

# Nested Lambda Expressions with Let expressions in C++ Template Metaprograms

Ábel Sinkovics,  
ELTE, Budapest

*The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003)*

# Outline

- Let expressions
- Template metaprogramming
- Let expressions in template metaprograms
- Lambda expressions
- Recursive let expressions

# Let expressions

- Example
  - Restaurant
  - Happy Hours

# Let expressions

```
burgerPrice hourOfDay =
```

```
  if hourOfDay < 20  
    then 5  
    else 4
```

# Let expressions

```
burgerPrice hourOfDay =  
  let  
    happyHoursBegin = 20  
    happyHoursPrice = 4  
    normalPrice = 5  
  in  
    if hourOfDay < 20  
      then 5  
      else 4
```

# Let expressions

```
burgerPrice hourOfDay =  
  let  
    happyHoursBegin = 20  
    happyHoursPrice = 4  
    normalPrice = 5  
  in  
    if hourOfDay < happyHoursBegin  
      then normalPrice  
      else happyHoursPrice
```

# Let expressions

```
divMod a b = (a `div` b, a `mod` b)
```

```
-- divMod 7 3
```

```
-- (2, 1)
```

```
let
```

```
  (d, m) = divMod 7 3
```

```
in
```

```
  d * 3 + m
```

# Let expressions

`divMod(A, B) -> {A div B, A rem B}.`

`% divMod(7, 3).`

`% {2, 1}.`

`{D, M} = divMod(7, 3),  
D * 3 + M.`



# C++ template metaprograms

- Code evaluated at compilation time
- Erwin Unruh, 1994.
- Strong connection with functional programming
- Library support: Boost metaprogramming library
- Turing-complete

