Inside Spirit X3

Redesigning Boost.Spirit for C++11

Joel de Guzman
Ciere Consulting

Agenda

- Quick Overview
- Parser Combinator
- Let’s Build a Toy Spirit X3
- Walk-through Spirit X3

What's Spirit

- A object oriented, recursive-descent parser and output generation library for C++
  - Implemented using template meta-programming techniques
  - Syntax of Parsing Expression Grammars (PEGs) directly in C++, used for input and output format specification
- Target grammars written entirely in C++
  - No separate tools to compile grammar
  - Seamless integration with other C++ code
  - Immediately executable

Spirit X3

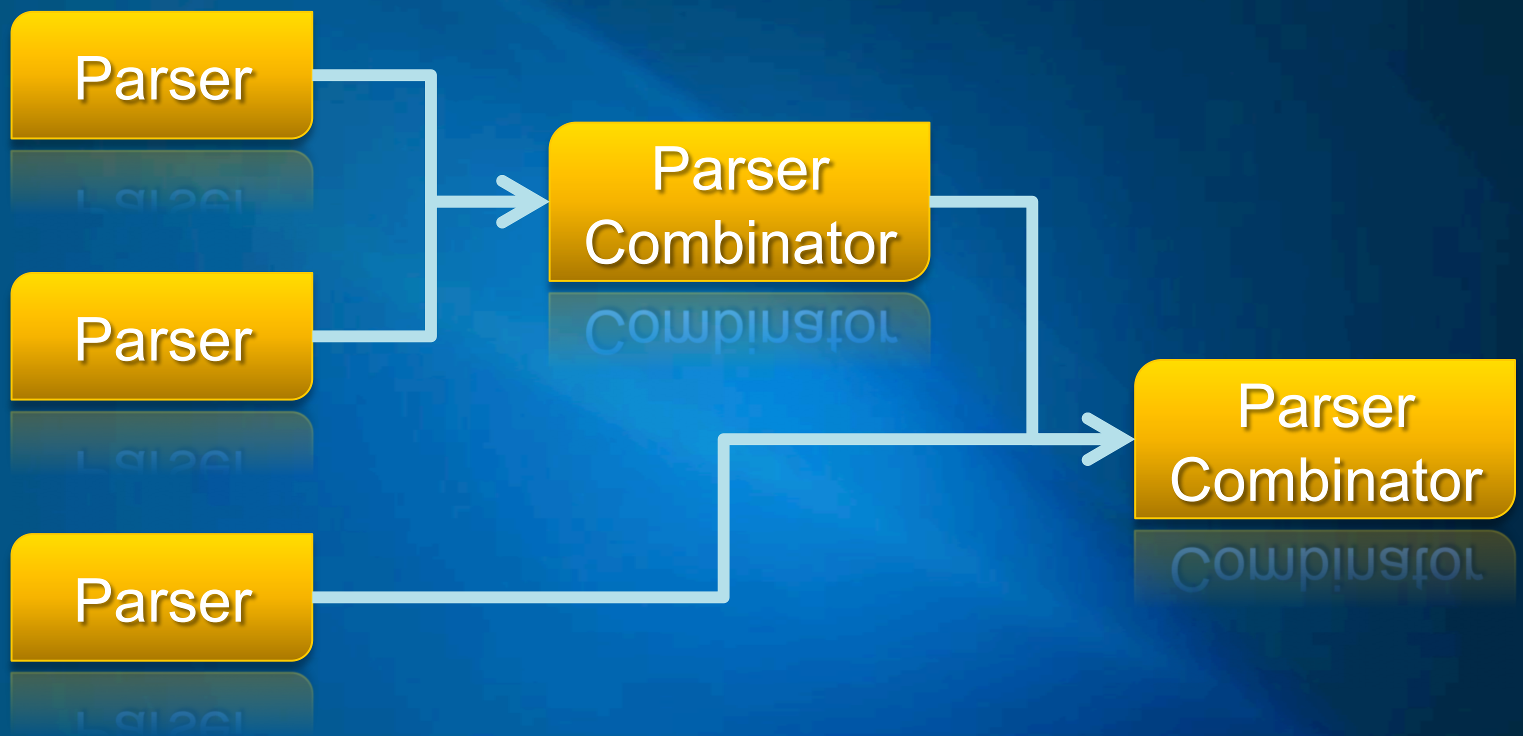
- Experimental
- C++11
- Hackable, simpler design
- Minimal code base and dependencies
    - MPL
    - Fusion
    - Phoenix?
    - Proto?
- Better error handling
- Faster compile times

calc4.cpp example
SpiritX3: TOTAL : 4.27 secs
Spirit2: TOTAL : 10.00 secs

Parser Combinator

- A Parser is a function
  - A character parser
  - A numeric parser
- Parsers can be composed to form higher order *parser* functions
  - E.g. a sequence parser accepts two parsers and returns a composite parser
  - Such a higher order *parser* function is called a Parser Combinator. A Parser Combinator accepts several parsers as input and returns a composite parser as result

Parser Combinator



Parser Combinator

- Primitives (plain characters, uint_, etc.)

```
bool match_char(char ch)
    { return ch== input(); }
```

- Sequences

```
bool match_sequence(F1 f1, F2 f2)
    { return f1() && f2(); }
```

- Alternatives

```
bool match_alternative(F1 f1, F2 f2)
    { return f1() || f2(); }
```

- Modifiers (kleen, plus, etc.)

```
bool match_kleene(F f)
    { while (f()); return true; }
```

- Nonterminals (factor, term, expr)

```
bool match_rule()
    { return match_rhs(); }
```

Parsing Expression Grammar

- Formal grammar for describing a formal language in terms of a set of rules used to recognize strings of this language
  - Does not require a tokenization stage
- Similar to Extended Backus-Naur Form (EBNF)
- Unlike (E)BNF, PEG's are not ambiguous
  - Exactly one valid parse tree for each PEG
- Any PEG can be directly represented as a recursive-descent parser
- Different Interpretation as EBNF
  - Greedy Loops
  - First come first serve alternates

