Kleene Parser Implementation

```
template <typename Iterator, typename Context, typename Attribute>
    static bool call_synthesize(
        Parser const& parser
      , Iterator& first, Iterator const& last
      , Context const& context, Attribute& attr, mpl::false_)
    {
        // synthesized attribute needs to be value initialized
        typedef typename Attribute::value_type value_type;
        value_type val = value_type();

        if (!parser.parse(first, last, context, val))
            return false;

        // push the parsed value into our attribute
        attr.push_back(val);
        return true;
    }
```

Kleene Parser Implementation

```
template <typename Iterator, typename Context, typename Attribute>
    static bool call_synthesize(
        Parser const& parser
      , Iterator& first, Iterator const& last
      , Context const& context, Attribute& attr, mpl::false_)
    {
        // synthesized attribute needs to be value initialized
        typedef typename
            traits::container_value<Attribute>::type
        value_type;
        value_type val = traits::value_initialize<value_type>::call();

        if (!parser.parse(first, last, context, val))
            return false;

        // push the parsed value into our attribute
        traits::push_back(attr, val);
        return true;
    }
```

Traits and Customization Points (CP)

```
template <typename Iterator, typename Context, typename Attribute>
    static bool call_synthesize(
        Parser const& parser
      , Iterator& first, Iterator const& last
      , Context const& context, Attribute& attr, mpl::false_)
    {
        // synthesized attribute needs to be value initialized
        typedef typename
            traits::container_value<Attribute>::type
        value_type;
        value_type val = traits::value_initialize<value_type>::call();

        if (!parser.parse(first, last, context, val))
            return false;

        // push the parsed value into our attribute
        traits::push_back(attr, val);
        return true;
    }
```

Sequence Parser

```
template <typename Left, typename Right>
struct sequence : binary_parser<Left, Right, sequence<Left, Right>>
{
    typedef binary_parser<Left, Right, sequence<Left, Right>> base_type;

    sequence(Left left, Right right)
        : base_type(left, right) {}

    template <typename Iterator, typename Context>
    bool parse(
        Iterator& first, Iterator const& last
      , Context const& context, unused_type) const;

    template <typename Iterator, typename Context, typename Attribute>
    bool parse(
        Iterator& first, Iterator const& last
      , Context const& context, Attribute& attr) const;
};
```

binary_parser

```
template <typename Left, typename Right, typename Derived>
struct binary_parser : parser<Derived>
{
    typedef binary_category category;
    typedef Left left_type;
    typedef Right right_type;
    static bool const has_attribute =
        left_type::has_attribute || right_type::has_attribute;
    static bool const has_action =
        left_type::has_action || right_type::has_action;

    binary_parser(Left left, Right right)
        : left(left), right(right) {}

    binary_parser const& get_binary() const { return *this; }

    Left left;
    Right right;
};
```

Sequence ET

```
template <typename Left, typename Right>
inline sequence<
    typename extension::as_parser<Left>::value_type
  , typename extension::as_parser<Right>::value_type>
operator>>(Left const& left, Right const& right)
{
    typedef sequence<
        typename extension::as_parser<Left>::value_type
      , typename extension::as_parser<Right>::value_type>
    result_type;

    return result_type(as_parser(left), as_parser(right));
}
```

Invalid Expressions

```
namespace extension
{
    template <typename T, typename Enable = void>
    struct as_parser {};
}

template <typename T>
inline typename extension::as_parser<T>::type
as_parser(T const& x)
{
    return extension::as_parser<T>::call(x);
}
```

Invalid Expressions

```
template <typename Subject>
inline kleene<typename extension::as_parser<Subject>::value_type>
operator*(Subject const& subject);
```

```
auto const xx = term >> *not_a_parser;
```

```
error: no match for 'operator*' in '*not_a_parser'
```

Invalid Expressions

```
template <typename Left, typename Right>
inline sequence<
    typename extension::as_parser<Left>::value_type
  , typename extension::as_parser<Right>::value_type>
operator>>(Left const& left, Right const& right)
```

```
auto const xx = term >> not_a_parser;
```

```
error: no match for 'operator>>'
    in 'term >> not_a_parser'
```

