Encapsulating a Grammar

```
namespace parser
{
  namespace g_definition
  {
    rule<class x> const x;
    auto const ax = char_('a') >> x;

    auto const g =
      x = char_('x') | ax;
  }
  using g_definition::g;
}
```

Walk-through Spirit X3

- Basic Parsers
  - Eps Parser
  - Int Parser
- Composite Parsers
  - Kleene Parser
  - Sequence Parser
  - Alternative Parser
- Nonterminals
  - Rule
  - Grammar
- Semantic Actions

Eps Parser

```
struct eps_parser : parser<eps_parser>
{
    typedef unused_type attribute_type;
    static bool const has_attribute = false;

    template <typename Iterator, typename Context, typename Attribute>
    bool parse(Iterator& first, Iterator const& last
      , Context const& context, Attribute& /*attr*/) const
    {
        x3::skip_over(first, last, context);
        return true;
    }
};
```

Attributes

- Parsers expose an attribute specific to their type
  - int_ → int
  - char_ → char
  - *int_ → std::vector<int>
  - int_ >> char_ → fusion::deque<int, char>
- Some parsers may have *unused* "don't care" attributes
  - literals: e.g. 'z', "hello"
  - eps, eoi, predicates: e.g. !p, &p

Attribute Categories

unused_attribute	unused
plain_attribute	int, char, double
container_attribute	std::vector<int>
tuple_attribute	fusion::list<int, char>
variant_attribute	variant<int, X>
optional_attribute	optional<int>

Attribute Propagation

a >> b

- Attribute Synthesis
  - a -> T, b -> U → (a >> b) -> tuple<T, U>
- Attribute Collapsing
  - a -> T, b ->unused → T
  - a -> unused, b ->U → U
  - a -> unused, b ->unused → unused
- Attribute Compatibility
  - (a >> b) := vector<T> → a := T, b := T
    → a := vector<T>, b := T
    → a := T, b := vector<T>
    → a := vector<T>, b := vector<T>

unused_type

```
struct unused_type
{
    unused_type() {}

    template <typename T>
    unused_type(T const&) {}

    template <typename T>
    unused_type const& operator=(T const&) const { return *this; }

    template <typename T>
    unused_type& operator=(T const&) { return *this; }

    unused_type const& operator=(unused_type const&) const { return *this; }
    unused_type& operator=(unused_type const&) { return *this; }
};
```

The Context Refined

```
template <typename ID, typename T,
    typename Next = unused_type>
struct context
{
    context(T& val, Next const& next)
        : val(val), next(next) {}

    template <typename ID_,
        typename Unused = void>
    struct get_result
    {
        typedef typename Next::template
         get_result<ID_>::type type;
    };

    template <typename Unused>
    struct get_result<mpl::identity<ID>, Unused>
    {
        typedef T& type;
    };

    T& get(mpl::identity<ID>) const
    {
        return val;
    }

    template <typename ID_>
    typename Next::template get_result<ID_>::type
    get(ID_ id) const
    {
        return next.get(id);
    }

    T& val;
    Next const& next;
};
```

The Context Refined

```
// unused_type can also masquerade as an empty context (see context.hpp)

template <typename ID>
struct get_result : mpl::identity<unused_type> {};

template <typename ID>
unused_type get(ID) const
{
    return unused_type();
}
```

skip_over

```
template <typename Iterator, typename Context>
inline void skip_over(
    Iterator& first, Iterator const& last, Context const& context)
{
    detail::skip_over(first, last, spirit::get<skipper_tag>(context));
}


template <typename Iterator, typename Skipper>
inline void skip_over(
    Iterator& first, Iterator const& last, Skipper const& skipper)
{
    while (first != last && skipper.parse(first, last, unused, unused))
        /***/;
}
```

Eps Parser

```
struct eps_parser : parser<eps_parser>
{
    typedef unused_type attribute_type;
    static bool const has_attribute = false;

    template <typename Iterator, typename Context, typename Attribute>
    bool parse(Iterator& first, Iterator const& last
      , Context const& context, Attribute& /*attr*/) const
    {
        x3::skip_over(first, last, context);
        return true;
    }
};

eps_parser const eps = eps_parser();
```

Int Parser

```
template <typename T, unsigned Radix = 10, unsigned MinDigits = 1 , int MaxDigits = -1>
struct int_parser : parser<int_parser<T, Radix, MinDigits, MaxDigits>>
{
   typedef T attribute_type;
   static bool const has_attribute = true;

   template <typename Iterator, typename Context, typename Attribute>
   bool parse(Iterator& first, Iterator const& last
     , Context const& context, Attribute& attr) const
   {
      typedef extract_int<T, Radix, MinDigits, MaxDigits> extract;
      x3::skip_over(first, last, context);
      return extract::call(first, last, attr);
   }
};

int_parser<int> const int_= int_parser<int>();
```