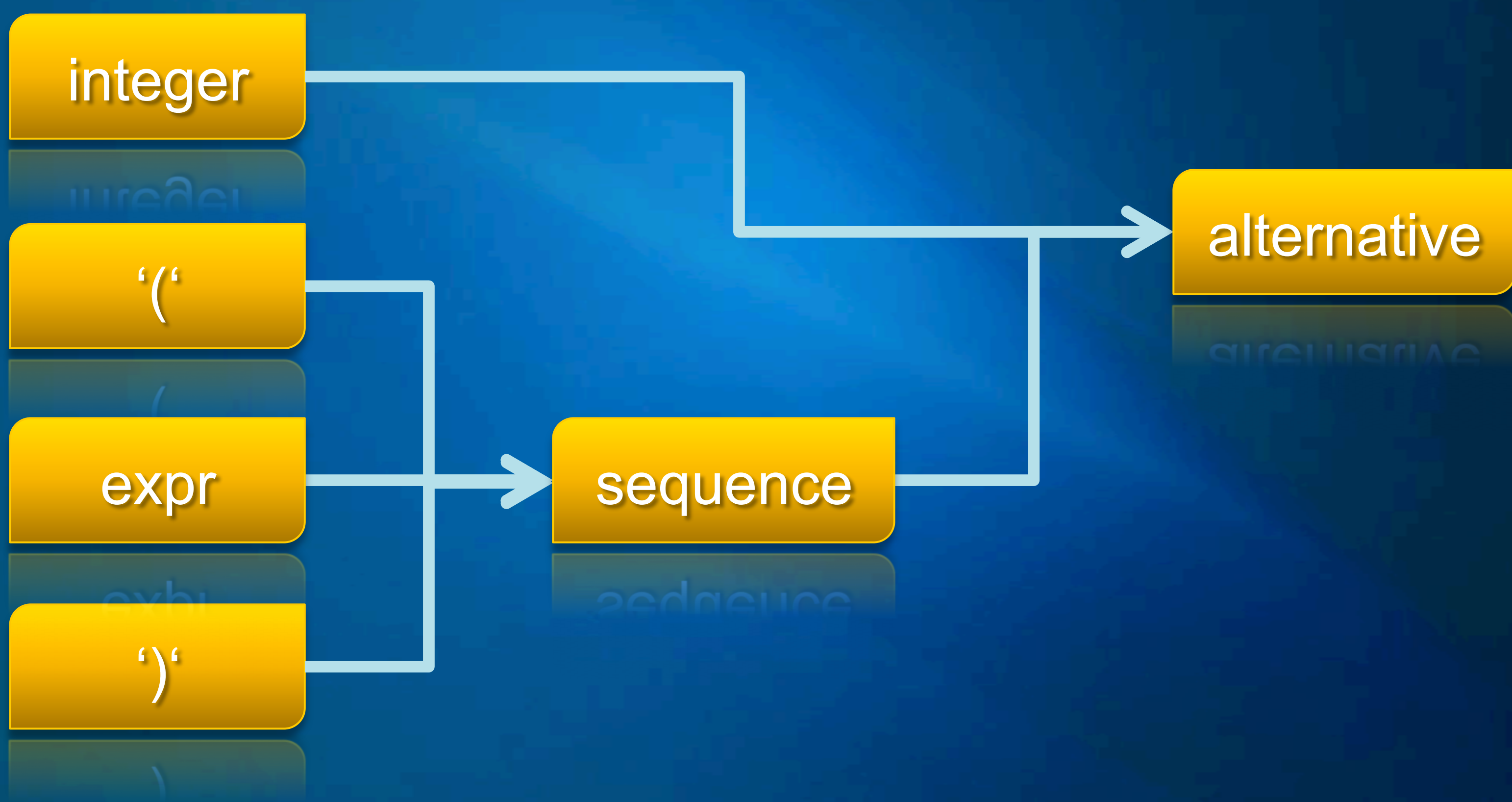
Calculator PEG Grammar

```
factor    ←    integer  /  '('  expr  ')'
term      ←    factor (('*'  factor)  /  ('/'  factor))*
expr      ←    term (('+' term)  /  ('-'  term))*
```

- A recursive descent parser is a top-down parser built from a set of mutually-recursive functions, each representing one of the grammar elements
- Thus the structure of the resulting program closely mirrors that of the grammar it recognizes

Parser Composition

factor ← integer / ‘(‘ expr ‘)’

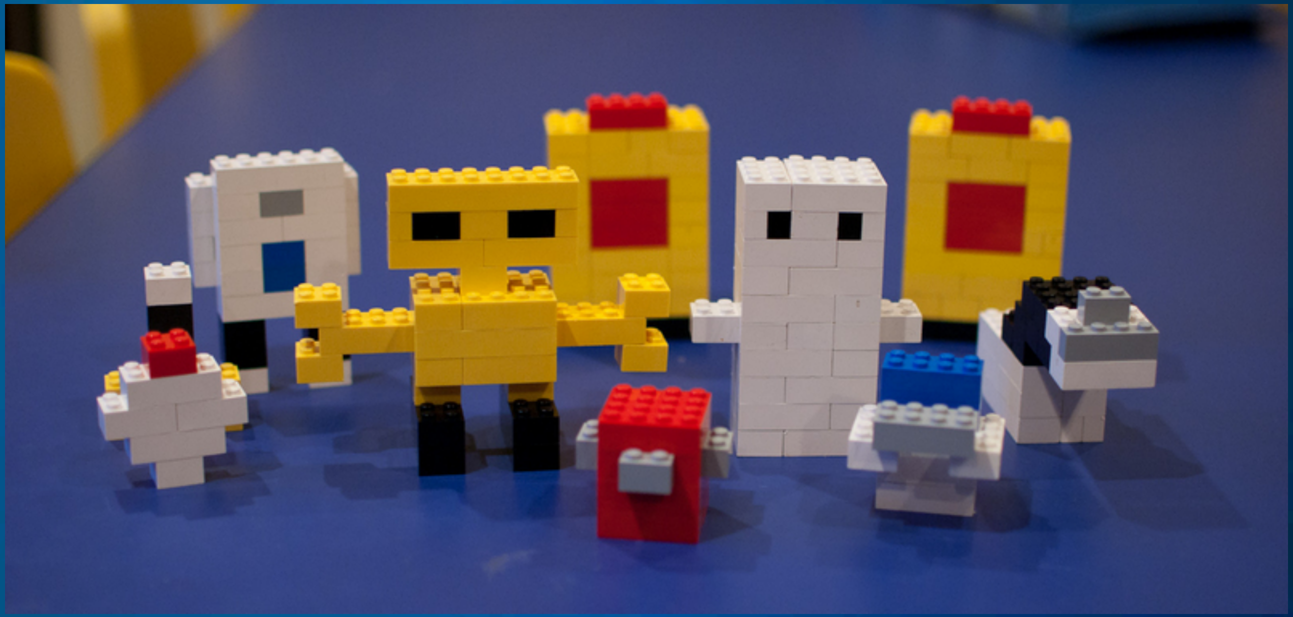


Parser Composition

factor ← integer / ‘(‘ expr ‘)’

```
bool match_fact()
{
   return     match_integer() ||
      (
               match_char( ‘(‘ )
         &&    match_expr()
         &&    match_char( ‘)’ )
      );
}
```

