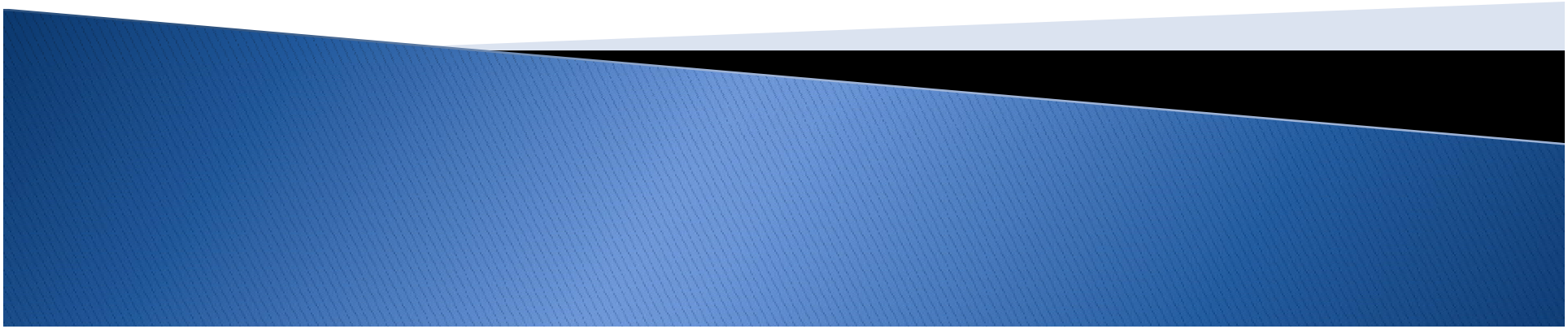
Boost.Spirit V2

Building a simple language compiler

Joel de Guzman
joel@boost-consulting.com
Boost Consulting

Hartmut Kaiser
hartmut.kaiser@gmail.com
Center for Computation and Technology
Louisiana State University



Agenda

- Overview
  - What's Spirit
  - Library Structure and Components
- Spirit.Classic
- Spirit.Qi and Spirit.Karma
  - Spirit.Qi
  - Spirit.Karma

Overview

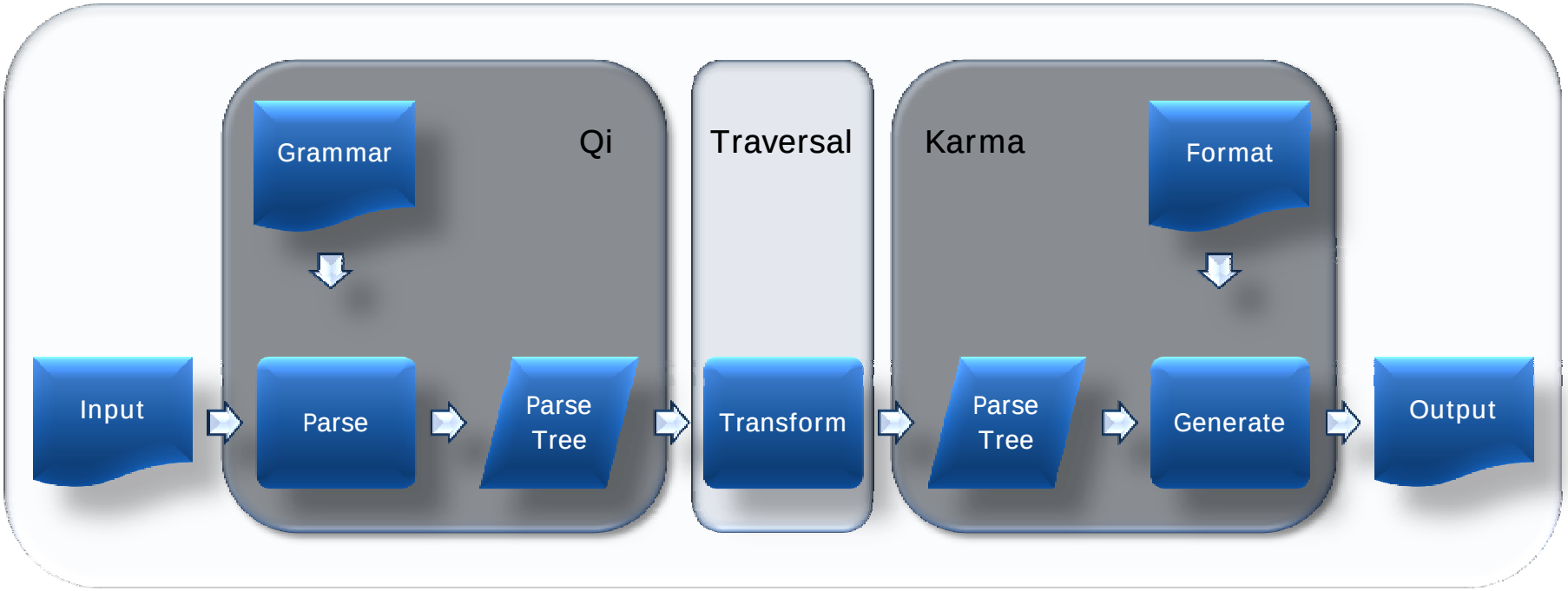
Where to get the stuff

- Spirit V2:
  - Now fully integrated with Boost SVN::trunk
  - Will be released as part of Boost V1.36
- Mailing lists:
  - Spirit mailing list: http://sourceforge.net/mail/?group_id=28447

What's Spirit

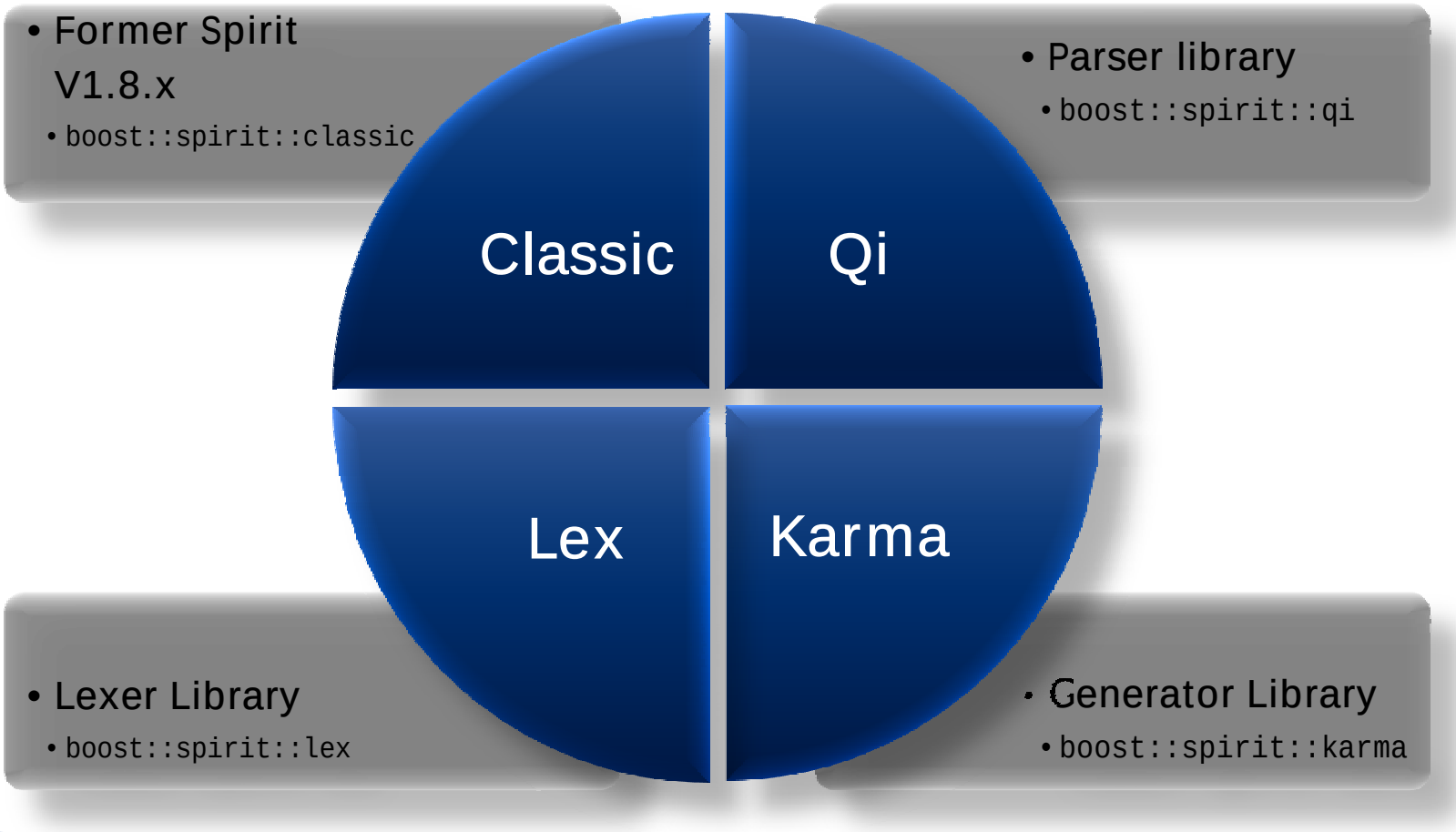
- A object oriented, recursive-descent parser and output generation library for C++
  - Implemented using template meta-programming techniques
  - Syntax of EBNF directly in C++, used for input and output format specification
- Target grammars written entirely in C++
  - No separate tools to compile grammar
  - Seamless integration with other C++ code
  - Immediately executable
- Domain Specific Embedded Languages (using Boost.Proto) for
  - Token definition (spirit::`lex`)
  - Parsing (spirit::`qi`)
  - Output generation (spirit::`karma`)