Sequence Parser Implementation

```
template <typename Iterator, typename Context>
bool parse(
    Iterator& first, Iterator const& last
  , Context const& context, unused_type) const
{
    Iterator save = first;
    if (this->left.parse(first, last, context, unused)
        && this->right.parse(first, last, context, unused))
        return true;
    first = save;
    return false;
}
```

Sequence Parser Implementation

```
template <typename Iterator, typename Context, typename Attribute>
bool parse(
    Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr) const
{
    return detail::parse_sequence(
        this->left, this->right, first, last, context, attr
      , typename traits::attribute_category<Attribute>::type());
    return false;
}
```

Sequence Parser Implementation

```
template <typename Left, typename Right
  , typename Iterator, typename Context, typename Attribute>
bool parse_sequence(
    Left const& left, Right const& right
  , Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr, traits::container_attribute)
{
    Iterator save = first;
    if (parse_into_container(left, first, last, context, attr)
        && parse_into_container(right, first, last, context, attr))
        return true;
    first = save;
    return false;
}
```

Sequence Parser Implementation

```
template <typename Left, typename Right
  , typename Iterator, typename Context, typename Attribute>
bool parse_sequence(
    Left const& left, Right const& right
  , Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr, traits::tuple_attribute)
{
    typedef detail::partition_attribute<Left, Right, Attribute> partition;
    typedef typename partition::l_pass l_pass;
    typedef typename partition::r_pass r_pass;
```

Continued…

Sequence Parser Implementation

```
    typename partition::l_part l_part = partition::left(attr);
    typename partition::r_part r_part = partition::right(attr);
    typename l_pass::type l_attr = l_pass::call(l_part);
    typename r_pass::type r_attr = r_pass::call(r_part);

    Iterator save = first;
    if (left.parse(first, last, context, l_attr)
        && right.parse(first, last, context, r_attr))
        return true;
    first = save;
    return false;
}
```

Partitioning

```
'{' >> int_ >> ',' >> int_ >> '}'
```

```
sequence<
   sequence<
      sequence<
         sequence<
            literal_char<>
          , int_parser<int>>
        , literal_char<>>
     , int_parser<int> >
  , literal_char<>>
```