Let's build a toy Spirit X3



The Parser Base Class

```
namespace boost { namespace spirit { namespace x3
{
    template <typename Derived>
    struct parser
    {
        Derived const& derived() const
        {
            return *static_cast<Derived const*>(this);
        }
    };
}
```

The parse member function

```
template <typename Iterator, typename Context>
bool parse(
    Iterator& first,
    Iterator last,
    Context const& ctx) const
```

Postconditions

- Upon return from p.parse the following post conditions should hold:
  - On a successful match, first is positioned one past the last matching character.
  - On a failed match, first is restored to its original position prior to entry.
  - No post-skips: trailing skip characters will not be skipped.

Our First Primitive Parser

```
template <typename Char>
struct char_parser : parser<char_parser<Char>>
{
   char_parser(Char ch) : ch(ch) {}

   template <typename Iterator, typename Context>
   bool parse(Iterator& first, Iterator last, Context const& ctx) const
   {
      if (first != last && *first == ch)
      {
         ++first;
         return true;
      }
      return false;
   }

   Char ch;
};
```

char_ ET

```
template <typename Char>
inline char_parser<Char> char_(Char ch)
{
   return char_parser<Char>(ch);
};
```

Our First Composite Parser

```
template <typename Left, typename Right>
struct sequence_parser : parser<sequence_parser<Left, Right>>
{
   sequence_parser(Left left, Right right)
      : left(left), right(right) {}

   template <typename Iterator, typename Context>
   bool parse(Iterator& first, Iterator last, Context const& ctx) const
   {
      return left.parse(first, last, ctx)
         && right.parse(first, last, ctx);
   }

   Left left;
   Right right;
};
```

Sequence ET

```
template <typename Left, typename Right>
inline sequence_parser<Left, Right> operator>>(
    parser<Left> const& left, parser<Right> const& right)
{
    return sequence_parser<Left, Right>(
        left.derived(), right.derived());
}
```

Another Composite Parser

```
template <typename Left, typename Right>
struct alternative_parser : parser<alternative_parser<Left, Right>>
{
   alternative_parser(Left left, Right right)
      : left(left), right(right) {}

   template <typename Iterator, typename Context>
   bool parse(Iterator& first, Iterator last, Context const& ctx) const
   {
      if (left.parse(first, last, ctx))
         return true;
      return right.parse(first, last, ctx);
   }

   Left left;
   Right right;
};
```

Alternative ET

```
template <typename Left, typename Right>
inline alternative_parser<Left, Right> operator|(
    parser<Left> const& left, parser<Right> const& right)
{
    return alternative_parser<Left, Right>(
        left.derived(), right.derived());
}
```

Simple Rules

```
auto abc =
        char_('a')
    >>  char_('b')
    >>  char_('c')
    ;

auto a_or_bc =
        char_('a')
    |   ( char_('b') >> char_('c') )
    ;
```

But how about Recursion?

- I want a rule that parses these inputs:
  - “x”
  - “ax”
  - “aax”
  - “aaaaax”
- In other words: I want zero or more ‘a’s followed by an ‘x’
- No, we don’t have the Kleene star yet ;-)

But how about Recursion?

```
auto const x = char_('x') | ax;
auto const ax = char_('a') >> x;
```

But how about Recursion?

```
auto const x = char_('x') | ax;
auto const ax = char_('a') >> x;
```

Nonterminals

- The rule is a polymorphic parser that acts as a named placeholder capturing the behavior of a PEG expression assigned to it.
- Naming a PEG expression allows it to be referenced later and makes it possible for the rule to call itself.
- This is one of the most important mechanisms and the reason behind the word “recursive” in recursive descent parsing.

Spirit-2 and Spirit-Classic style

- Uses type-erasure
  - Abstract class with virtual functions
  - Boost or std function

```
rule<Iterator> x, ax;
x = char_('x') | ax;
ax = char_('a') >> x;
```

Problems with type-erasure

- All template parameters for parse should be known before hand.
  - Hence the rule needs to know the “scanner” type (Spirit-Classic) and the Iterator type (Spirit-2).
- Code bloat
  - The virtual functions force instantiations even if, in the end, they are not really used. Same with Boost or std function.
- Prevents optimizations
  - The virtual function is an opaque wall. In general, compilers cannot see beyond this opaque wall and cannot perform optimizations.

X3 style

- Does not use type-erasure
- Inspired by Spirit-Classic *Subrules*
  - Taken to the next level with the help of C++11 facilities that were not available at the time (e.g. auto and variadic templates)
  - V2 and Classic subrules are compile time monsters with its heavy reliance on expression templates

The Context

- Allows functions to efficiently access data from other stack frames
  - Caller sets up a Context
  - Callee retrieve the Context as needed
- On demand (pull vs. push)
- Data can be polymorphic
- Efficient alternative to passing arguments to functions
- Data can cross multiple stack frames
- Allows multiple contexts to be linked up