What's Spirit



- Provides two independent but well integrated components of the text processing transformation chain:
  Parsing (Qi) and Output generation (Karma)

Library Structure



Spirit Components

- Spirit Classic (`spirit::classic`)
- Create lexical analyzers (`spirit::lex`)
  - Token definition (patterns, values, lexer states)
  - Semantic actions, i.e. attach code to matched tokens
- Parsing (`spirit::qi`)
  - Grammar specification
    - Token sequence definition
    - Semantic actions, i.e. attaching code to matched sequences
    - Parsing expression grammar (PEG)
    - Error handling
- Output generation (`spirit::karma`)
  - Grammar specification
    - Same as above
  - Formatting directives
    - Alignment, whitespace delimiting, line wrapping, indentation

Spirit.Classic

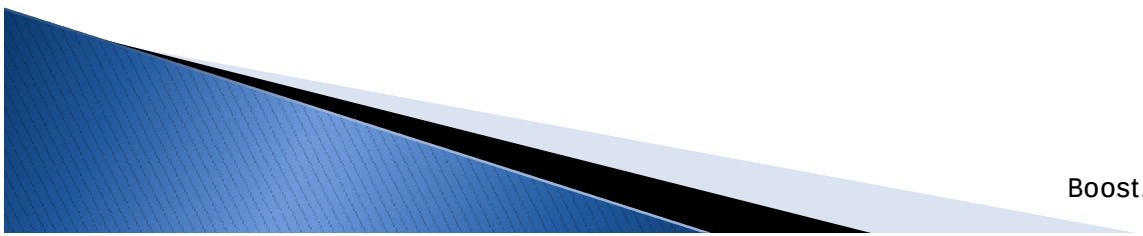
Former Spirit V1.8.x

Spirit.Classic

- Former Spirit V1.8.x
  - Existing applications compile unchanged
    - But lots of deprecated header warnings
  - To remove the warnings:
    - Included headers
      - Modify:
        ```
        #include <boost/spirit/core.hpp> to
        #include <boost/spirit/include/classic_core.hpp>
        ```
    - Namespace references
      - Change:
        ```
        namespace boost::spirit to
        namespace boost::spirit::classic
        ```
      - Or define:
        ```
        #define BOOST_SPIRIT_USE_OLD_NAMESPACE 1
        ```

Spirit.Qi and Spirit.Karma

The Yin and Yang of Parsing and Output Generation



Parsing expression grammar

- Does not require a tokenization stage
  - But it doesn't prevent it
- Similar to REs (regular expressions) being added to the Extended Backus-Naur Form
- Unlike 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 BNF
  - Greedy Loops
  - First come first serve alternates

PEG Operators

PEG	Description
a b	Sequence
a / b	Alternative
a*	Zero or more
a+	One or more
a?	Optional
&a	And-predicate
!a	Not-predicate

Spirit versus PEG Operators

PEG	Spirit
a b	Qi: a >> b Karma: a << b
a / b	a | b
a*	*a
a+	+a
&a	&a
!a	!a
a?	-a (changed from V1!)

More Spirit Operators

Syntax	Description
a || b	Sequential-or (non-shortcutting)
a - b	Difference
a % b	List
a ^ b	Permutation
a > b	Expect (Qi only)
a < b	Anchor (Karma only)
a[f]	Semantic Action

More about Parsers and Generators

- Parsers and generators are fully attributed
  - Each component either provides or expects a value of a specific type
- Currently recursive descent implementation
- Spirit makes the compiler generate format driven parser and generator routines
  - Using Boost.Proto the expression tree is directly converted into a corresponding parser/generator execution tree

Recap: Anatomy of a RD parser

```
group       = '(' >> expression >> ')';
factor      = uint_ | group;
term        = factor >> *(('*' >> factor) | ('/' >> factor));
expression  = term >> *(('+' >> term) | ('-' >> term));
```

- Elements:
  - Primitives (plain characters, `uint_`, etc.)
  - Nonterminals (`group`, `factor`, `term`, `expression`)
  - Sequences (`operator>>()`)
  - Alternatives (`operator|()`)
  - Modifiers (kleen, plus, etc.)

Recap: Anatomy of a RD parser

- Elements:
  - Primitives (plain characters, `uint_`, etc.)

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

  - Sequences (`operator>>()`)

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

  - Alternatives (`operator|()`)

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

  - Modifiers (kleen, plus, etc.)

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

  - Nonterminals (`group`, `factor`, `term`, `expression`)

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

Recap: Anatomy of a RD parser

```
// group        = '(' >> expression >> ')';
bool match_group()
{
    return
        match_sequence(
            match_sequence(
                bind(match_char, '('),
                match_expression
            ),
            bind(match_char, ')')
        );
}
```

Recap: Anatomy of a RD parser

```
// term         = factor >> *(('*' >> factor) | ('/' >> factor));
bool match_term()
{
    return
        match_sequence(
            match_factor,
            bind(match_kleene,
                bind(match_alternative,
                    bind(match_sequence, bind(match_char, '*'),
                        match_factor)),
                    bind(match_sequence, bind(match_char, '/'),
                        match_factor))
                )
            )
        );
}
```

Parser and Generator Primitives

- int_, char_, double_, ...
- lit, symbols
- alnum, alpha, digit, ...
- bin, oct, hex
- byte, word, dword, qword, ...
- stream
- typed_stream<A>
- none

Parser and Generator Directives

- Directives
  - lexeme[] (Qi), verbatim[], delimit[] (Karma)
  - nocase[] (Qi), upper_case[], lower_case[] (Karma)
  - omit[], raw[]
  - left_align[], right_align[], center[]

  - lazy(), eps, eps()

The Direct Parser API

- Parsing without skipping

```
template <typename Iterator, typename Expr>
bool parse(Iterator first, Iterator last, Expr const& p);

template <typename Iterator, typename Expr, typename Attr>
bool parse(Iterator first, Iterator last, Expr const& p,
    Attr& attr);

int i = 0; std::string str("1");
parse (str.begin(), str.end(), int_, i);
```

The Direct Parser API

- Parsing with skipping (phrase parsing)

```
template <typename Iterator, typename Expr, typename Skipper>
bool phrase_parse(Iterator& first, Iterator last,
    Expr const& xpr, Skipper const& skipper);

template <typename Iterator, typename Expr, typename Skipper,
    typename Attr>
bool phrase_parse(Iterator& first, Iterator last,
    Expr const& xpr, Attr& attr, Skipper const& skipper);

int i = 0; std::string str("  1");
phrase_parse (str.begin(), str.end(), int_, i, space);
```

The Stream based Parser API

- Parsing without skipping

```
template <typename Expr>
detail::match_manip<Expr>
    match(Expr const& xpr);

template <typename Expr, typename Attribute>
detail::match_manip<Expr, Attribute>
    match(Expr const& xpr, Attribute& attr);

int i = 0;
is >> match(int_, i);
```

The Stream based Parser API

- Parsing with skipping (phrase parsing)

```
template <typename Expr, typename Skipper>
detail::match_manip<Expr, unused_t, Skipper>
    phrase_match(Expr const& xpr, Skipper const& s);

template <typename Expr, typename Attribute,
    typename Skipper>
detail::match_manip<Expr, Attribute, Skipper>
    phrase_match(Expr const& xpr, Attribute& attr,
        Skipper const& s);

int i = 0;
is >> phrase_match(int_, i, space);
```