tuple<int, int>

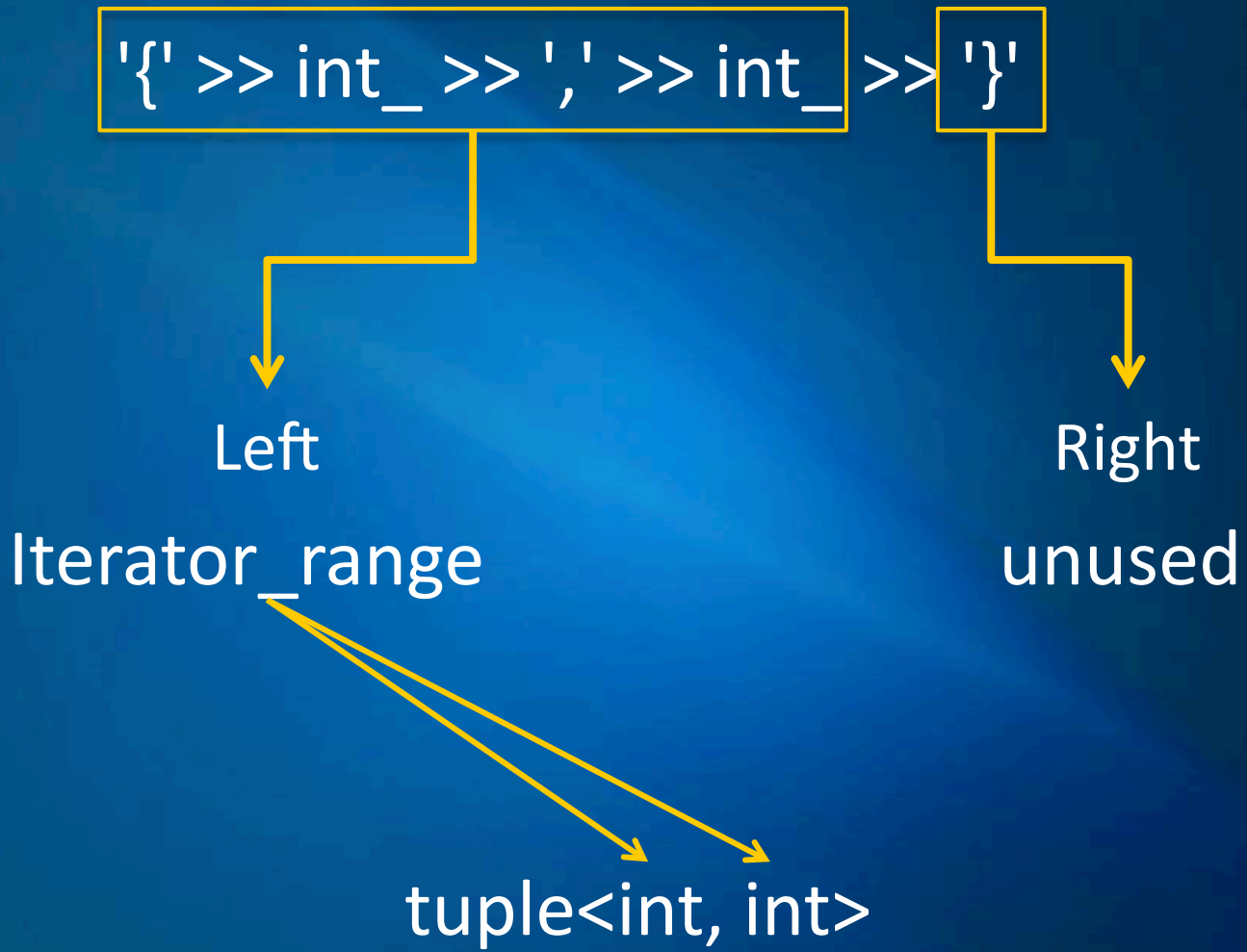
Partitioning

'{' >> int_ >> ',' >> int_ >> '}'

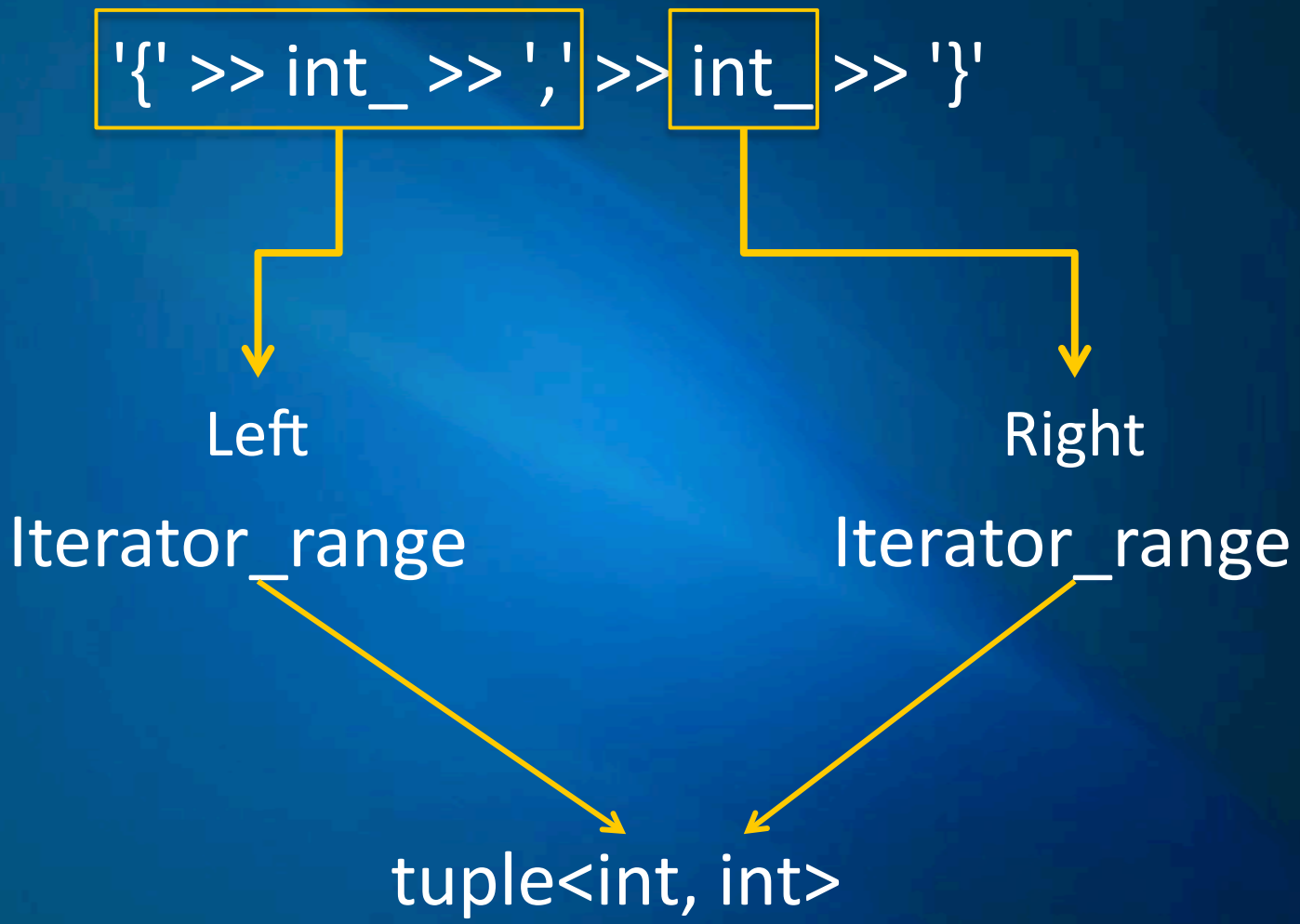
```
sequence<
   sequence<
      sequence<
         sequence<
            literal_char<>
          , int_parser<int>>
        , literal_char<>>
     , int_parser<int> >
  , literal_char<>>
```