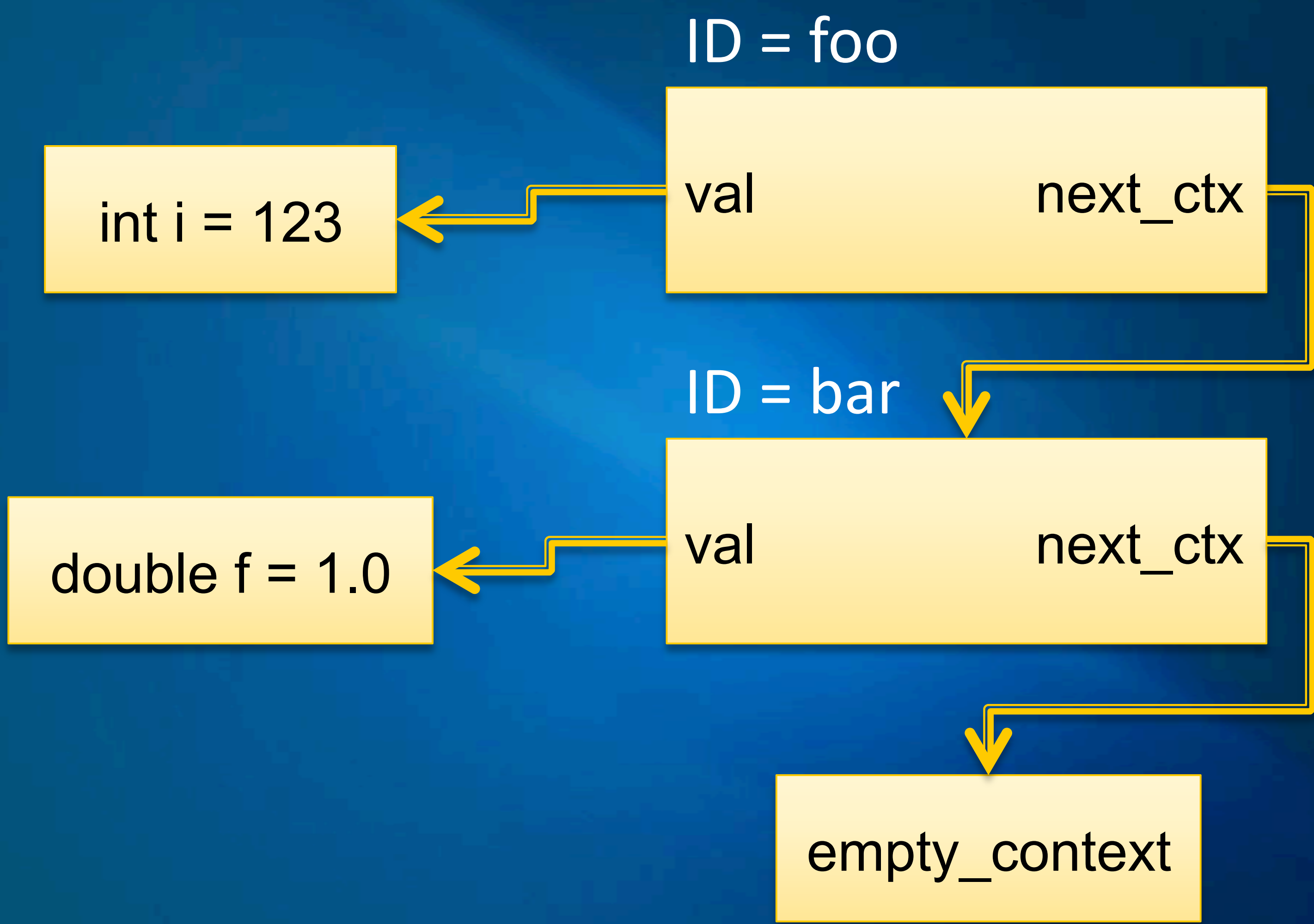
The Context

```
template <typename ID, typename T, typename NextContext>
struct context
{
   context(T const& val, NextContext const& next_ctx)
      : val(val), next_ctx(next_ctx) {}

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

   template <typename Identity>
   decltype(std::declval<NextContext>().get(Identity()))
   get(Identity id) const
   {
      return next_ctx.get(id);
   }

   T const& val;
   NextContext const& next_ctx;
};
```

The Empty Context

```
struct empty_context
{
   struct undefined {};
   template <typename ID>
   undefined get(ID) const
   {
      return undefined();
   }
};
```

The Context


ID = foo
val
next_ctx
int i = 123
ID = bar
val
next_ctx
double f = 1.0
empty_context

Example Context Usage

```
struct foo_id;

template <typename Context>
void bar(Context const& ctx)
{
      std::cout << ctx.get(mpl::identity<foo_id>()) << std::endl;
}

void foo()
{
   int i = 123;
   empty_context empty_ctx;
   context<foo_id , int, empty_context> ctx(i, empty_ctx);
   bar(ctx);
}
```

Example Context Usage

```
struct foo_id;

template <typename Context>
void bar(Context const& ctx)
{
      std::cout << ctx.get(mpl::identity<foo_id>()) << std::endl;
}

void foo()
{
   int i = 123;
   empty_context empty_ctx;
   context<foo_id , int, empty_context> ctx(i, empty_ctx);
   bar(ctx);
}
```

The Rule Definition

```
template <typename ID, typename RHS>
struct rule_definition : parser<rule_definition<ID, RHS>>
{
    rule_definition(RHS rhs)
        : rhs(rhs) {}

    template <typename Iterator, typename Context>
    bool parse(Iterator& first, Iterator last, Context const& ctx) const
    {
        context<ID, RHS, Context> this_ctx(rhs,  ctx);
        return rhs.parse(first, last, this_ctx);
    }

    RHS rhs;
};
```

The Rule

```
template <typename ID>
struct rule : parser<rule<ID>>
{
    template <typename Derived>
    rule_definition<ID, Derived>
    operator=(parser<Derived> const& definition) const
    {
        return rule_definition<ID, Derived>(definition.derived());
    }

    template <typename Iterator, typename Context>
    bool parse(Iterator& first, Iterator last, Context const& ctx) const
    {
        return ctx.get(mpl::identity<ID>()).parse(first, last, ctx);
    }
};
```

The main parse function

```
template <typename Iterator, typename Derived>
inline bool parse(parser<Derived> const& p, Iterator& first, Iterator last)
{
   empty_context ctx;
   return p.derived().parse(first, last, ctx);
}
```

Our Recursive Rule X3 style

```
rule<class x> const x;
auto const ax = char_('a') >> x;
auto const start =
    x = char_('x') | ax;
```

