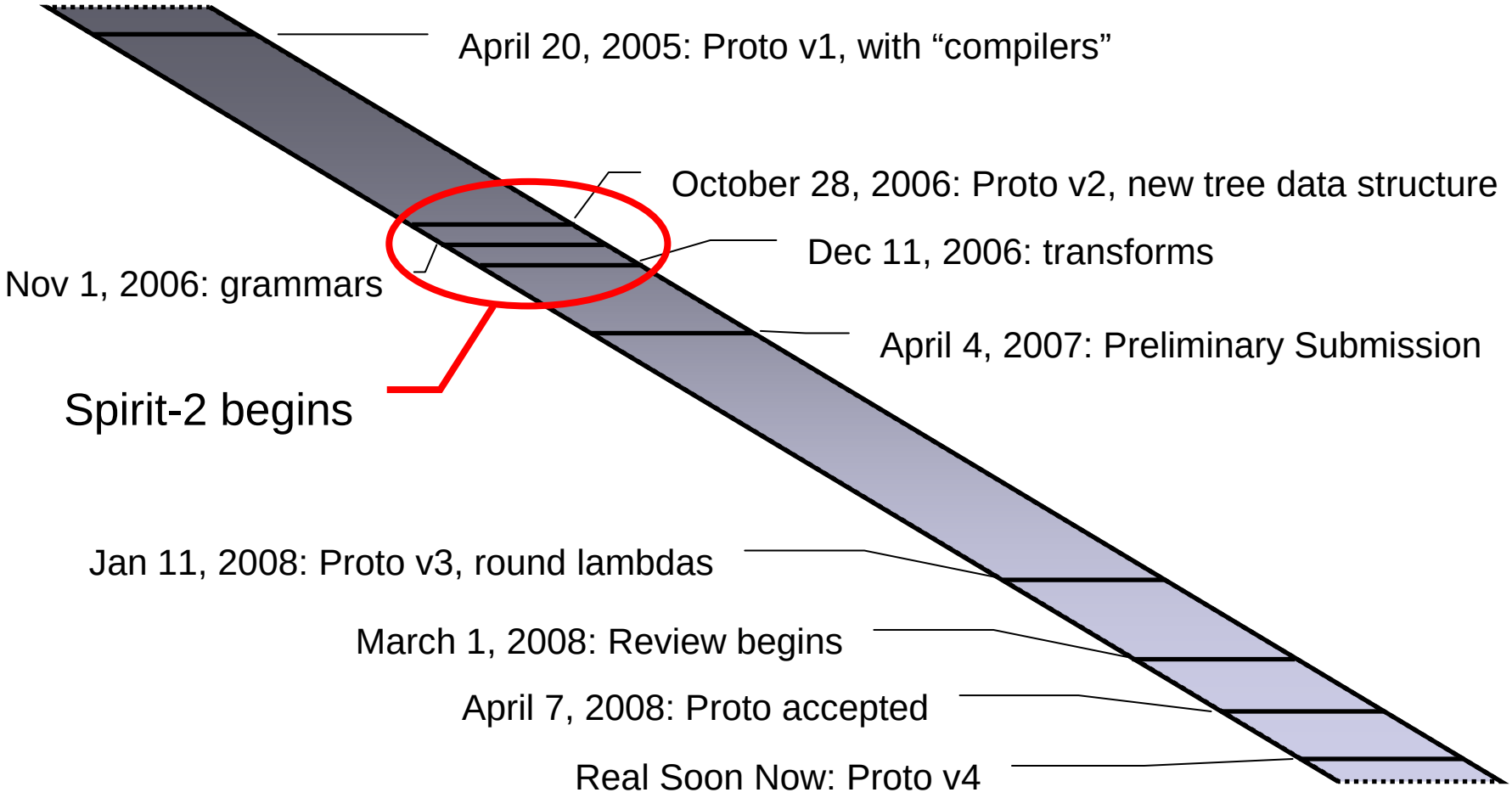
Round Lambda Notation

```
at(proto::_data, proto::_value)
```



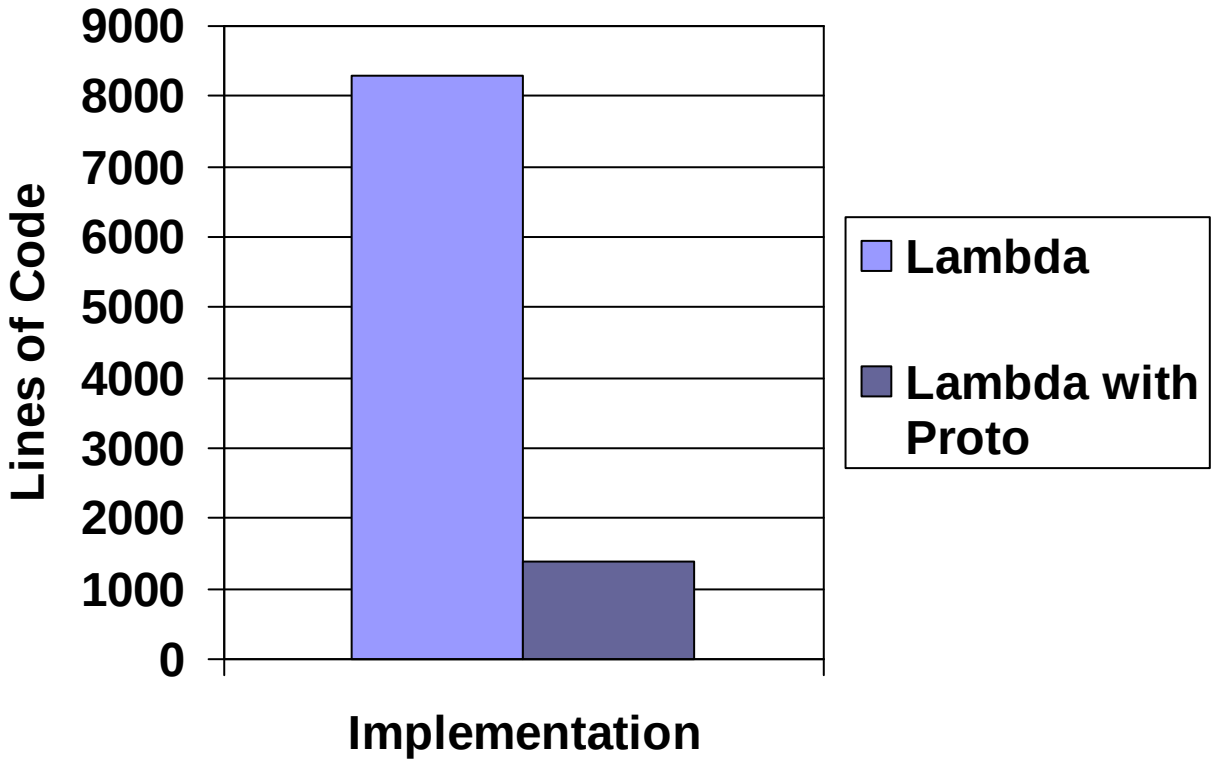

```
struct _data : proto::transform<_data>
{
    template<class Expr, class State, class Data>
    struct impl : proto::transform_impl<Expr, State, Data>
    {
        typedef Data result_type;
        Data operator()( typename impl::expr_param  expr
                       , typename impl::state_param state
                       , typename impl::data_param  data ) const
        {
            return data;
        }
    };
};
```

Proto Evolutionary Timeline



Surprise!

- I've successfully ported the Boost Lambda Library to Proto



Questions?