tuple<int, int>

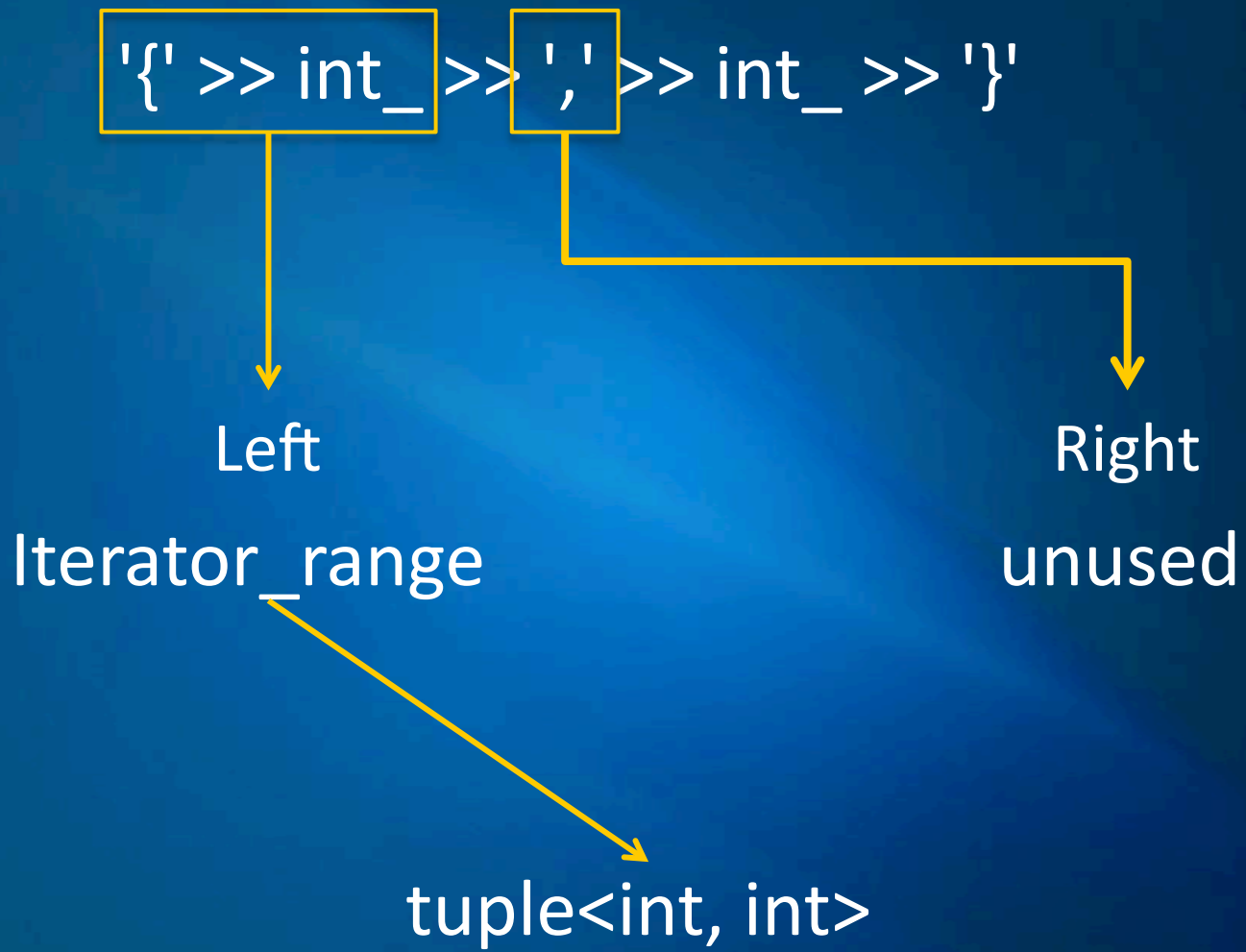
Partitioning



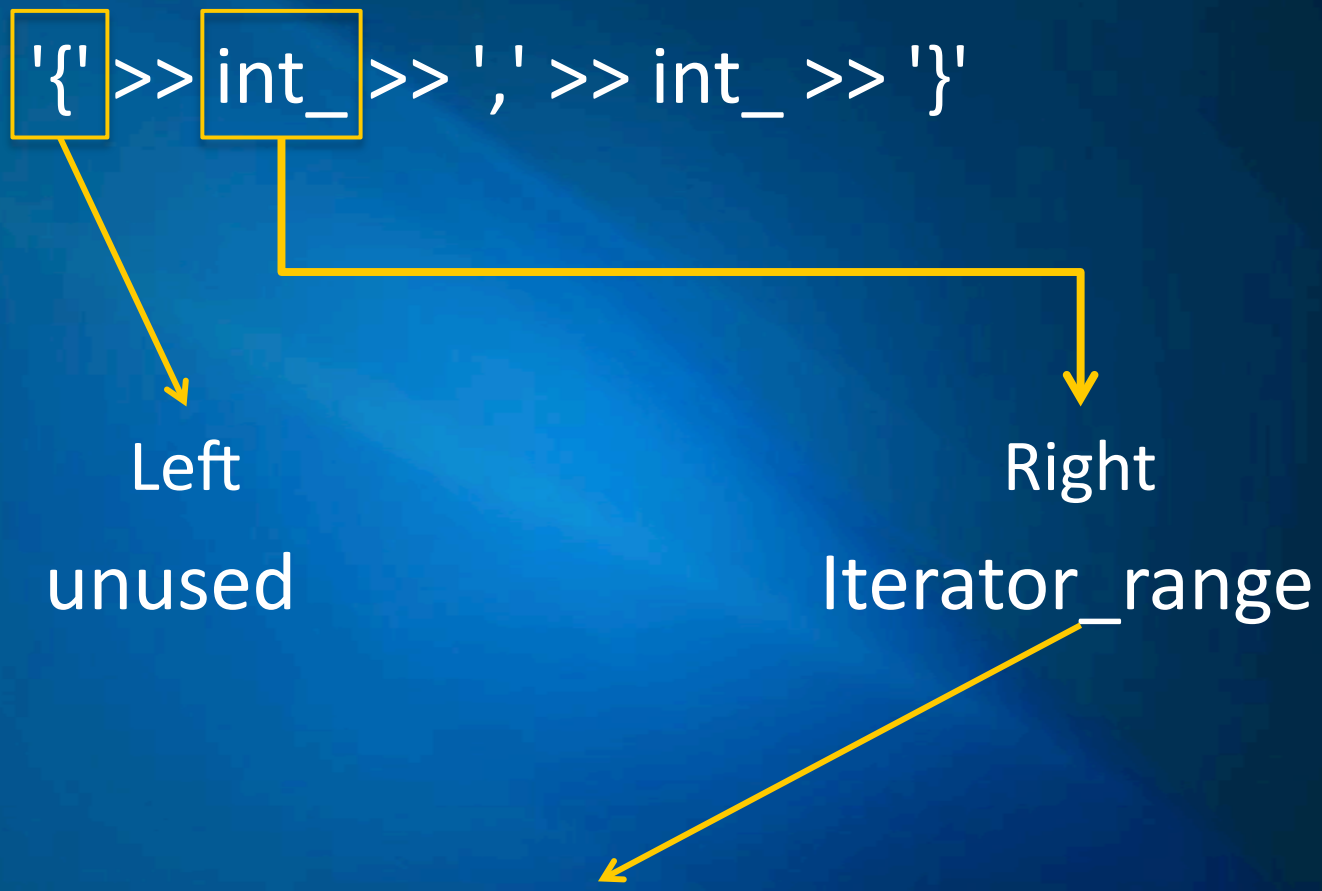
Partitioning



Partitioning



Partitioning



Alternative Parser

```
template <typename Left, typename Right>
struct alternative : binary_parser<Left, Right, alternative<Left, Right>>
{
    typedef binary_parser<Left, Right, alternative<Left, Right>> base_type;

    alternative(Left left, Right right)
        : base_type(left, right) {}

    template <typename Iterator, typename Context>
    bool parse(
        Iterator& first, Iterator const& last
      , Context const& context, unused_type) const;

    template <typename Iterator, typename Context, typename Attribute>
    bool parse(
        Iterator& first, Iterator const& last
      , Context const& context, Attribute& attr) const;
};
```

Alternative ET

```
template <typename Left, typename Right>
inline alternative<
    typename extension::as_parser<Left>::value_type
  , typename extension::as_parser<Right>::value_type>
operator|(Left const& left, Right const& right)
{
    typedef alternative<
        typename extension::as_parser<Left>::value_type
      , typename extension::as_parser<Right>::value_type>
    result_type;

    return result_type(as_parser(left), as_parser(right));
}
```

Alternative Parser Implementation

```
template <typename Iterator, typename Context>
bool parse(
    Iterator& first, Iterator const& last
  , Context const& context, unused_type) const
{
    return this->left.parse(first, last, context, unused)
        || this->right.parse(first, last, context, unused);
}
```