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>
```

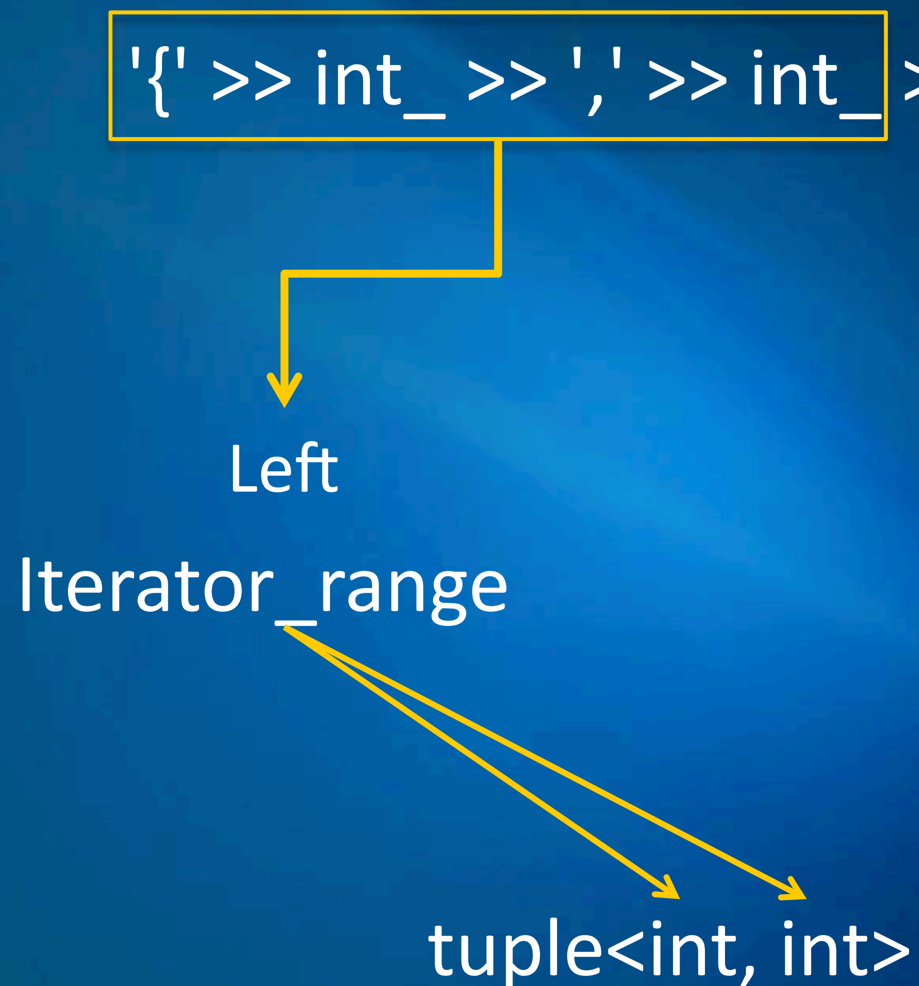
Partitioning

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

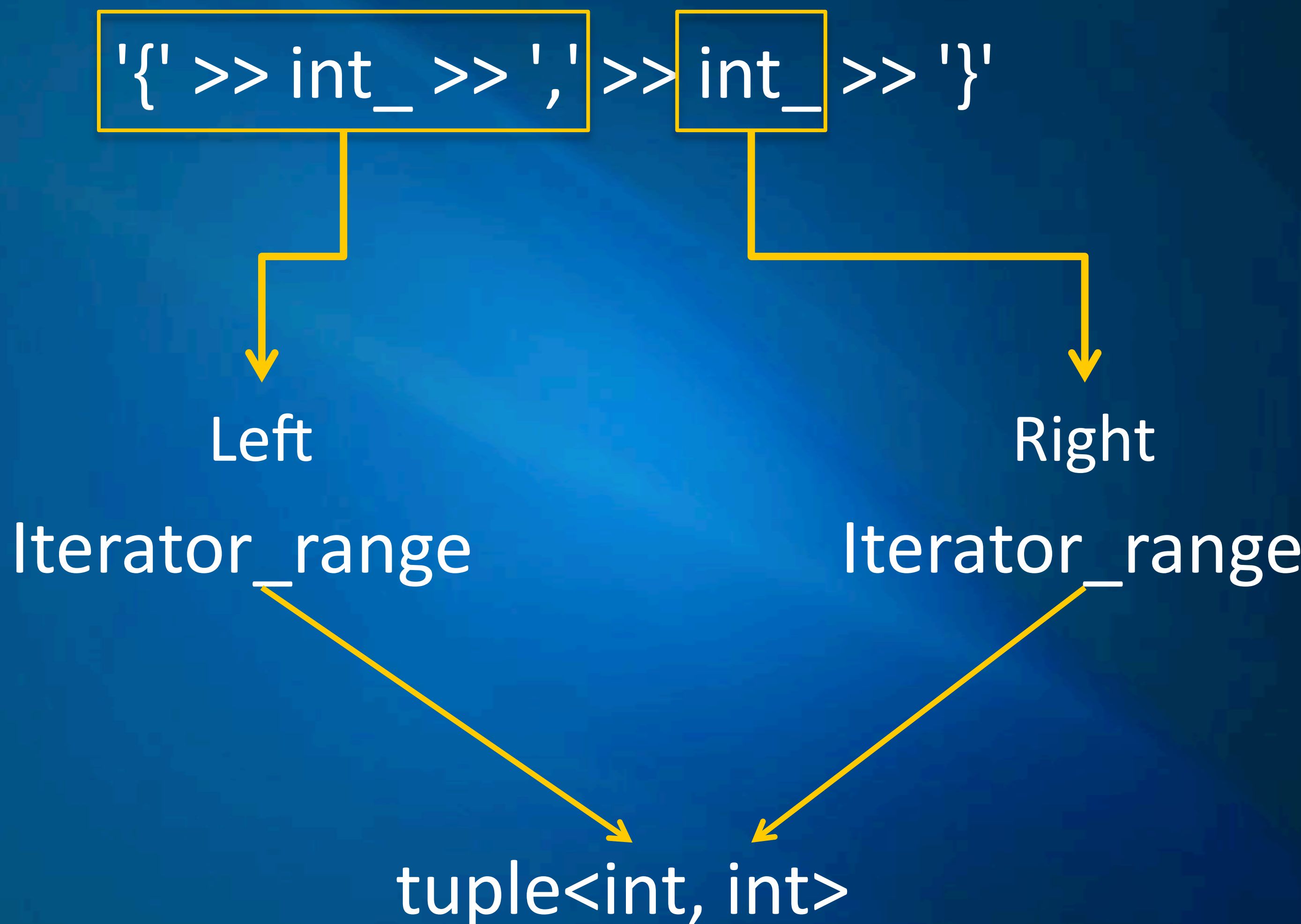
```
sequence<
    sequence<
        sequence<
            sequence<
                literal_char<>
              , int_parser<int>>
          , literal_char<>>
      , int_parser<int> >
  , literal_char<>>
```