Kleene Parser

```
template <typename Subject>
struct kleene : unary_parser<Subject, kleene<Subject>>
{
   typedef unary_parser<Subject, kleene<Subject>> base_type;
   typedef typename traits::attribute_of<Subject>::type subject_attribute;
   static bool const handles_container = true;

   typedef typename
      traits::build_container<subject_attribute>::type
   attribute_type;

   kleene(Subject const& subject)
     : base_type(subject) {}

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

unary_parser

```
template <typename Subject, typename Derived>
struct unary_parser : parser<Derived>
{
    typedef unary_category category;
    typedef Subject subject_type;
    static bool const has_attribute = Subject::has_attribute;
    static bool const has_action = Subject::has_action;

    unary_parser(Subject subject)
      : subject(subject) {}

    unary_parser const& get_unary() const { return *this; }

    Subject subject;
};
```

Kleene ET

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

    return result_type(as_parser(subject));
}
```

as_parser

```
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);
}
```

as_parser

```
template <>
struct as_parser<unused_type>
{
    typedef unused_type type;
    typedef unused_type value_type;
    static type call(unused_type)
    {
        return unused;
    }
};
```

as_parser

```
template <typename Derived>
struct as_parser<Derived
    , typename enable_if<is_base_of<parser_base, Derived>>::type>
{
    typedef Derived const& type;
    typedef Derived value_type;
    static type call(Derived const& p)
    {
        return p;
    }
};
```

as_parser

```
template <>
struct as_parser<char>
{
    typedef literal_char<
        char_encoding::standard, unused_type>
    type;

    typedef type value_type;

    static type call(char ch)
    {
        return type(ch);
    }
};
```