Parser Types and their Attributes

	Qi parser types	Attribute Type
Primitive components	• int_, char_, double_, ... • bin, oct, hex • byte, word, dword, qword, ... • stream • typed_stream<A> • symbol<A>	• int, char, double, ... • int • uint8_t, uint16_t, uint32_t, int64_t, ... • spirit::hold_any (~ boost::any) • Explicitely specified (A) • Explicitely specified (A)
Non-terminals	• rule<A()>, grammar<A()>	• Explicitely specified (A)
Operators	• *a (kleene) • +a (one or more) • -a (optional) • a % b (list) • a >> b (sequence) • a | b (alternative) • &a (predicate/eps) • !a (not predicate) • a ^ b (permutation)	• std::vector<A> • std::vector<A> • boost::optional<A> • std::vector<A> • fusion::vector<A, B> • boost::variant<A, B> • No attribute • No attribute • fusion::vector<boost::optional<A>, boost::optional<B> >
Directives	• lexeme[a], omit[a], nocase[a] ... • raw[]	• A • boost::iterator_range<Iterator>
Semantic action	• a[f]	• A

Semantic Actions

- Construct allowing to attach code to a parser component
  - Gets executed *after* the invocation of the parser
  - May *receive* values from the parser to store or manipulate
  - May use local variables or rule arguments
- Syntax:

```
int i = 0;
int_[ref(i) = _1]
```

- Easiest way to write semantic actions is phoenix
  - `_1, _2, …` refer to elements of parser
  - `_a, _b, …` refer to locals (for `rule<>`'s)
  - `_r1, _r2, …` refer to arguments (for `rule<>`'s))
  - `pass` allows to make match fail (by assigning false)

Semantic Actions

- Any function or function object can be called

```
void f(Attribute const&, Context&, bool&);
void f(Attribute const&, Context&);
void f(Attribute const&);
void f();
```

- Attribute
  - Simple parser: just the attribute value
  - Compound generator: `fusion::vector<A1, A2, …>` (AN: attributes of single parsers)
- Context
  - Normally `unused_type` (except for `rule<>`'s and `grammar<>`'s, where this is a complex data structure)
  - Can be used to access rule's locals and attributes
- bool
  - Allows to make the parser fail (by assigning `false`)

Semantic Actions

- Plain function:

```
void write(int const& i) { std::cout << i; }
int_[&write]
```

- But it's possible to use `boost::lambda`…

```
using boost::lambda::_1;
int_[std::cout << _1]
```

- … and `boost::bind` as well

```
void write(int const& i) { std::cout << i; }
int_[boost::bind(&write, _1)]
```

Building a simple VM compiler

Spirit.Qi

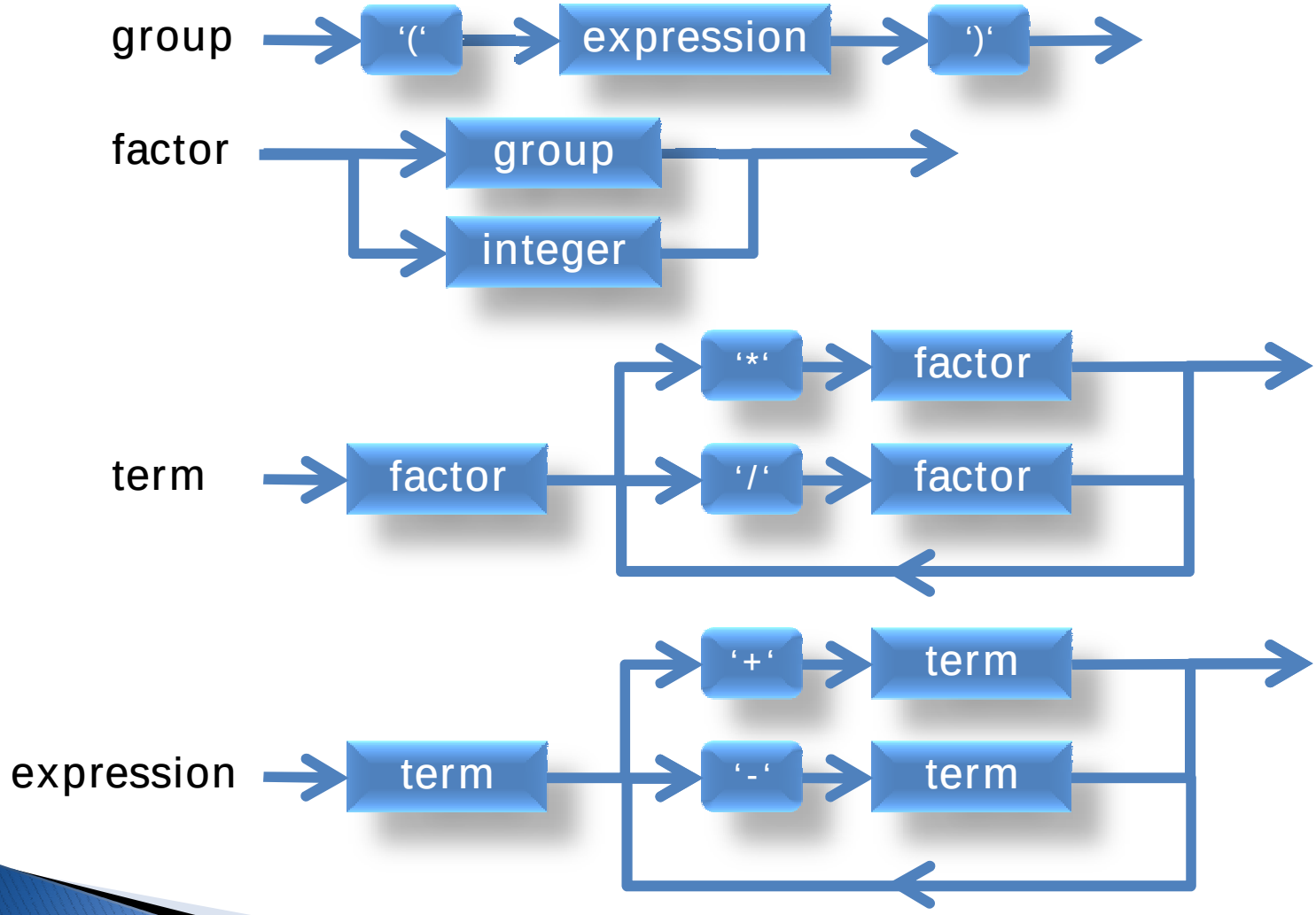
Parser Library

The humble calculator

here we go again…

```
12345
-12345
+12345
1 + 2
1 * 2
1/2 + 3/4
1 + 2 + 3 + 4
1 * 2 * 3 * 4
(1 + 2) * (3 + 4)
(-1 + 2) * (3 + -4)
1 + ((6 * 200) - 20) / 6
(1 + (2 + (3 + (4 + 5))))
```

Syntax Diagram



PEG

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

EBNF (ISO/IEC 14977)

```
group       = "(" , expression , ")";
factor      = integer | group;
term        = factor , (("*" , factor) | ("/" , factor))*;
expression  = term , (("+" , term) | ("-" , term))*;
```

Spirit
DSEL hosted by C++

```
group      = '(' >> expression >> ')';
factor     = uint_ | group;
term       = factor >> *(('*' >> factor) | ('/' >> factor));
expression = term >> *(('+' >> term) | ('-' >> term));
```

calc1.cpp

- Simple expression parser
- Syntax checker
- No semantic evaluation (no semantic actions)

```
template <typename Iterator>
struct calculator : grammar_def<Iterator, space_type>
{
    calculator()
    {
        expression =
            term
            >> *(   ('+' >> term)
                |   ('-' >> term)
                )
            ;

        term =
            factor
            >> *(   ('*' >> factor)
                |   ('/' >> factor)
                )
            ;

        factor =
            uint_
            |   '(' >> expression >> ')'
            |   ('-' >> factor)
            |   ('+' >> factor)
            ;
    }

    rule<Iterator, space_type> expression, term, factor;
};
```