```
tuple<int, int>
```

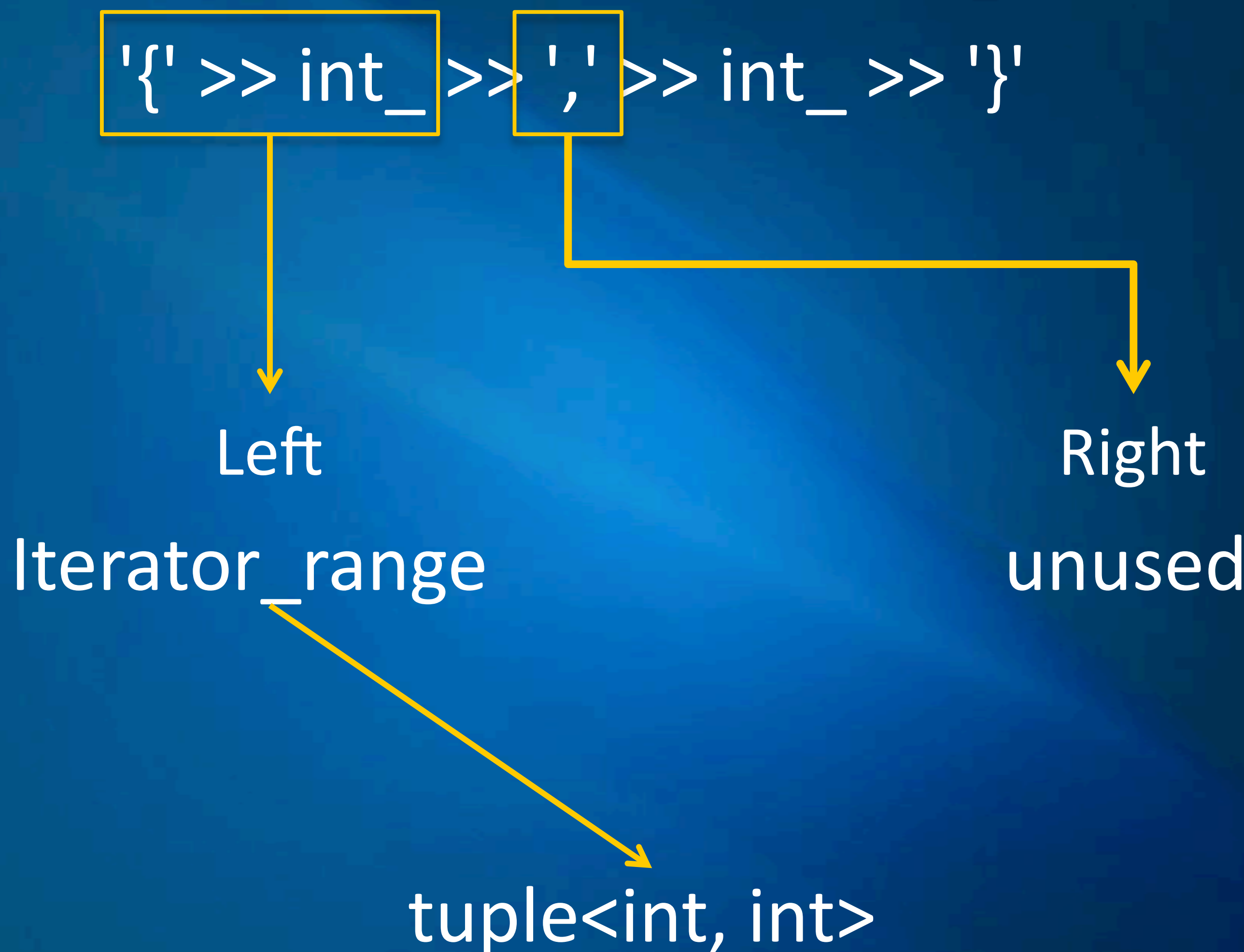
Partitioning



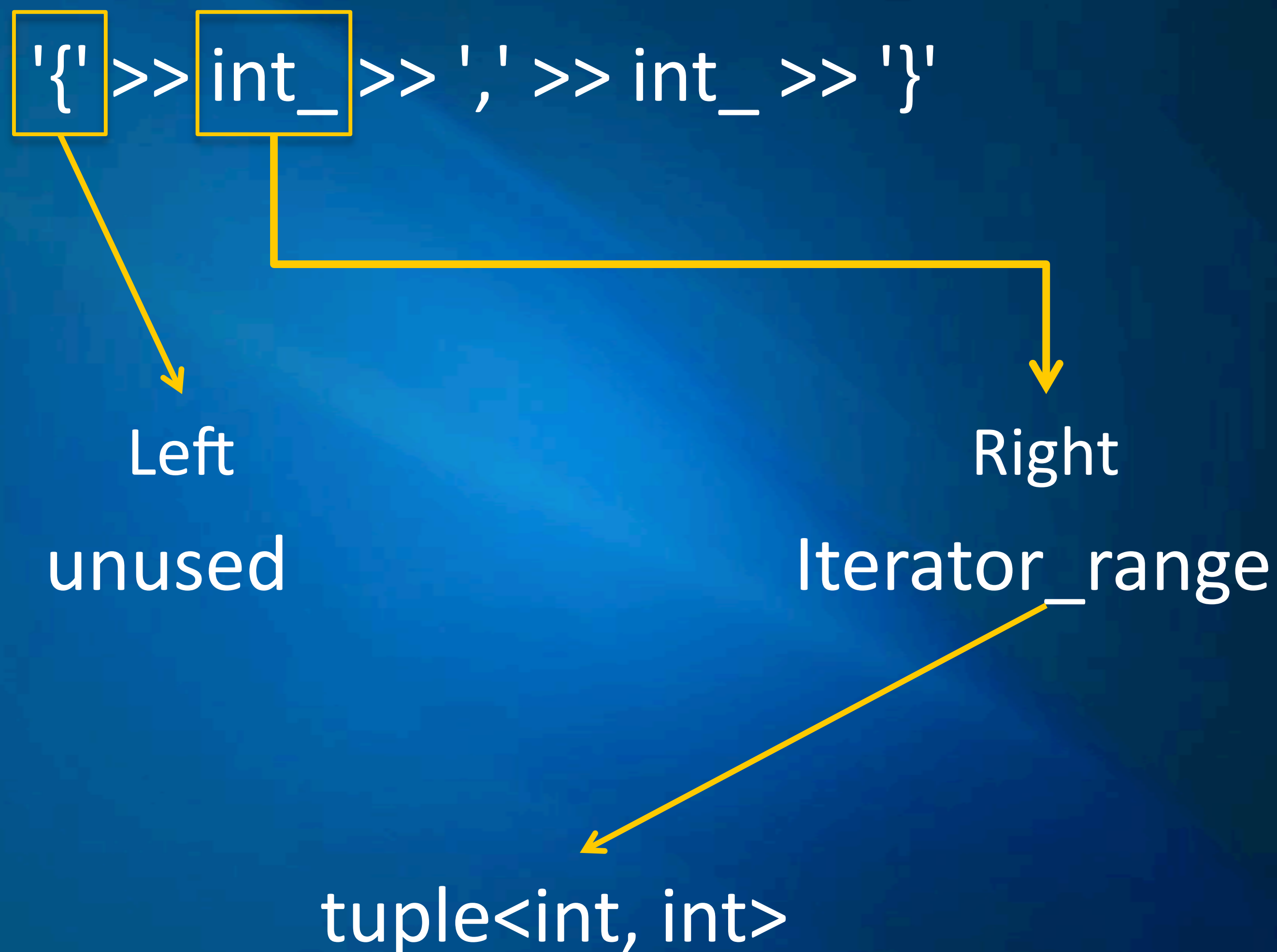
Partitioning



Partitioning



Partitioning



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

+alpha_ | (int_ >> ‘,’ >> int_) | char_

variant<char, std::string, std::pair<int, int>>

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;
```