# C++ template metaprograms

```
template <class T>  
struct makeConst  
{  
    typedef const T type;  
};
```

Argument list

Name

Body

```
makeConst<int>::type
```

# C++ template metaprograms

```
template <class T>  
struct makeConst  
{  
    typedef const T type;  
};
```

Argument list

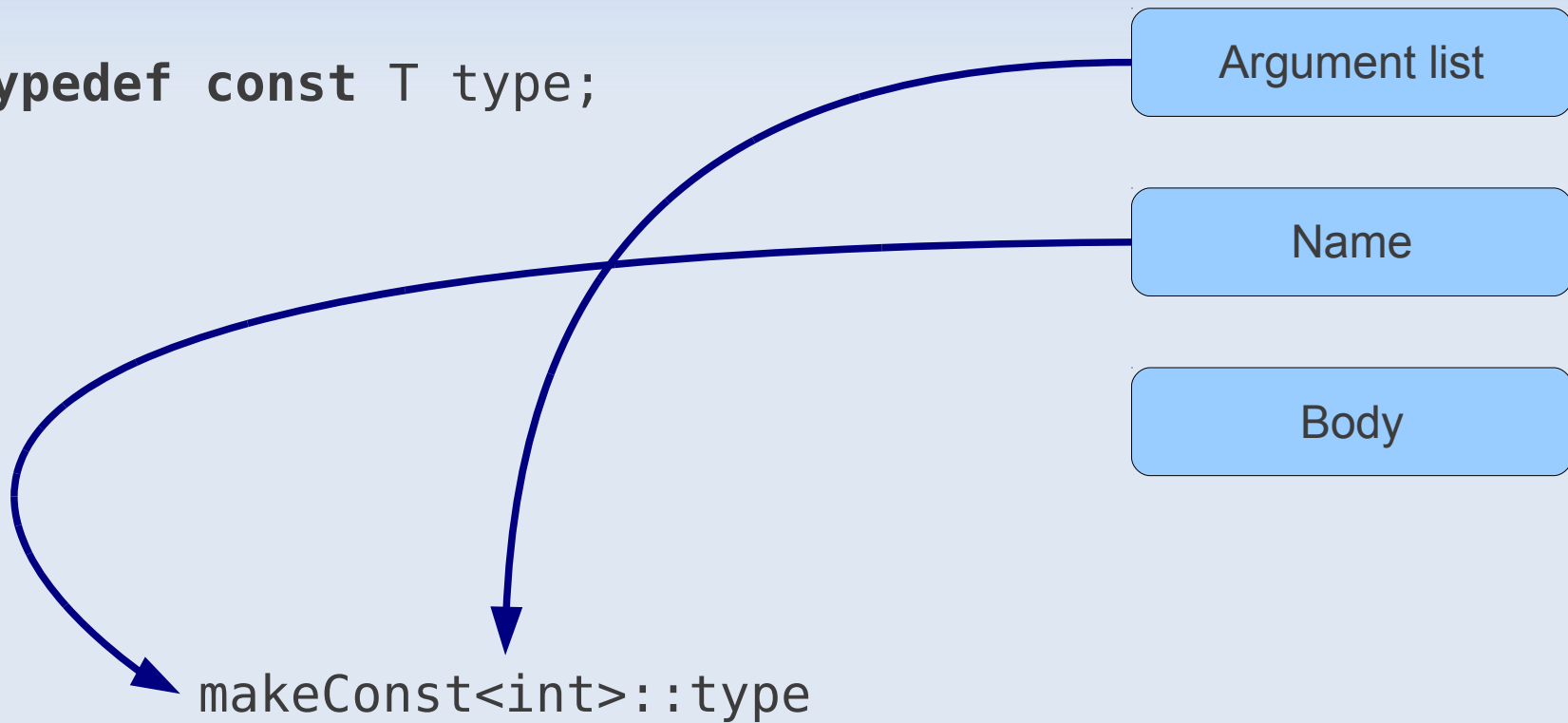
Name

Body

makeConst<int>::type

# C++ template metaprograms

```
template <class T>  
struct makeConst  
{  
    typedef const T type;  
};
```



# Template metafunction class

```
struct makeConst
{
    template <class T>
    struct apply
    {
        typedef const T type;
    };
};
```

Argument list

Name

Body

```
makeConst::apply<int>::type
```

# Template metafunction class

```
struct makeConst  
{  
    template <class T>  
    struct apply  
    {  
        typedef const T type;  
    };  
};
```

Argument list

Name

Body

makeConst::apply<int>::type

# Template metafunction class

```
struct makeConst
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };
};
```

Argument list

Name

Body

makeConst::apply<int>::type

# Let expressions

```
burgerPrice hourOfDay =  
  let  
    happyHoursBegin = 20  
    happyHoursPrice = 4  
    normalPrice = 5  
  in  
    if hourOfDay < happyHoursBegin  
      then normalPrice  
      else happyHoursPrice
```



# Let expressions

```
template <                >  
struct burgerPrice :
```

```
{};
```

# Let expressions

```
template <class hourOfDay>  
struct burgerPrice :
```

```
{};
```

# Let expressions

```
template <class hourOfDay>  
struct burgerPrice :
```

```
    boost::mpl::if_<
```

```
    >
```

```
{};
```

# Let expressions

```
template <class hourOfDay>  
struct burgerPrice :
```

```
    boost::mpl::if_ <  
        boost::mpl::less<hourOfDay,          20 > ,
```

```
>
```

```
{};
```

# Let expressions

```
template <class hourOfDay>  
struct burgerPrice :
```

```
    boost::mpl::if_ <  
        boost::mpl::less<hourOfDay, boost::mpl::int_<20> > ,
```

```
>
```

```
{};
```

# Let expressions

```
template <class hourOfDay>  
struct burgerPrice :
```

```
    boost::mpl::if_ <  
        boost::mpl::less<hourOfDay, boost::mpl::int_<20> >,  
        boost::mpl::int_<5>,  
        boost::mpl::int_<4>  
    >
```

```
{};
```

