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

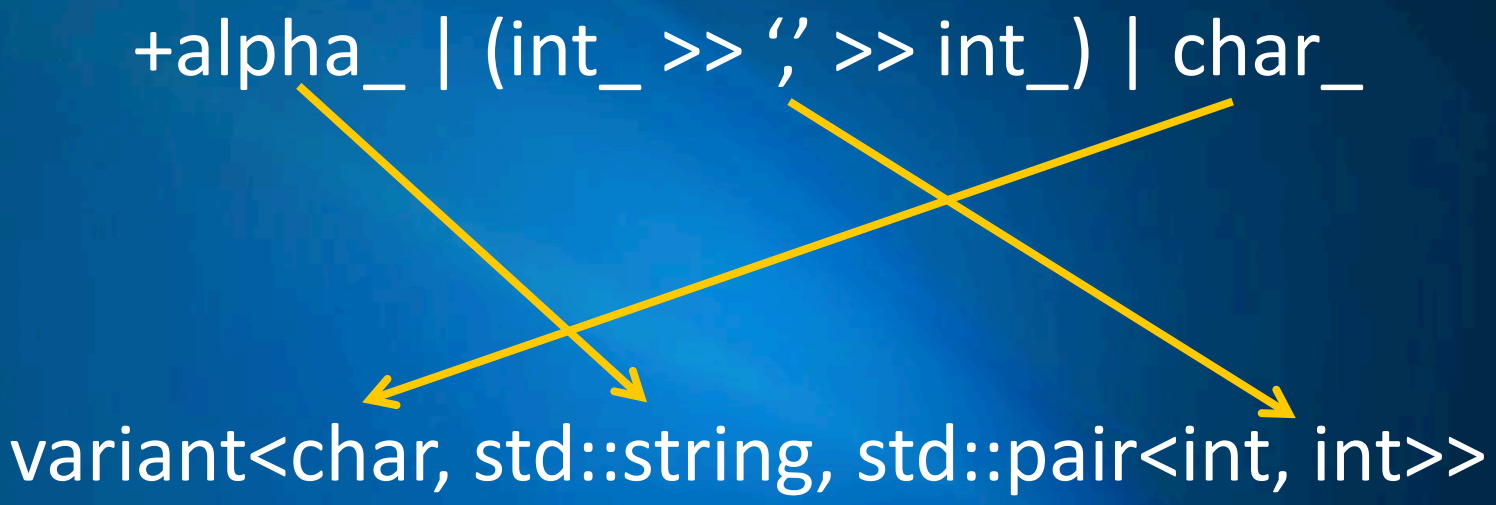
```
template <typename Iterator, typename Context, typename Attribute>
bool parse(
    Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr) const
{
    if (detail::parse_alternative(this->left, first, last, context, attr))
        return true;
    if (detail::parse_alternative(this->right,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

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!!!