Rule Definition

```
template <typename Iterator, typename Context, typename Attribute_>
bool parse(Iterator& first, Iterator const& last
  , Context const& context, Attribute_& attr) const
{
    rule_context<Attribute> r_context = { 0 };

    auto rule_ctx1 = make_context<rule_context_with_id_tag<ID>>(r_context, context);
    auto rule_ctx2 = make_context<rule_context_tag>(r_context, rule_ctx1);
    auto this_context = make_context<ID>(*this, rule_ctx2);

    return detail::parse_rule<attribute_type, ID>::call_rule_definition(
        rhs, name, first, last, this_context, attr, r_context.attr_ptr);
}
```

call_rule_definition

```
template <typename RHS, typename Iterator, typename Context
    , typename ActualAttribute, typename AttributePtr>
static bool call_rule_definition(
    RHS const& rhs
  , char const* rule_name
  , Iterator& first, Iterator const& last
  , Context const& context, ActualAttribute& attr, AttributePtr*& attr_ptr)
{
    typedef traits::make_attribute<Attribute, ActualAttribute> make_attribute;

    // do down-stream transformation, provides attribute for
    // rhs parser
    typedef traits::transform_attribute<
        typename make_attribute::type, Attribute, parser_id>
    transform;
```

Continued...

call_rule_definition

```
    typedef typename make_attribute::value_type value_type;
    typedef typename transform::type transform_attr;
    value_type made_attr = make_attribute::call(attr);
    transform_attr attr_ = transform::pre(made_attr);

    attr_pointer_scope<typename remove_reference<transform_attr>::type>
        attr_scope(attr_ptr, boost::addressof(attr_));
    if (parse_rhs(rhs, first, last, context, attr_))
    {
        // do up-stream transformation, this integrates the results
        // back into the original attribute value, if appropriate
        traits::post_transform(attr, attr_);

        return true;
    }
    return false;
}
```

Rule

```
template <typename ID, typename Attribute = unused_type>
struct rule : parser<rule<ID, Attribute>>
{
    typedef ID id;
    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(char const* name = "unnamed") : name(name) {}

    template <typename RHS>
    rule_definition<ID, typename extension::as_parser<RHS>::value_type, Attribute>
    operator=(RHS const& rhs) const;

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

    char const* name;
};
```

Rule

```
template <typename RHS>
rule_definition<ID, typename extension::as_parser<RHS>::value_type, Attribute>
operator=(RHS const& rhs) const
{
    typedef rule_definition<
        ID, typename extension::as_parser<RHS>::value_type, Attribute>
    result_type;

    return result_type(as_parser(rhs), name);
}
```

Rule

```
template <typename Iterator, typename Context, typename Attribute_>
bool parse(Iterator& first, Iterator const& last
  , Context const& context, Attribute_& attr) const
{
    return detail::parse_rule<attribute_type, ID>::call_from_rule(
        spirit::get<ID>(context), name
      , first, last, context, attr
      , spirit::get<rule_context_with_id_tag<ID>>(context));
}
```

Rule

```
template <typename RuleDef, typename Iterator, typename Context
    , typename ActualAttribute>
static bool call_from_rule(
    RuleDef const& rule_def
  , char const* rule_name
  , Iterator& first, Iterator const& last
  , Context const& context, ActualAttribute& attr, unused_type)
{
    // This is called when a rule-body has *not yet* been established.
    // The rule body is established by the rule_definition class, so
    // we call it to parse and establish the rule-body.

    return rule_def.parse(first, last, context, attr);
}
```

Rule

```
template <typename RuleDef, typename Iterator, typename Context
    , typename ActualAttribute, typename AttributeContext>
static bool call_from_rule(
    RuleDef const& rule_def
  , char const* rule_name
  , Iterator& first, Iterator const& last
  , Context const& context, ActualAttribute& attr, AttributeContext& attr_ctx)
{
    // This is called when a rule-body has already been established.
    // The rule body is already established by the rule_definition class,
    // we will not do it again. We'll simply call the RHS by calling
    // call_rule_definition.

    return call_rule_definition(
        rule_def.rhs, rule_name, first, last
      , context, attr, attr_ctx.attr_ptr);
}
```

X3 Calculator Grammar

```
///////////////////////////////////////////////////////////////////////////////////////
//  The calculator grammar
///////////////////////////////////////////////////////////////////////////////////////
namespace calculator_grammar
{
    using x3::uint_;
    using x3::char_;

    x3::rule<class expression, ast::program> const expression("expression");
    x3::rule<class term, ast::program> const  term("term");
    x3::rule<class factor, ast::operand> const  factor("factor");
```