# Let expressions

```
typedef boost::mpl::int_<20> happyHoursBegin;  
typedef boost::mpl::int_<4> happyHoursPrice;  
typedef boost::mpl::int_<5> normalPrice;
```

```
template <class hourOfDay>  
struct burgerPrice :
```

```
    boost::mpl::if_<  
        boost::mpl::less<hourOfDay, happyHoursBegin    >,  
        normalPrice    ,  
        happyHoursPrice  
    >
```

```
{};
```

# Let expressions

```
template <class hourOfDay>
struct burgerPrice :
    LET
    happyHoursBegin = boost::mpl::int_<20>,
    happyHoursPrice = boost::mpl::int_<4>,
    normalPrice = boost::mpl::int_<5>
    IN
    boost::mpl::if_<
        boost::mpl::less<hourOfDay, happyHoursBegin    >,
        normalPrice    ,
        happyHoursPrice
    >
{};
```



# Let expressions

```
struct happyHoursBegin;
struct happyHoursPrice;
struct normalPrice;

template <class hourOfDay>
struct burgerPrice :
    let<
        happyHoursBegin, boost::mpl::int_<20>,
        happyHoursPrice, boost::mpl::int_<4>,
        normalPrice, boost::mpl::int_<5>,

        boost::mpl::if_<
            boost::mpl::less<hourOfDay, happyHoursBegin >,
            normalPrice,
            happyHoursPrice
        >
    >
    >
    {};
```

# Nullary metafunctions

```
struct nullary_metafunction  
{  
    typedef int type;  
};
```

# Nullary metafunctions

```
struct nullary_metafunction
{
    typedef int type;
};

some_metafunction<int, double>
```

# Nullary metafunctions

```
struct nullary_metafunction  
{  
    typedef int type;  
};
```

```
some_metafunction<int, double>
```

```
nullary_metafunction::type  
some_metafunction<int, double>::type
```

# Let expressions

```
struct happyHoursBegin;  
struct happyHoursPrice;  
struct normalPrice;
```

```
template <class hourOfDay>  
struct burgerPrice :
```

```
    let<  
        happyHoursBegin, boost::mpl::int_<20>,  
        happyHoursPrice, boost::mpl::int_<4>,  
        normalPrice, boost::mpl::int_<5>,  
        |
```

```
        boost::mpl::if_ <  
            boost::mpl::less<hourOfDay, happyHoursBegin >,  
            normalPrice  
            ,  
            happyHoursPrice  
        >
```

```
>
```

```
{};
```

Nullary metafunction



# Implementation of let

```
template <class name, class exp, class body>  
struct let  
    {};
```

# Implementation of let

```
let<  
  x, boost::mpl::int_<13>,  
  boost::mpl::plus<x, x>  
>::type
```

```
template <class name, class exp, class body>  
struct let  
    {};
```

# Implementation of let

```
let<  
  x, boost::mpl::int_<13>,  
  boost::mpl::plus<x, x>  
>::type
```



```
boost::mpl::plus<  
  boost::mpl::int_<13>,  
  boost::mpl::int_<13>  
>
```

```
template <class name, class exp, class body>  
struct let  
    {};
```



# Implementation of let

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

name is not the same as body



```
template <class name, class exp, class body>  
struct let_impl  
    {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
let_impl<
  x, boost::mpl::int_<13>,
  x
>
```

name is not the same as body



```
template <class name, class exp, class body>
struct let_impl          {};
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

# Implementation of let



```
let_impl<
  x, boost::mpl::int_<13>,
  x
>
```

name is not the same as body



```
template <class name, class exp, class body>
struct let_impl          {};
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
template <class name, class exp, class body>  
struct let_impl : body {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
let<  
  x, boost::mpl::int_<13>,  
  boost::mpl::int_<11>  
>::type
```

```
template <class name, class exp, class body>  
struct let_impl : body {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
let<  
  x, boost::mpl::int_<13>,  
  boost::mpl::int_<11>  
>::type
```

```
boost::mpl::int_<11>::type
```



```
template <class name, class exp, class body>  
struct let_impl : body {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
let<  
  x, boost::mpl::int_<13>,  
  boost::mpl::int_<11>  
>::type
```

~~boost::mpl::int\_<11>::type~~

boost::mpl::int\_<11>