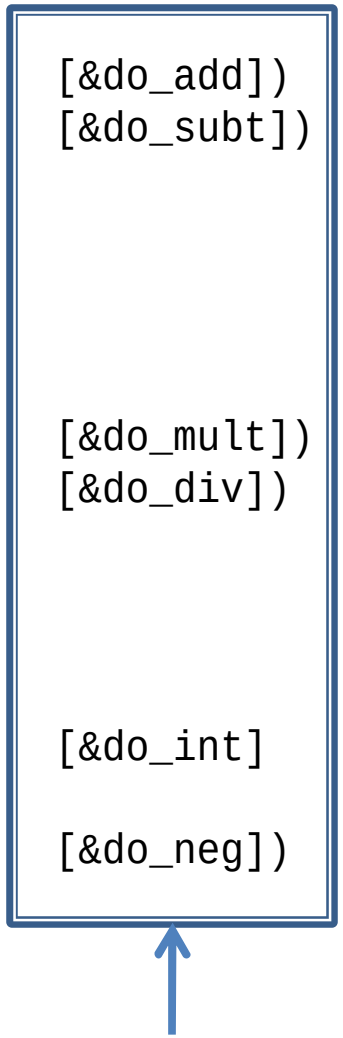
```
template <typename Iterator>
struct calculator
  : grammar_def<Iterator, space_type>
{
    calculator()
    { /*…definition here*/ }

    rule<Iterator, space_type>
        expression, term, factor;
};
```

```
typedef std::string::const_iterator iterator_type;
typedef calculator<iterator_type> calculator;

//  Our grammar definition
calculator def;

// Our grammar
grammar<calculator> calc(def, def.expression);

// Parse!
std::string::const_iterator iter = str.begin();
std::string::const_iterator end = str.end();
bool r = phrase_parse(iter, end, calc, space);
```

calc2.cpp

- Semantic actions using plain functions
- No evaluation
- Prints code suitable for a stack based virtual machine

```
expression =
    term
    >> *(   ('+' >> term          [&do_add])
        |   ('-' >> term          [&do_subt])
        )
    ;

term =
    factor
    >> *(   ('*' >> factor        [&do_mult])
        |   ('/' >> factor        [&do_div])
        )
    ;

factor =
    uint_                         [&do_int]
    |   '(' >> expression >> ')'
    |   ('-' >> factor            [&do_neg])
    |   ('+' >> factor)
    ;
```



Semantic Actions

```
void do_int(int n)
{ std::cout << "push " << n << std::endl; }

void do_add()
{ std::cout << "add\n"; }

void do_subt()
{ std::cout << "subtract\n"; }

void do_mult()
{ std::cout << "mult\n"; }

void do_div()
{ std::cout << "divide\n"; }

void do_neg()
{ std::cout << "negate\n"; }
```

```
1 * 2
```

```
push 1
push 2
mult
```

```
1 + ((6 * 200) - 20) / 6


push 1
push 6
push 200
mult
push 20
subtract
push 6
divide
add
```

calc3.cpp

- Expression evaluation
- Using phoenix semantic actions
- The parser is essentially an "interpreter" that evaluates expressions on the fly

calc2.cpp

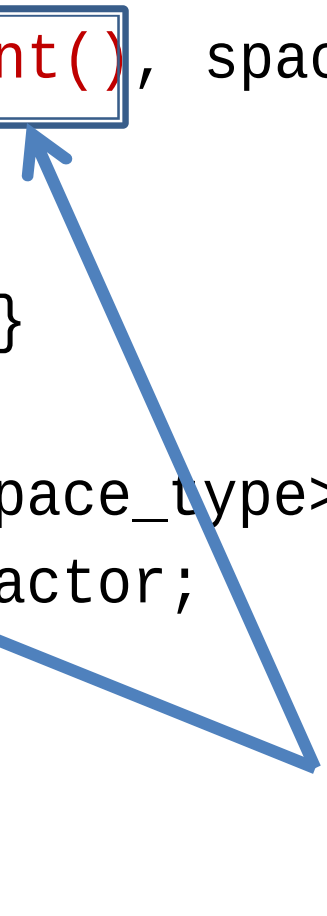
```
template <typename Iterator>
struct calculator
  : grammar_def<Iterator, space_type>
{
    calculator()
    { /*…definition here*/ }

    rule<Iterator, space_type>
        expression, term, factor;
};
```

calc3.cpp

```
template <typename Iterator>
struct calculator
  : grammar_def<Iterator, int(), space_type>
{
    calculator()
    { /*…definition here*/ }

    rule<Iterator, int(), space_type>
        expression, term, factor;
};
```

calc3.cpp

```
template <typename Iterator>
struct calculator
  : grammar_def<Iterator, int(), space_type>
{
    calculator()
    { /*…definition here*/ }

    rule<Iterator, int(), space_type>
        expression, term, factor;
};
```



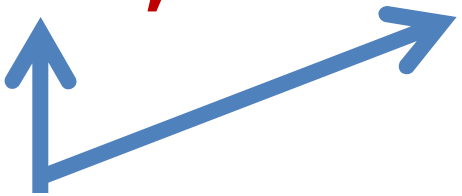
Attributes

- Synthesized Attribute == Function result
  - Used to pass semantic information up the parse tree
- Inherited Attribute == Function parameters
  - Used to pass semantic information down from parent nodes

```
int(int, char)
```

Synthesized Attribute

```
int(int, char)
```



Inherited
Attribute(s)

```
expression =
    term                            [_val = _1]
    >> *(   ('+' >> term            [_val += _1]
        |   ('-' >> term            [_val -= _1]
        )
    ;

term =
    factor                          [_val = _1]
    >> *(   ('*' >> factor          [_val *= _1]
        |   ('/' >> factor          [_val /= _1]
        )
    ;

factor =
    uint_                           [_val = _1]
    |   '(' >> expression           [_val = _1] >> ')'
    |   ('-' >> factor              [_val = -_1]
    |   ('+' >> factor              [_val = _1])
    ;
```

Semantic Actions

```
expression =
    term                            [_val = _1]
    >> *(   ('+' >> term            [_val += _1])
        |   ('-' >> term            [_val -= _1])
        )
    ;

term =
    factor                          [_val = _1]
    >> *(   ('*' >> factor          [_val *= _1])
        |   ('/' >> factor          [_val /= _1])
        )
    ;

factor =
    uint_                           [_val = _1]
    |   '(' >> expression           [_val = _1] >> ')'
    |   ('-' >> factor              [_val = -_1])
    |   ('+' >> factor              [_val = _1])
    ;
```

Synthesizied Attribute Placeholder

```
1 * 2

result = 3
```

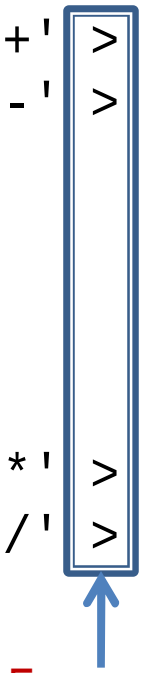
```
1 + ((6 * 200) - 20) / 6
```

```
result = 197
```

calc4.cpp

- Similar to calc3
- This time, we'll incorporate error handling and reporting

```
expression =
    term                                [_val = _1]
    >> *(   ('+' > term                 [_val += _1])
        |   ('-' > term                 [_val -= _1])
        )
    ;

term =
    factor                              [_val = _1]
    >> *(   ('*' > factor               [_val *= _1])
        |   ('/' > factor               [_val /= _1])
        )
    ;

factor =
    uint_                               [_val = _1]
    |   '(' > expression                [_val = _1] > ')'
    |   ('-' > factor                   [_val = -_1])
    |   ('+' > factor                   [_val = _1])
    ;
```



Expectations!!!

```
expression.name("expression");
term.name("term");
factor.name("factor");

on_error<fail>
(
    expression
  , std::cout
        << val("Error! Expecting ")
        << _4
        << val(" here: \"")
        << construct<std::string>(_3, _2)
        << val("\"")
        << std::endl
);
```

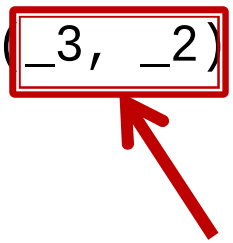
```
expression.name("expression");
term.name("term");
factor.name("factor");
```

Naming the Rules

```
on_error<fail>
(
    expression
  , std::cout
        << val("Error! Expecting ")
        << _4
        << val(" here: \"")
        << construct<std::string>(_3, _2)
        << val("\"")
        << std::endl
);
```

```
expression.name("expression");
term.name("term");
factor.name("factor");

on_error<fail>
(
    expression
  , std::cout
        << val("Error! Expecting ")
        << _4
        << val(" here: \"")
        << construct<std::string>(_3, _2)
        << val("\"")
        << std::endl
);
```

```
expression.name("expression");
term.name("term");
factor.name("factor");

on_error<fail>
(
    expression
  , std::cout
        << val("Error! Expecting ")
        << _4
        << val(" here: \"")
        << construct<std::string>(_3, _2)
        << val("\"")
        << std::endl
);
```