Kleene Parser Implementation

```
template <typename Iterator, typename Context, typename Attribute>
bool parse(Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr) const
{
    while (detail::parse_into_container(
        this->subject, first, last, context, attr))
        ;
    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 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>

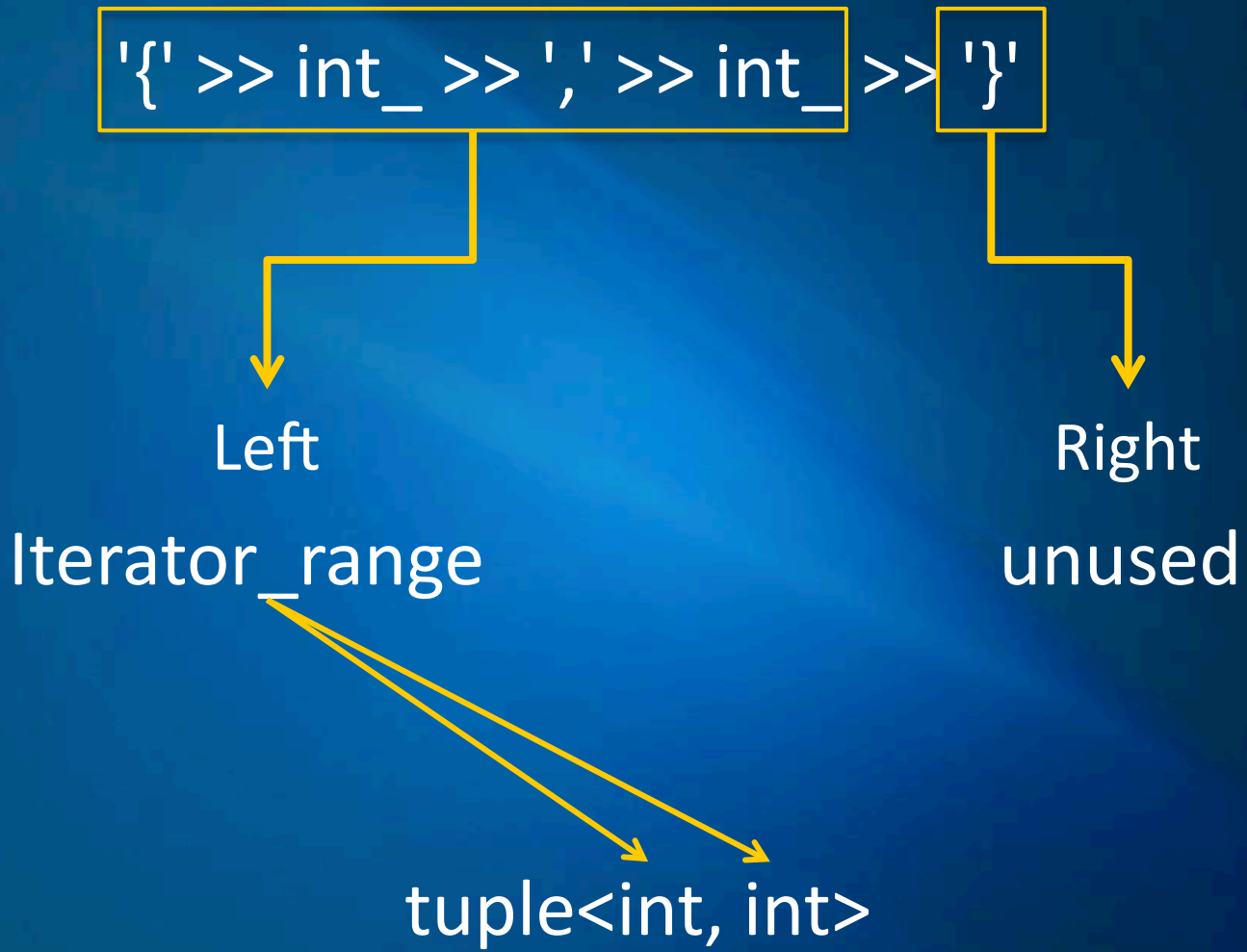