```
template <class name, class exp, class body>  
struct let_impl : body {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```



# Implementation of let

```
template <class name, class exp, class body>  
struct let_impl : id<body> {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
template <class body>
struct id
{
    typedef body type;
};
```

```
template <class name, class exp, class body>
struct let_impl : id<body> {};
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
let<
  x, boost::mpl::int_<13>,
  boost::mpl::plus<x, x>
>::type
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
template <  
    class name,  
    class exp,
```

```
>  
struct let_impl<name, exp, > :
```

```
    {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

```
let<  
    x, boost::mpl::int_<13>,  
    boost::mpl::plus<x, x>  
>::type
```

# Implementation of let

```
template <
  class name,
  class exp,
  template <class, class> class t,
>
struct let_impl<name, exp, t<
  > > :
  {};
```

```
let<
  x, boost::mpl::int_<13>,
  boost::mpl::plus<x, x>
>::type
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

# Implementation of let

```
template <
  class name,
  class exp,
  template <class, class> class t,
  class a1,
  class a2
>
struct let_impl<name, exp, t<a1, a2> > :

    {};
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

```
let<
  x, boost::mpl::int_<13>,
  boost::mpl::plus<x, x>
>::type
```

# Implementation of let

```
template <
  class name,
  class exp,
  template <class, class> class t,
  class a1,
  class a2
>
struct let_impl<name, exp, t<a1, a2> > :
  id<
    t<
      a1,
      a2
    >
  > {};
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

```
let<
  x, boost::mpl::int_<13>,
  boost::mpl::plus<x, x>
>::type
```

# Implementation of let

```
template <
  class name,
  class exp,
  template <class, class> class t,
  class a1,
  class a2
>
struct let_impl<name, exp, t<a1, a2> > :
  id<
    t<
      typename let<name, exp, a1>::type,
      typename let<name, exp, a2>::type
    >
  > {};
```

```
template <class name, class exp, class body>
struct let : let_impl<name, exp, body> {};
```

```
let<
  x, boost::mpl::int_<13>,
  boost::mpl::plus<x, x>
>::type
```



# Implementation of let

```
template <class name, class exp>  
struct let<name, exp, name> : id<exp> {};
```

```
template <class name, class exp, class body>  
struct let : let_impl<name, exp, body> {};
```

# Name shadowing

```
let
  x = 11
in
  x + (let x = 13 in x + 11)
```

# Name shadowing

```
let  
  x = 11  
in  
  x + (let x = 13 in x + 11)
```

11 + (let x = 13 in x + 11)

# Name shadowing

```
let  
  x = 11  
in  
  x + (let x = 13 in x + 11)
```

11 + (let x = 13 in x + 11)

11 + (                      13 + 11)

# Name shadowing

```
let  
  x = 11  
in  
  x + (let x = 13 in x + 11)
```

```
graph TD; A((11)) --> B[11 + (let x = 13 in x + 11)]; B --> C[11 + (13 + 11)];
```

```
11 + (let x = 13 in x + 11)
```

```
11 + (13 + 11)
```

# Name shadowing

```
let<  
  x, int_<11>,  
>
```

# Name shadowing

```
let<  
  x, int_<11>,  
  plus<x,                                     >  
>
```

# Name shadowing

```
let<
  x, int_<11>,
  plus<x, let<x, int_<13>,
> >
>
```



# Name shadowing

```
let<  
  x, int_<11>,  
  plus<x, let<x, int_<13>, plus<x, int_<11> > > >  
>
```

# Name shadowing

```
let<
  x, int_<11>,
  plus<x, let<x, int_<13>, plus<x, int_<11> > > >
>
```

```
plus<
  int_<11>,
  let<int_<11>, int_<13>, plus<int_<11>, int_<11> > >
>
```