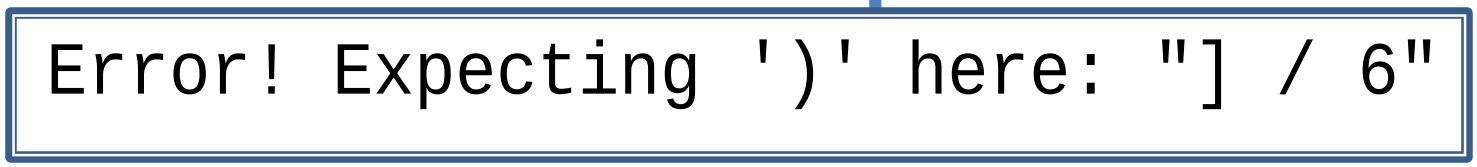


iterators to error-position and end of input

```
expression.name("expression");
term.name("term");
factor.name("factor");

on_error<fail>
(
    expression
  , std::cout
        << val("Error! Expecting ")
        << _4
        << val(" here: \"")
        << construct<std::string>(_3, _2)
        << val("\"")
        << std::endl
);
```

On Error, Fail Parsing

```
1 + ((6 * 200) - 20] / 6
```

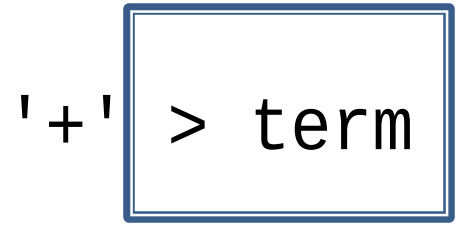


```
Error! Expecting ')' here: "] / 6"
```

```
'(' > expression > ')'
```

$#@!

```
1 + banana

Error! Expecting term here: " banana"
```



$#@!

calc5.cpp

- Compiler to a simple VM
- Same-o-same-o calculator grammar
- Uses Phoenix to compile the byte codes

```
enum byte_code
{
    op_neg,     //  negate the top stack entry
    op_add,     //  add top two stack entries
    op_sub,     //  subtract top two stack entries
    op_mul,     //  multiply top two stack entries
    op_div,     //  divide top two stack entries
    op_int,     //  push constant integer into the
                //  stack
};
```

```
class vmachine
{
public:

    vmachine(unsigned stackSize = 4096)
      : stack(stackSize)
      , stack_ptr(stack.begin())
    {
    }

    int top() const { return stack_ptr[-1]; };
    void execute(std::vector<int> const& code);

private:

    std::vector<int> stack;
    std::vector<int>::iterator stack_ptr;
};
```

```
void vmachine::execute(
    std::vector<int> const& code)
{
    std::vector<int>::const_iterator pc = code.begin();
    stack_ptr = stack.begin();

    while (pc != code.end())
    {
        switch (*pc++)
        {
            /*...*/
        }
    }
}
```

```
case op_neg:
    stack_ptr[-1] = -stack_ptr[-1];
    break;

case op_add:
    --stack_ptr;
    stack_ptr[-1] += stack_ptr[0];
    break;

case op_sub:
    --stack_ptr;
    stack_ptr[-1] -= stack_ptr[0];
    break;
```

```
case op_mul:
    --stack_ptr;
    stack_ptr[-1] *= stack_ptr[0];
    break;

case op_div:
    --stack_ptr;
    stack_ptr[-1] /= stack_ptr[0];
    break;

case op_int:
    *stack_ptr++ = *pc++;
    break;
```

```
expression =
    term
    >> *(   ('+' > term
        |   ('-' > term
        )
    ;

term =
    factor
    >> *(   ('*' > factor
        |   ('/' > factor
        )
    ;

factor =
    uint_



    |   '(' > expression > ')'
    |   ('-' > factor
    |   ('+' > factor)
    ;
```

```
[push_back(ref(code), op_add)])
[push_back(ref(code), op_sub)])




[push_back(ref(code), op_mul)])
[push_back(ref(code), op_div)])




[
    push_back(ref(code), op_int),
    push_back(ref(code), _1)
]

[push_back(ref(code), op_neg)])
```

Our Compiler

calc6.cpp

- Variables and assignment
- Symbol table
- Grammar modularization. Grammars for:
  - expression
  - Statement
- Local variables
- Synthesized and Inherited Attributes
- The auto rule!
- “raw” (transduction parsing)

The VM updated

```
enum byte_code
{
    op_neg,     //  negate the top stack entry
    op_add,     //  add top two stack entries
    op_sub,     //  subtract top two stack entries
    op_mul,     //  multiply top two stack entries
    op_div,     //  divide top two stack entries

    op_load,    //  load a variable
    op_store,   //  store a variable
    op_int,     //  push constant integer
                //  into the stack
};
```

The VM updated

```
void vmachine::execute(
    std::vector<int> const& code, int nvars)
{
    std::vector<int>::const_iterator
        pc = code.begin();
    std::vector<int>::iterator
        locals = stack.begin();
    stack_ptr = stack.begin() + nvars;

    while (pc != code.end())
    {
        /*...*/
    }
}
```

The VM updated

```
case op_load:
    *stack_ptr++ = locals[*pc++];
    break;

case op_store:
    --stack_ptr;
    locals[*pc++] = stack_ptr[0];
    break;
```

Our expression grammar and compiler

```
template <typename Iterator>
struct expression : grammar_def<Iterator, space_type>
{
    expression(
        std::vector<int>& code
      , symbols<char, int>& vars);

    rule<Iterator, space_type>
        expr, additive_expr, multiplicative_expr
      , unary_expr, primary_expr, variable;

    std::vector<int>& code;
    symbols<char, int>& vars;
    function<compile_op> op;
};
```

Our expression grammar and compiler

```
template <typename Iterator>
struct expression : grammar_def<Iterator, space_type>
{
    expression(
        std::vector<int>& code
      , symbols<char, int>& vars);

    rule<Iterator, space_type>
        expr, additive_expr, multiplicative_expr
      , unary_expr, primary_expr, variable;

    std::vector<int>& code;
    symbols<char, int>& vars;
    function<compile_op> op;
};
```

The byte codes

Our expression grammar and compiler

```
template <typename Iterator>
struct expression : grammar_def<Iterator, space_type>
{
    expression(
        std::vector<int>& code
      , symbols<char, int>& vars);

    rule<Iterator, space_type>
        expr, additive_expr, multiplicative_expr
      , unary_expr, primary_expr, variable;

    std::vector<int>& code;
    symbols<char, int>& vars;
    function<compile_op> op;
};
```

The Symbol Table

Our expression grammar and compiler

```
template <typename Iterator>
struct expression : grammar_def<Iterator, space_type>
{
    expression(
        std::vector<int>& code
      , symbols<char, int>& vars);

    rule<Iterator, space_type>
        expr, additive_expr, multiplicative_expr
      , unary_expr, primary_expr, variable;

    std::vector<int>& code;
    symbols<char, int>& vars;
    function<compile_op> op;
};
```

A Phoenix function That compiles byte codes

Our expression grammar and compiler

```
template <typename Iterator>
struct expression : grammar_def<Iterator, space_type>
{
    expression(
        std::vector<int>& code
      , symbols<char, int>& vars);

    rule<Iterator, space_type>
        expr, additive_expr, multiplicative_expr
      , unary_expr, primary_expr, variable;

    std::vector<int>& code;
    symbols<char, int>& vars;
    function<compile_op> op;
};
```

Renaming the rules and adding some more

Our expression grammar and compiler (Part 1)

```
expr =
    additive_expr.alias()
    ;

additive_expr =
    multiplicative_expr
    >> *(   ('+' > multiplicative_expr  [op(op_add)])
        |   ('-' > multiplicative_expr  [op(op_sub)])
        )
    ;

multiplicative_expr =
    unary_expr
    >> *(   ('*' > unary_expr           [op(op_mul)])
        |   ('/' > unary_expr           [op(op_div)])
        )
    ;
```

Our expression grammar and compiler (Part 2)

```
unary_expr =
        primary_expr
    |   ('-' > primary_expr             [op(op_neg)])
    |   ('+' > primary_expr)
    ;

primary_expr =
    uint_                               [op(op_int, _1)]
    |   variable
    |   '(' > expr > ')'
    ;

variable =
    (
        lexeme[
            vars
            >> !(alnum | '_')           // make sure we have whole words
        ]
    )                                   [op(op_load, _1)]
    ;
```