Partitioning

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

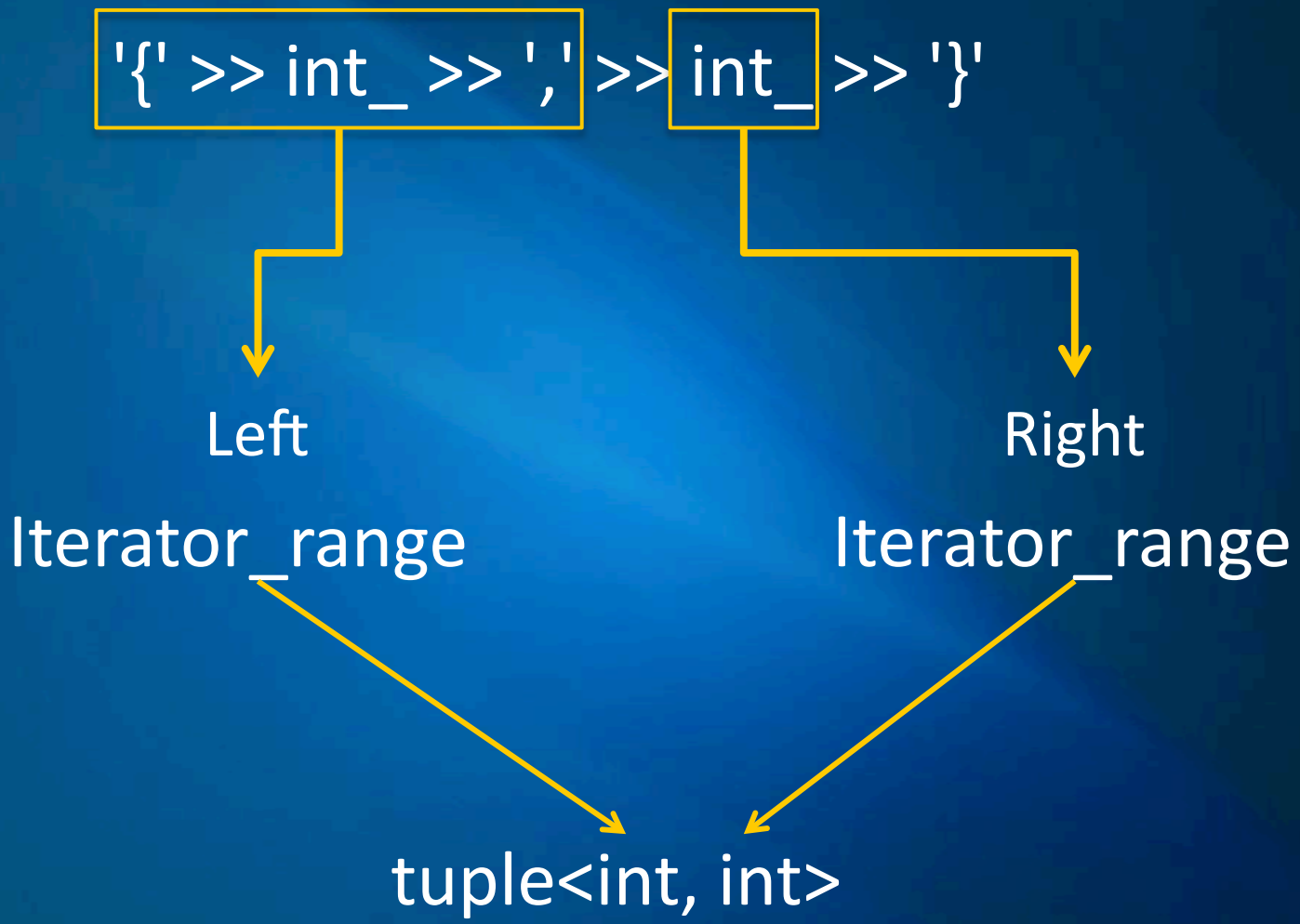
```
sequence<
   sequence<
      sequence<
         sequence<
            literal_char<>
          , int_parser<int>>
        , literal_char<>>
     , int_parser<int> >
  , literal_char<>>
```

tuple<int, int>

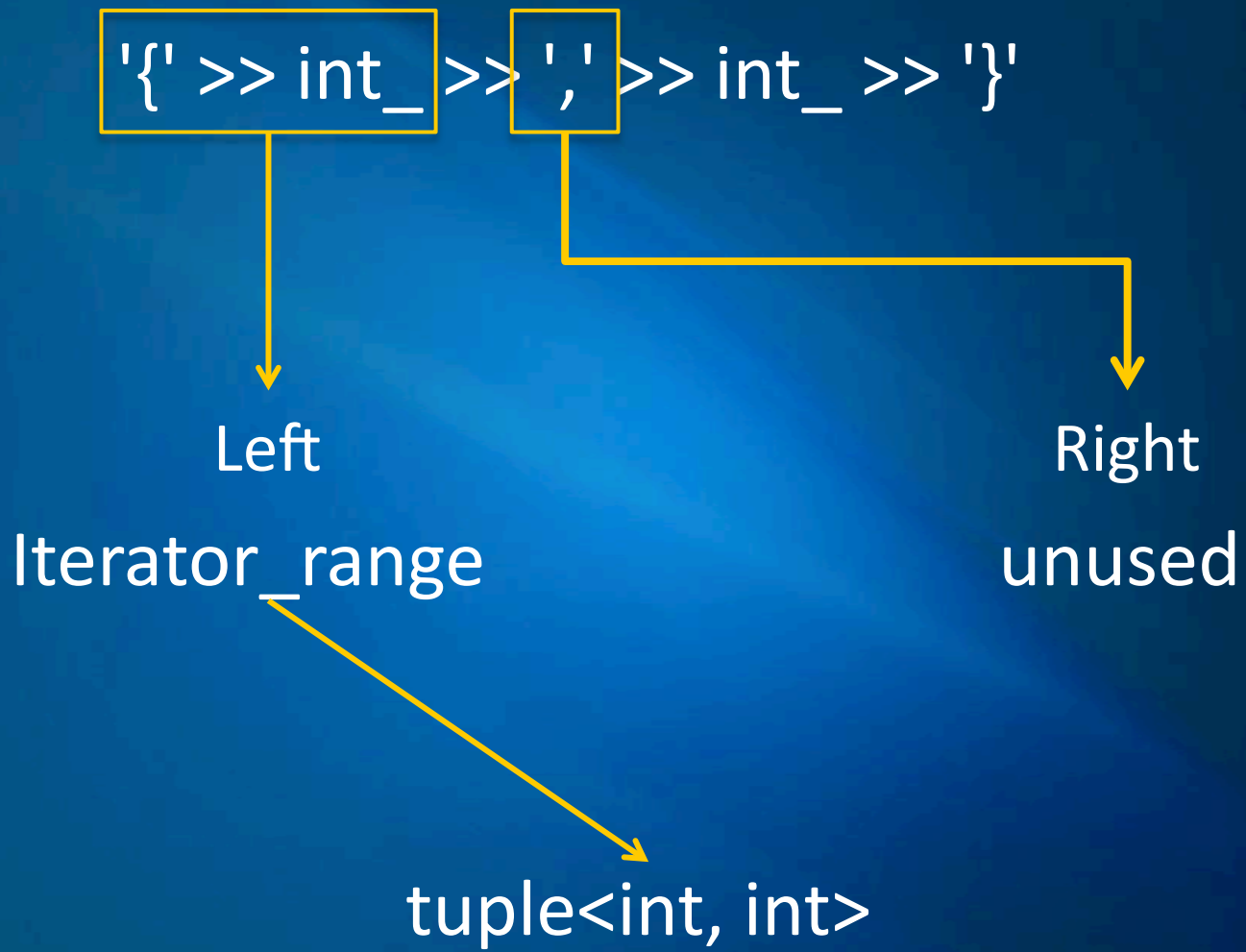
Partitioning



Partitioning



Partitioning



Partitioning