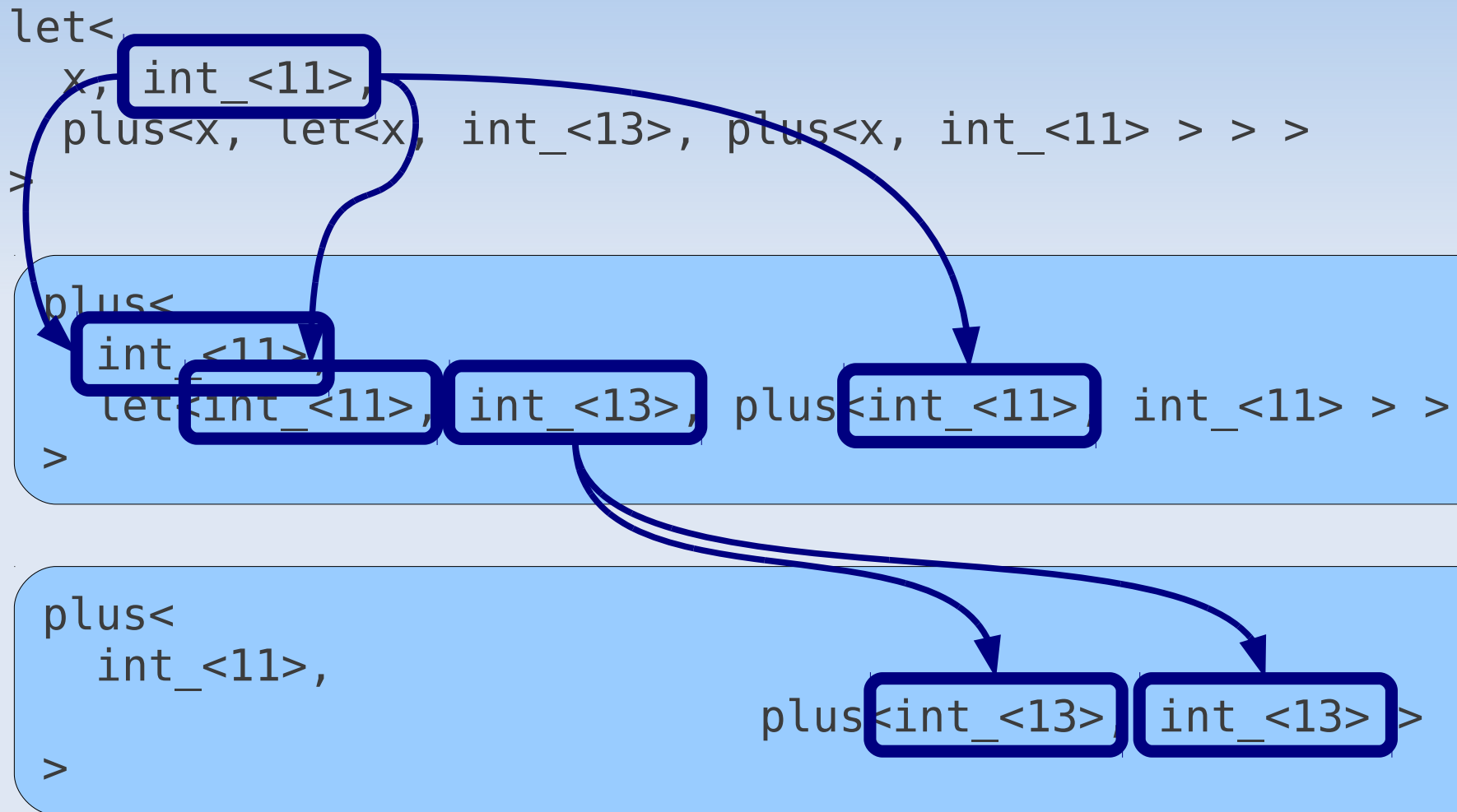
# Name shadowing

```
let<
  x, int_<11>,
  plus<x, let<x, int_<13>, plus<x, int_<11> > > >
>
```

```
plus<
  int_<11>,
  let<int_<11>, int_<13>, plus<int_<11>, int_<11> > >
>
```

```
plus<
  int_<11>,
  plus<int_<13>, int_<13> >
>
```