Alternative Parser Implementation

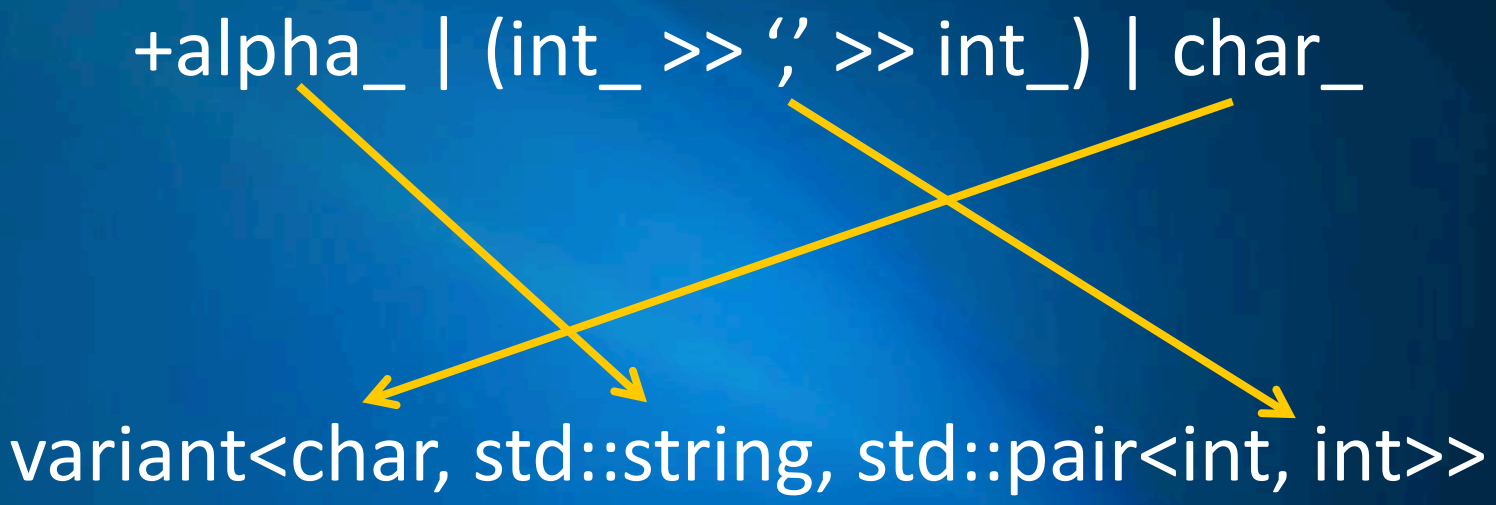
```
template <typename Iterator, typename Context, typename Attribute>
bool parse(
    Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr) const
{
    if (detail::parse_alternative(this->left, first, last, context, attr))
        return true;
    if (detail::parse_alternative(this->right, first, last, context, attr))
        return true;
    return false;
}
```

Alternative Parser Implementation

```
template <typename Parser, typename Iterator, typename Context, typename Attribute>
bool parse_alternative(
    Parser const& p, Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr)
{
    typedef detail::pass_variant_attribute<Parser, Attribute> pass;

    typename pass::type attr_ = pass::call(attr);
    if (p.parse(first, last, context, attr_))
    {
        if (!pass::is_alternative)
            traits::move_to(attr_, attr);
        return true;
    }
    return false;
}
```

Variant Attribute Mapping



find_substitute

```
template <typename Variant, typename Attribute>
struct find_substitute
{
    // Get the type from the variant that can be a substitute for Attribute.
    // If none is found, just return Attribute

    typedef Variant variant_type;
    typedef typename variant_type::types types;
    typedef typename mpl::end<types>::type end;

    typedef typename
        mpl::find_if<types, is_same<mpl::_1, Attribute> >::type
    iter_1;
```

Continued…

find_substitute

```
    typedef typename
        mpl::eval_if<
            is_same<iter_1, end>,
            mpl::find_if<types, traits::is_substitute<mpl::_1, Attribute> >,
            mpl::identity<iter_1>
        >::type
    iter;

    typedef typename
        mpl::eval_if<
            is_same<iter, end>,
            mpl::identity<Attribute>,
            mpl::deref<iter>
        >::type
    type;
};
```

Rule Definition

```
template <typename ID, typename RHS, typename Attribute>
struct rule_definition : parser<rule_definition<ID, RHS, Attribute>>
{
    typedef rule_definition<ID, RHS, Attribute> this_type;
    typedef ID id;
    typedef RHS rhs_type;
    typedef Attribute attribute_type;
    static bool const has_attribute = !is_same<Attribute, unused_type>::value;
    static bool const handles_container = traits::is_container<Attribute>::value;

    rule_definition(RHS rhs, char const* name)
      : rhs(rhs), name(name) {}

    template <typename Iterator, typename Context, typename Attribute_>
    bool parse(Iterator& first, Iterator const& last
      , Context const& context, Attribute_& attr) const;

    RHS rhs;
    char const* name;
};
```

Rule Context

```
template <typename Attribute>
struct rule_context
{
   Attribute& val() const
   {
      BOOST_ASSERT(attr_ptr);
      return *attr_ptr;
   }

   Attribute* attr_ptr;
};

struct rule_context_tag;

template <typename ID>
struct rule_context_with_id_tag;
```