Our statement grammar and compiler

```
template <typename Iterator>
struct statement : grammar_def<Iterator, space_type>
{
    statement(std::vector<int>& code);

    std::vector<int>& code;
    symbols<char, int> vars;
    int nvars;

    expression<Iterator> expr_def;
    grammar<expression<Iterator> > expr;
    rule<Iterator, space_type> start, var_decl;
    rule<Iterator, std::string(), space_type> identifier;
    rule<Iterator, int(), space_type> var_ref;
    rule<Iterator, locals<int>, space_type> assignment;
    rule<Iterator, void(int), space_type> assignment_rhs;

    function<var_adder> add_var;
    function<compile_op> op;
};
```

Our statement grammar and compiler

```
template <typename Iterator>
struct statement : grammar_def<Iterator, space_type>
{
    statement(std::vector<int>& code);

    std::vector<int>& code;
    symbols<char, int> vars;
    int nvars;

    expression<Iterator> expr_def;
    grammar<expression<Iterator> > expr;
    rule<Iterator, space_type> start, var_decl;
    rule<Iterator, std::string(), space_type> identifier;
    rule<Iterator, int(), space_type> var_ref;
    rule<Iterator, locals<int>, space_type> assignment;
    rule<Iterator, void(int), space_type> assignment_rhs;

    function<var_adder> add_var;
    function<compile_op> op;
};
```

Some Phoenix functions

Our statement grammar and compiler

```
template <typename Iterator>
struct statement : grammar_def<Iterator, space_type>
{
    statement(std::vector<int>& code);

    std::vector<int>& code;
    symbols<char, int> vars;
    int nvars;

    expression<Iterator> expr_def;
    grammar<expression<Iterator> > expr;
    rule<Iterator, space_type> start, var_decl;
    rule<Iterator, std::string(), space_type> identifier;
    rule<Iterator, int(), space_type> var_ref;
    rule<Iterator, locals<int>, space_type> assignment;
    rule<Iterator, void(int), space_type> assignment_rhs;

    function<var_adder> add_var;
    function<compile_op> op;
};
```

Rule with int synthesized attribute

Our statement grammar and compiler

```
template <typename Iterator>
struct statement : grammar_def<Iterator, space_type>
{
    statement(std::vector<int>& code);

    std::vector<int>& code;
    symbols<char, int> vars;
    int nvars;

    expression<Iterator> expr_def;
    grammar<expression<Iterator> > expr;
    rule<Iterator, space_type> start, var_decl;
    rule<Iterator, std::string(), space_type> identifier;
    rule<Iterator, int(), space_type> var_ref;
    rule<Iterator, locals<int>, space_type> assignment;
    rule<Iterator, void(int), space_type> assignment_rhs;

    function<var_adder> add_var;
    function<compile_op> op;
};
```

Rule with void synthesized attribute And int inherited attribute

Our statement grammar and compiler

```
template <typename Iterator>
struct statement : grammar_def<Iterator, space_type>
{
    statement(std::vector<int>& code);

    std::vector<int>& code;
    symbols<char, int> vars;
    int nvars;

    expression<Iterator> expr_def;
    grammar<expression<Iterator> > expr;
    rule<Iterator, space_type> start, var_decl;
    rule<Iterator, std::string(), space_type> identifier;
    rule<Iterator, int(), space_type> var_ref;
    rule<Iterator, locals<int>, space_type> assignment;
    rule<Iterator, void(int), space_type> assignment_rhs;

    function<var_adder> add_var;
    function<compile_op> op;
};
```

Rule with int local variable

Our statement grammar and compiler

```
template <typename Iterator>
struct statement : grammar_def<Iterator, space_type>
{
    statement(std::vector<int>& code);

    std::vector<int>& code;
    symbols<char, int> vars;
    int nvars;

    expression<Iterator> expr_def;
    grammar<expression<Iterator> > expr;
    rule<Iterator, space_type> start, var_decl;
    rule<Iterator, std::string(), space_type> identifier;
    rule<Iterator, int(), space_type> var_ref;
    rule<Iterator, locals<int>, space_type> assignment;
    rule<Iterator, void(int), space_type> assignment_rhs;

    function<var_adder> add_var;
    function<compile_op> op;
};
```

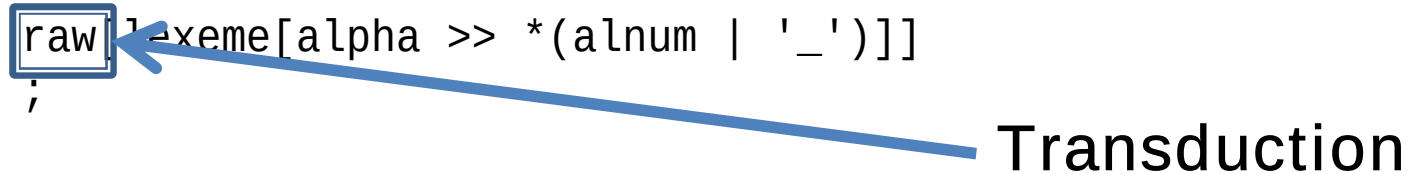
The modular expression Grammar

Our statement grammar and compiler (Part1)

```
identifier %=
    raw[lexeme[alpha >> *(alnum | '_')]]
    ;

var_ref =
    lexeme
    [
            vars        [_val = _1]
        >>  !(alnum | '_') // make sure we have whole words
    ]
    ;

var_decl =
        "var"
    >   !var_ref        // make sure the variable isn't redeclared
    >   identifier      [add_var(_1, ref(nvars))]
    >   (';' | '=' > assignment_rhs(ref(nvars)-1))
    ;
```

Our statement grammar and compiler (Part1)

```
identifier %=
    raw[lexeme[alpha >> *(alnum | '_')]]
    ;

var_ref =
    lexeme
    [
            vars        [_val = _1]
        >>  !(alnum | '_') // make sure we have whole words
    ]
    ;

var_decl =
        "var"
    >   !var_ref        // make sure the variable isn't redeclared
    >   identifier      [add_var(_1, ref(nvars))]
    >   (';' | '=' > assignment_rhs(ref(nvars)-1))
    ;
```



Our statement grammar and compiler (Part1)


```
identifier %=
    raw[lexeme[alpha >> *(alnum | '_')]]
    ;

var_ref =
    lexeme
    [
            vars        [_val = _1]
        >>  !(alnum | '_') // make sure we have whole words
    ]
    ;

var_decl =
        "var"
    >   !var_ref        // make sure the variable isn't redeclared
    >   identifier      [add_var(_1, ref(nvars))]
    >   (';' | '=' > assignment_rhs(ref(nvars)-1))
    ;
```

Our statement grammar and compiler (Part1)


```
identifier %=
    raw[lexeme[alpha >> *(alnum | '_')]]
    ;

var_ref =
    lexeme
    [
            vars        [_val = _1]
        >>  !(alnum | '_') // make sure we have whole words
    ]
    ;

var_decl =
        "var"
    >   !var_ref        // make sure the variable isn't redeclared
    >   identifier      [add_var(_1, ref(nvars))]
    >   (';' | '=' > assignment_rhs(ref(nvars)-1))
    ;
```

```
a_rule %= some-expression;
```

Is equivalent to:

```
a_rule = some-expression
         [
             val_ = _1
         ];
```

Recall:

- Parsers have a corresponding (synthesized) attribute.
- The Rule has an explicitly specified (synthesized) attribute.

Parser Types and their Attributes

	Qi parser types	Attribute Type
Primitive components	• int_, char_, double_, ... • bin, oct, hex • byte, word, dword, qword, ... • stream • typed_stream<A> • symbol<A>	• int, char, double, ... • int • uint8_t, uint16_t, uint32_t, int64_t, ... • spirit::hold_any (~ boost::any) • Explicitely specified (A) • Explicitely specified (A)
Non-terminals	• rule<A()>, grammar<A()>	• Explicitely specified (A)
Operators	• *a (kleene) • +a (one or more) • -a (optional) • a % b (list) • a >> b (sequence) • a | b (alternative) • &a (predicate/eps) • !a (not predicate) • a ^ b (permutation)	• std::vector<A> • std::vector<A> • boost::optional<A> • std::vector<A> • fusion::vector<A, B> • boost::variant<A, B> • No attribute • No attribute • fusion::vector<boost::optional<A>, boost::optional<B> >
Directives	• lexeme[a], omit[a], nocase[a] ... • raw[]	• A • boost::iterator_range<Iterator>
Semantic action	• a[f]	• A

When you have:

```
a_rule = some-expression;
```

Where the attribute of *a_rule* is compatible with the attribute of *some-expression*

You can write it as:

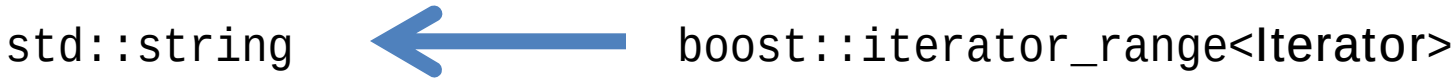
```
a_rule %= some-expression;
```

Whereby the attribute of *some-expression* is automatically assigned to *a_rule*.