# Name shadowing



# Name shadowing

```
let<
```

```
template <  
  class name,  
  class exp1,  
  class exp2,  
  class body
```

```
>
```

```
struct let_impl<name, exp1, let<name, exp2, body> > :  
  id<let<name, exp2, body> >  
{};
```

```
int_<11>,
```

```
plus<int_<13>, int_<13> >
```

```
>
```

# Let expressions

- Examples of let expressions
- Let expressions in C++ template metaprograms

```
let<  
  x, boost::mpl::plus<a, b>,  
  boost::mpl::mult<x, x>  
>
```

# Nested lambda expressions

- List of lists

```
[  
  [ 1, 2],  
  [ 3]  
]
```

# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```



# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```

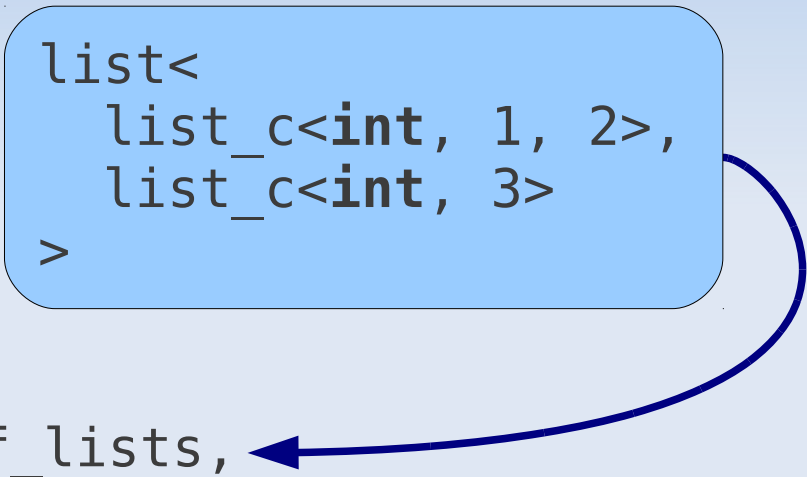
- Add every element the length of the list it is in.

```
list<
  list_c<int, 3, 4>,
  list_c<int, 4>
>
```

# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```



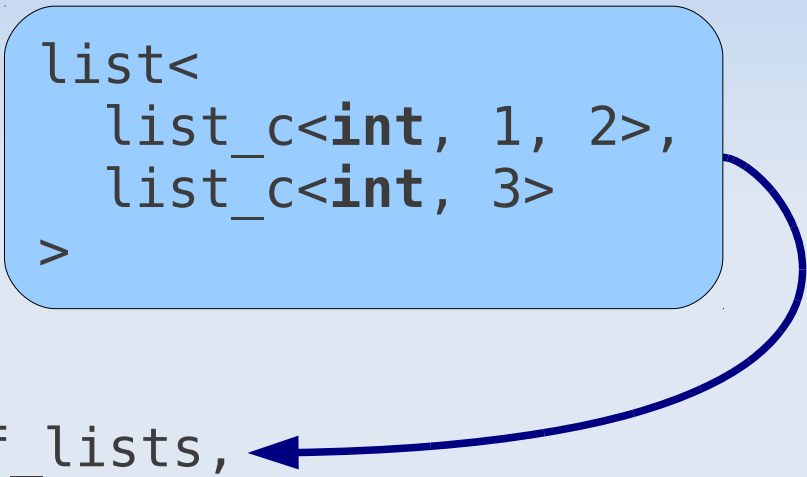
```
transform<
  original_list_of_lists,
  lambda<
```

```
>
>
```

# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```



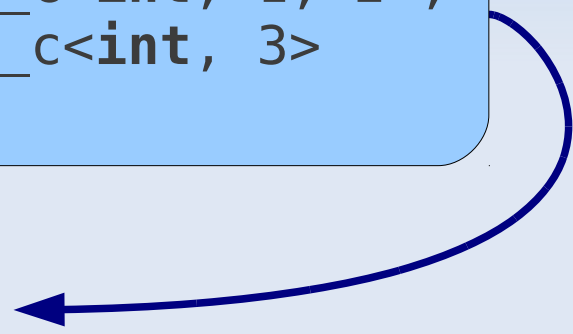
```
transform<
  original_list_of_lists,
  lambda<
    transform<
      1
      lambda<
        >
      >
    >
  >
  >
  >
```

# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```

```
transform<
  original_list_of_lists,
  lambda<
    transform<
      1,
      lambda<
        plus<
          >,
        >
      >
    >
  >
>
```

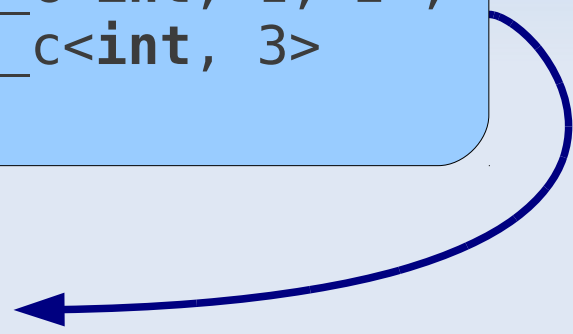


# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```

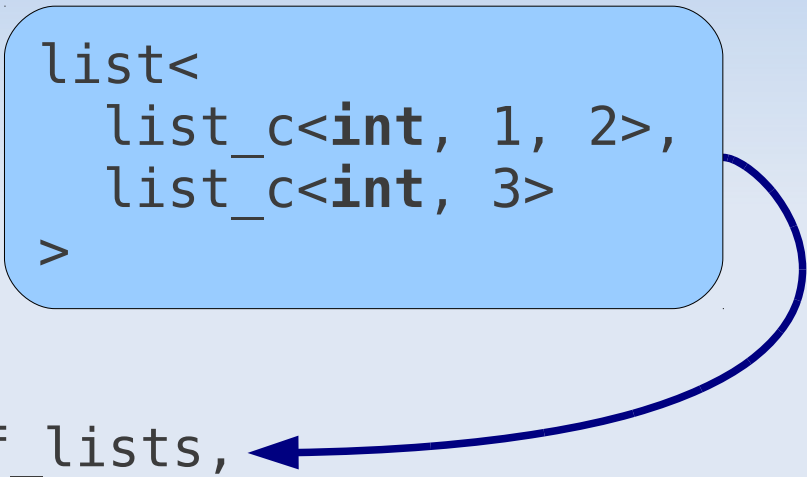
```
transform<
  original_list_of_lists,
  lambda<
    transform<
      _1,
      lambda<
        plus<_1,
      >,
    >,
  >,
>
```



# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```



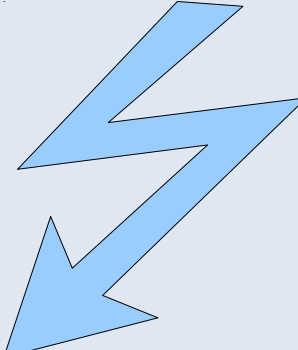
```
transform<
  original_list_of_lists,
  lambda<
    transform<
      _1,
      lambda<
        plus<_1, size<
          > >
        >
      >
    >
  >
>
```

# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```

```
transform<
  original_list_of_lists,
  lambda<
    transform<
      _1,
      lambda<
        plus<_1, size<
          >>
        >
      >
    >
  >
  >
  >
  >
```