X3 Calculator Grammar

```
auto const expression_def =
    term
    >>   *(  (char_('+') >> term)
           | (char_('-') >> term)
           )
    ;

auto const term_def =
    factor
    >>   *(  (char_('*') >> factor)
           | (char_('/') >> factor)
           )
  ;

auto const factor_def =
           uint_
    |      '(' >> expression >> ')'
    |      (char_('-') >> factor)
    |      (char_('+') >> factor)
 ;
```

X3 Calculator Grammar

```
auto const calculator = x3::grammar(
        "calculator"
       , expression = expression_def
       , term = term_def
       , factor = factor_def
);

} // namespace calculator_grammar end

using calculator_grammar::calculator;
```

Grammar

```
template <typename Elements>
struct grammar_parser : parser<grammar_parser<Elements>>
{
    typedef typename
        remove_reference<
            typename fusion::result_of::front<Elements>::type
        >::type::second_type
    start_rule;

    typedef typename start_rule::attribute_type attribute_type;
    static bool const has_attribute = start_rule::has_attribute;

    grammar_parser(char const* name, Elements const& elements)
     : name(name), elements(elements) {}
```

Continued...

Grammar

```
    template <typename Iterator, typename Context, typename Attribute_>
    bool parse(Iterator& first, Iterator const& last
      , Context const& context, Attribute_& attr) const
    {
        grammar_context<Elements, Context> our_context(elements, context);
        return fusion::front(elements).second.parse(first, last, our_context, attr);
    }

    char const* name;
    Elements elements;
};
```

Grammar

```
template <typename ...Elements>
grammar_parser<fusion::map<fusion::pair<typename Elements::id, Elements>...>>
grammar(char const* name, Elements const&... elements)
{
   typedef fusion::map<fusion::pair<typename Elements::id, Elements>...> sequence;
   return grammar_parser<sequence>(name,
      sequence(fusion::make_pair<typename Elements::id>(elements)...));
}
```

Grammar Context

```
template <typename Elements, typename Next>
struct grammar_context
{
    grammar_context(Elements const& elements, Next const& next)
        : elements(elements), next(next) {}

    template <typename ID>
    struct get_result
    {
        typedef typename ID::type id_type;
        typedef typename mpl::eval_if<
            fusion::result_of::has_key<Elements const, id_type>
          , fusion::result_of::at_key<Elements const, id_type>
          , typename Next::template get_result<ID>>::type
        type;
    };
```

Continued...

Grammar Context

```
    template <typename ID>
    typename get_result<ID>::type
    get(ID id) const
    {
        typedef typename ID::type id_type;
        typename fusion::result_of::has_key<Elements, id_type> has_key;
        return get_impl(id, has_key);
    }

    Elements const& elements;
    Next const& next;
};
```

Continued...

Grammar Context

```
template <typename ID>
typename get_result<ID>::type
get_impl(ID id, mpl::true_) const
{
    typedef typename ID::type id_type;
    return fusion::at_key<id_type>(elements);
}

template <typename ID>
typename get_result<ID>::type
get_impl(ID id, mpl::false_) const
{
    return next.get(id);
}
```

Semantic Actions

```
template <typename Subject, typename Action>
struct action : unary_parser<Subject, action<Subject, Action>>
{
    typedef unary_parser<Subject, action<Subject, Action>> base_type;
    static bool const is_pass_through_unary = true;
    static bool const has_action = true;

    action(Subject const& subject, Action f)
      : base_type(subject), f(f) {}

    typedef typename traits::attribute_of<Subject>::type attribute_type;

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

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

    Action f;
};
```

Semantic Actions

```
template <typename Iterator, typename Context>
bool parse(Iterator& first, Iterator const& last
  , Context const& context, unused_type) const
{
    typedef traits::make_attribute<attribute_type, unused_type> make_attribute;
    typedef traits::transform_attribute<
        typename make_attribute::type, attribute_type, parser_id>
    transform;

    // synthesize the attribute since one is not supplied
    typename make_attribute::type made_attr = make_attribute::call(unused_type());
    typename transform::type attr = transform::pre(made_attr);
    return parse(first, last, context, attr);
}
```

Semantic Actions

```
template <typename Iterator, typename Context, typename Attribute>
bool parse(Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr) const
{
    Iterator save = first;
    if (this->subject.parse(first, last, context, attr))
    {
        // call the function, passing the enclosing rule's context
        // and the subject's attribute.
        f(context, attr);
        return true;

        // reset iterators if semantic action failed the match
        // retrospectively
        first = save;
    }
    return false;
}
```

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;
```

Semantic Actions

```
struct f
{
    template <typename Context>
    void operator()(Context const& ctx, char c) const
    {
        _val(ctx) += c;         // get<rule_context_tag>(ctx).val() += c;
    }
};

std::string s;
typedef rule<class r, std::string> rule_type;

auto rdef = rule_type()
    = alpha             [f()]
    ;

BOOST_TEST(test_attr("abcdef", +rdef, s));
BOOST_TEST(s == "abcdef");
```

Semantic Actions

```
std::string s;
typedef rule<class r, std::string> rule_type;

auto rdef = rule_type()
    = alpha               [ [](auto& ctx, char c){ _val(ctx) += c; } ]
    ;
```

Generic Lambda: C++14 ?

Semantic Actions

```
std::string s;
typedef rule<class r, std::string> rule_type;
typedef rule_type::context ctx;

auto rdef = rule_type()
    = alpha            [ [](ctx r, char c){ r.val += c; } ]
    ;
```

```
template <typename Attribute>
struct rule_context_proxy
{
    template <typename Context>
    rule_context_proxy(Context& context)
      : val(_val(context)) {}
    Attribute& val;
};
```

Semantic Actions (Phoenix)

```
std::string s;
typedef rule<class r, std::string> rule_type;

auto rdef = rule_type()
    = alpha               [ _val += c ]
    ;
```

Semantic Actions (Alternative Idea)

```
class r_id {};
std::string s;
typedef rule<r_id, std::string> rule_type;

auto rdef = rule_type()
    = alpha
    ;

/*…*/

template  <typename Context, typename RHS>
void on_success(r_id, Context const& ctx, char c)
{
    _val(ctx) += c;
}
```

Wrapping Up

- Spirit X3 is Evolving
- https://github.com/djowel/spirit_x3
- Contributors! We need you!
  - Documentation / Tutorials
  - Porting Karma
  - Porting Lex
  - Testing, Benchmarks
  - Fun stuff! (Experimental Research)

THANK YOU!!!