No need for semantic actions!

Our statement grammar and compiler (Part1)

```
identifier %= raw[lexeme[alpha >> *(alnum | '_')]];
```



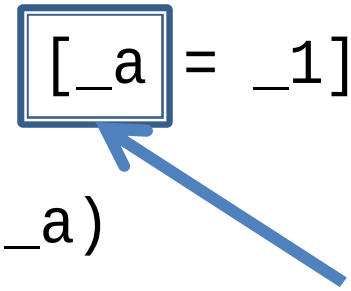
boost::iterator_range<Iterator> is compatible with std::string if Iterator points to a char. Both are STL sequences with begin/end.

Back to our statement grammar and compiler (Part1)

```
identifier %=
    raw[lexeme[alpha >> *(alnum | '_')]]
    ;

var_ref =
    lexeme
    [
            vars        [_val = _1]
        >>  !(alnum | '_') // make sure we have whole words
    ]
    ;

var_decl =
        "var"
    >   !var_ref        // make sure the variable isn't redeclared
    >   identifier      [add_var(_1, ref(nvars))]
    >   (';' | '=' > assignment_rhs(ref(nvars)-1))
    ;
```

Statement grammar and compiler (Part2)

```
assignment =
        var_ref         [_a = _1]
    >>  '='
    >   assignment_rhs(_a)
    ;

assignment_rhs =
        expr
    >   char_(';')        [op(op_store, _r1)]
    ;

start = +(var_decl | assignment);
```

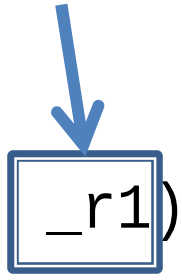


Local Variable Placeholder

Statement grammar and compiler (Part2)

```
assignment =
        var_ref             [_a = _1]
    >>  '='
    >   assignment_rhs(_a)
    ;

assignment_rhs =
        expr
    >   char_(';')           [op(op_store, _r1)]
    ;

start = +(var_decl | assignment);
```

Inherited Attribute Placeholder



calc7.cpp

- Builds on calc6
- Boolean expressions
- Control structures

The VM updated again

```
op_not,         //  boolean negate the top stack entry
op_eq,          //  compare the top two stack entries for ==
op_neq,         //  compare the top two stack entries for !=
op_lt,          //  compare the top two stack entries for <
op_lte,         //  compare the top two stack entries for <=
op_gt,          //  compare the top two stack entries for >
op_gte,         //  compare the top two stack entries for >=

op_and,         //  logical and top two stack entries
op_or,          //  logical or top two stack entries

op_true,        //  push constant 0 into the stack
op_false,       //  push constant 1 into the stack

op_jump_if,     //  jump to an absolute position in the code if top stack
                //  evaluates to false
op_jump         //  jump to an absolute position in the code
```

The VM updated again

```
case op_jump:
    pc = code.begin() + *pc;
    break;

case op_jump_if:
    if (!bool(stack_ptr[-1]))
        pc = code.begin() + *pc;
    else
        ++pc;
    --stack_ptr;
    break;
```

expression grammar updated

```
equality_expr =
    relational_expr
    >> *(   ("==" > relational_expr     [op(op_eq)])
        |   ("!=" > relational_expr     [op(op_neq)])
        )
    ;

relational_expr =
    logical_expr
    >> *(   ("<=" > logical_expr        [op(op_lte)])
        |   ('<' > logical_expr         [op(op_lt)])
        |   (">=" > logical_expr        [op(op_gte)])
        |   ('>' > logical_expr         [op(op_gt)])
        )
    ;

logical_expr =
    additive_expr
    >> *(   ("&&" > additive_expr       [op(op_and)])
        |   ("||" > additive_expr       [op(op_or)])
        )
    ;
```

statement grammar upgrade

```
rule<Iterator, space_type>
    statement_, statement_list, compound_statement
;

rule<Iterator, locals<int>, space_type> if_statement;
rule<Iterator, locals<int, int>, space_type> while_statement;
```

statement grammar upgrade (if statement part 1)

```
if_statement =
        lit("if")
    >>  '('
    >   expr            [
                            op(op_jump_if, 0), // we shall fill
                                               // this (0) in later
                            _a = size(ref(code))-1 // mark its position
                        ]
    >   ')'
    >   statement_      [
                            // now we know where to jump to
                            // (after the if branch)
                            ref(code)[_a] = size(ref(code))
                        ]
    >>
```

More... (else branch)

statement grammar upgrade (if statement part 2: else branch)

```
    >>
       -(
            lexeme[
                "else"
                >> !(alnum | '_') // make sure we have whole words
            ]
                        [
                            ref(code)[_a] += 2, // adjust for the "else" jump
                            op(op_jump, 0), // we shall fill this (0) in later
                            _a = size(ref(code))-1 // mark its position
                        ]
        >   statement_  [
                            // now we know where to jump to
                            // (after the else branch)
                            ref(code)[_a] = size(ref(code))
                        ]
        )
    ;
```

statement grammar upgrade (2)

```
while_statement =
        lit("while")    [
                            _a = size(ref(code)) // mark our position
                        ]
    >>  '('
    >   expr            [
                            op(op_jump_if, 0), // we shall fill this
                                               // (0) in later
                            _b = size(ref(code))-1 // mark its position
                        ]
    >   ')'
    >   statement_      [
                            op(op_jump, _a), // loop back
                            // now we know where to jump to (to exit the loop)
                            ref(code)[_b] = size(ref(code))
                        ]
    ;
```

statement grammar upgrade (3)

```
compound_statement =
    '{' >> -statement_list >> '}'
    ;

statement_ =
        var_decl
    |   assignment
    |   compound_statement
    |   if_statement
    |   while_statement
    ;

statement_list = +statement_;
```

The mini_c compiler
"I was once a calculator"

- Not a calculator anymore!
- Full fledged minimal C-like programming language
- Builds on top of calc7
- KISS: No types other than ints

The mini_c VM

```
op_stk_adj,     // adjust the stack (for args and locals)
op_call,        // function call
op_return       // return from function
```

The mini_c VM

```
int vmachine::execute(
    std::vector<int> const& code
  , std::vector<int>::const_iterator pc
  , std::vector<int>::iterator frame_ptr
)
{
    std::vector<int>::iterator stack_ptr = frame_ptr;

    while (true)
    {
        BOOST_ASSERT(pc != code.end());

        switch (*pc++)
        {
            /*...*/
        }
    }
}
```

The mini_c VM

```
case op_stk_adj:
    stack_ptr += *pc++;
    break;

case op_return:
    return stack_ptr[-1];
```

The mini_c VM

```
case op_call:
    {
        int nargs = *pc++;
        int jump = *pc++;

        // a function call is a recursive call to execute
        int r = execute(
            code
          , code.begin() + jump
          , stack_ptr - nargs
        );

        // cleanup after return from function
        stack_ptr[-nargs] = r;      //  get return value
        stack_ptr -= (nargs - 1);   //  the stack will now contain
                                    //  the return value
    }
    break;
```

The mini_c skipper grammar

```
template <typename Iterator>
struct white_space_def : grammar_def<Iterator>
{
    white_space_def()
    {
        start =
                space                                   // tab/space/cr/lf
            |   "/*" >> *(char_ - "*/") >> "*/"         // C-style comments
            ;
    }

    rule<Iterator> start;
};
```

The mini_c expression grammar functions!:

```
typedef grammar<white_space_def<Iterator> > white_space;
rule<Iterator, locals<function_info, int>, white_space> function_call;
symbols<char, function_info>& functions;
```

```
struct function_info
{
    int arity;
    int address;
};
```

The mini_c expression grammar function call:

```
typedef grammar<white_space_def<Iterator> > white_space;
rule<Iterator, locals<function_info, int>, white_space> function_call;
symbols<char, function_info>& functions;
```

```
function_call =
    functions                           [_a = _1]
    >>  '('
    >> -(
            expr                        [++_b]
            >> *(',' > expr             [++_b])
        )
    >   char_(')')                      [op(_a, _b, pass)]
    ;
```