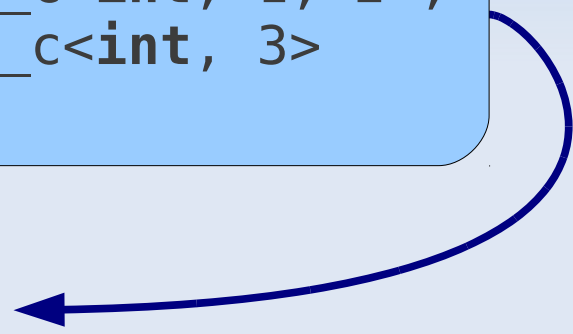


# Nested lambda expressions

- List of lists

```
list<
  list_c<int, 1, 2>,
  list_c<int, 3>
>
```

```
transform<
  original_list_of_lists,
  our::lambda<current_list,
  transform<
    current_list,
    our::lambda<current_item,
      plus<current_item, size<current_list> >
    >
  >
  >
  >
  >
```





# Building our lambda

```
template <class name, class body>  
struct lambda  
{  
  
};
```

# Building our lambda

```
template <class name, class body>
struct lambda
{
    template <class exp>
    struct apply : {}
};
```

# Building our lambda

```
template <class name, class body>
struct lambda
{
    template <class exp>
    struct apply : let<name, exp, body>::type {};
};
```

# Let and lambda expressions

- Examples of let expressions
- Let expressions in C++ template metaprograms
- Nested lambda expressions

# Recursive let expressions

- Factorial

```
let<  
  fact,  
  lambda<
```

```
  >,  

```

```
>
```

# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<

    >
  >,
>
```

# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<
      equal_to<n, int_<0> >,

    >
  >,
>
```

# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<
      equal_to<n, int_<0> >,
      int_<1>,
    >
  >,
>
```



# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<
      equal_to<n, int_<0> >,
      int_<1>,
      times<
        , n>
      >
    >,
  >
>
```

# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<
      equal_to<n, int_<0> >,
      int_<1>,
      times<apply<fact,          >, n>
    >
  >,
>
```

# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<
      equal_to<n, int_<0> >,
      int_<1>,
      times<apply<fact, minus< , > >, n>
    >
  >,
>
```

# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<
      equal_to<n, int_<0> >,
      int_<1>,
      times<apply<fact, minus<n, int_<1> > >, n>
    >
  >,
>
```

# Recursive let expressions

- Factorial

```
let<
  fact,
  lambda< n,
    lazy_eval_if<
      equal_to<n, int_<0> >,
      int_<1>,
      times<apply<fact, minus<n, int_<1> > >, n>
    >
  >,
  apply<fact, int_<3> >
>
```

# Recursive let expressions

```
apply<
```

```
  fact,
```

```
  int_<3>
```

```
>
```

# Recursive let expressions

apply<

```
lambda< n,  
  lazy_eval_if<  
    equal_to<n, int_<0> > ,  
    int_<1> ,  
    times<apply<fact, minus<n, int_<1> > > , n>  
  >  
> ,  
  int_<3>  
>
```

# Recursive let expressions

```
apply<
  let<
    fact,
    lambda< n,
      lazy_eval_if<
        equal_to<n, int_<0> >,
        int_<1>,
        times<apply<fact, minus<n, int_<1> > >, n>
      >
    >,
    lambda< n,
      lazy_eval_if<
        equal_to<n, int_<0> >,
        int_<1>,
        times<apply<fact, minus<n, int_<1> > >, n>
      >
    >
  >,
  int_<3>
>
```



# Recursive let expressions

```
template <class name, class exp, class body>  
struct letrec :  
  
{};
```

# Recursive let expressions

```
template <class name, class exp, class body>  
struct letrec :  
    let<name,                exp , body>  
{};
```

# Recursive let expressions

```
template <class name, class exp, class body>  
struct letrec :  
    let<name, letrec<name, exp, exp>, body>  
{};
```

# Summary

- Let expressions are widely available
- Many functional programming languages support them
- No native support for it in C++ Template Metaprogramming
- It can be built as a library
- Using it, Boost MPL's lambda support can be improved

# Q & A

Ábel Sinkovics  
[abel@sinkovics.hu](mailto:abel@sinkovics.hu)

<http://github.com/sabel83/mpllibs>