The mini_c expression grammar functions call:

```
// special overload for function calls
void operator()(
    function_info const& info, int got_nargs, bool& parse_result) const
{
    if (got_nargs == info.arity)
    {
        code.push_back(op_call);
        code.push_back(info.arity);
        code.push_back(info.address);
    }
    else
    {
        parse_result = false; // fail the parse
        std::cerr << "wrong number of args" << std::endl;
    }
}
```

The mini_c statement grammar functions return:

```
symbols<char, function_info>& functions;
int nvars;
bool has_return;

rule<Iterator, white_space> return_statement;
```

```
return_statement =
        lexeme[
            "return"
            >> !(alnum | '_') // make sure we have whole words
        ]
    >> -(
            eps(ref(has_return)) > expr     [op(op_return)]
        )
    >   ';'
    ;
```

The mini_c program grammar

```
template <typename Iterator>
struct program : grammar_def<Iterator, grammar<white_space_def<Iterator> > >
{
    program(std::vector<int>& code);

    type

    std:
    rule
    rule

    type

    func

    symb
    stat
    gram

    rule
    boos
    boos
    boos
};
```

```
typedef
    locals<
        std::string // function name
      , int         // address
    >
function_locals;

symbols<char, function_info> functions;
statement<Iterator> statement_def;
grammar<statement<Iterator> > statement;

rule<Iterator, function_locals, white_space> function;
boost::phoenix::function<function_adder> add_function;
boost::phoenix::function<function_state_reset> state_reset;
boost::phoenix::function<compile_op> op;
```

The mini_c program grammar

```
function =
        (
            lit("void")                     [ref(has_return) = false]
        |   lit("int")                      [ref(has_return) = true]
        )
    >>  !functions                          // no duplicate functions!
    >>  identifier                          [_a = _1]
    >>  '('
    >  -(
            identifier                      [add_var(_1)]
            >> *(',' > identifier           [add_var(_1)])
        )
    >   ')'
    ;
```

The mini_c program grammar

```
function =
        (
            lit("void")                     [ref(has_return) = false]

    >>
    >>  >   char_('{')                      [
    >>                                          _b = size(ref(code)),
    >                                           add_function(
                                                    _a      // function name
                                                  , ref(nvars)      // arity
    >                                             , size(ref(code)) // address
    >                                           ),
                                                op(op_stk_adj, 0)   // adjust this later
                                            ]
        >   statement
        >   char_('}')                      [state_reset(_b)]
        ;

    >
    ;
```

That's it! ;–)

mini_c sample (1)

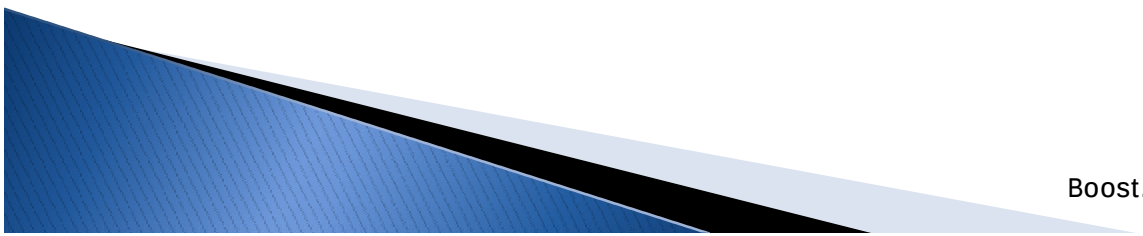
```
/* My first mini program */

int pow2(n)
{
    int a = 2;
    int i = 1;
    while (i < n)
    {
        a = a * 2;
        i = i + 1;
    }
    return a;
}

int main()
{
    return pow2(10);
}
```

mini_c sample (1)

```
/* The factorial */

int factorial(n)
{
    if (n <= 0)
        return 1;
    else
        return n * factorial(n-1);
}

int main(n)
{
    return factorial(n);
}
```

Spirit.Karma

Generator Library



Output Generation

- Karma is the Yang to Qi's Yin
  - Everything you know about Qi's parsers is still true but has to be applied upside down
- Qi gets input from input iterators, Karma outputs the generated data using a output iterator
- Qi uses operator>>(), Karma uses operator<<()
- Qi's semantic actions receive a value, Karma's semantic actions provide one
- Qi's parser attributes are passed up, Karma's attributes are passed down

Comparison Qi/Karma

	Qi	Karma
Main component	parser	generator
Main routine	parse(), phrase_parse()	generate(), generate_delimited()
Primitive components	• int_, char_, double_, ... • bin, oct, hex • byte, word, dword, qword, ... • stream	• int_, char_, double_, ... • bin, oct, hex • byte, word, dword, qword, ... • stream
Non-terminals	• rule, grammar	• rule, grammar
Operators	• * (kleene) • + (one or more) • - (optional) • % (list) • >> (sequence) • | (alternative) • & (predicate/eps) • ! (not predicate) • ^ (permutation)	• * (kleene) • + (one or more) • - (optional) • % (list) • << (sequence) • | (alternative) • & (predicate/eps) • ! (not predicate)
Directives	• lexeme[], omit[], raw[] • nocase[]	• verbatim[], delimit[] • left_align[], center[], right_align[] • upper[], lower[], wrap[], indent[]
Semantic Action	receives value	provides value

Comparison Qi/Karma

	Qi	Karma
Rule and grammar definition	rule<Iterator, Sig, Locals> grammar<Iterator, Sig, Locals> Iterator: input iterator Sig: T(...)	rule<OutIter, Sig, Locals> grammar<OutIter, Sig, Locals> OutIter: output iterator Sig: void(...) (no return value)
Placeholders	Inherited attributes: _r1, _r2, ... Locals: _a, _b, ... Synthesised attribute: _val References to components: _1, _2 ...	Inherited attributes: _r1, _r2, ... Locals: _a, _b, ... References to components: _1, _2 ...
Semantic actions	int_[ref(i) = _1] (char_ >> int_) [ref(c) = _1, ref(i) = _2]	int_[_1 = ref(i)] (char_ << int_) [_1 = ref(c), _2 = ref(i)]
Attributes	• Return type (attribute) of a parser component is the type of the values it generates, it must be convertible to the target type. • Attributes are propagated up. • Attributes are passed as non-const& • Parser components may not have target attribute value	• The attribute of a generator component is the type of the values it expects, i.e. the provided value must be convertible to this type. • Attributes are passed down. • Attributes are passed as const& • Generator components need always a ‚source' value: either literal or parameter

Output generation

- Karma is a library for flexible generation of arbitrary character sequences
  - Based on the idea, that a grammar usable to parse an input sequence may as well be used to generate the very same sequence in the output
  - For parsing of some input most programmers use parser generator tools
  - Need similar tools: ‘unparser generators’
- Karma is such a tool
  - Inspired by the StringTemplate library (ANTLR)
  - Allows strict model-view separation (Separation of format and data)
  - Defines a DSEL (domain specific embedded language) allowing to specify the structure of the output to generate in a language derived from PEG

Output generation

- DSEL was modeled after the PEG as used for parsing, i.e. set of rules describing what output is generated in what sequence:

```
int_(10) << lit("123") << char_('c')          // 10123c

(int_ << lit)[_1 = val(10), _2 = val("123")]  // 10123

vector<int> v = { 1, 2, 3 };
(*int_)[_1 = ref(v)]                          // 123

(int_ % ",") [_1 = ref(v)]                    // 1,2,3
```

Output Generation

- Three ways of associating data (values) with formating rules:
  - Using literals, direct association of values
    - `int_(10), char_('c')`
  - Explicit passing of values to API functions
    - Direct generator API
    - Stream based generator API
  - Semantic actions, direct assignment of values
    - `int_[_1 = val(10)]`

The Direct Generator API

- Generating output without delimiters

```
template <typename OutputIterator, typename Expr>
bool generate(OutputIterator first, Expr const& p);

template <typename OutputIterator, typename Expr,
  typename Attribute>
bool generate(OutputIterator first, Expr const& p,
  Attribute const& attr);

int i = 42;
generate (sink, int_, i);  // outputs: "42"
```

The Direct Generator API

- Generating output using delimiters

```
template <typename OutputIterator, typename Expr,
  typename Delimiter>
bool generate(OutputIterator first, Expr const& p,
  Delimiter const& delim);

template <typename OutputIterator, typename Expr,
  typename Attribute, typename Delimiter>
bool generate(OutputIterator first, Expr const& p,
  Attribute const& attr, Delimiter const& delim);

int i = 42;
generate (sink, int_, i, space);  // outputs: "42 "
```

The Stream based Generator API

- Generating output without delimiters

```
template <typename Expr>
format_manip<Expr>
  format(Expr const& xpr);

template <typename Expr, typename Attribute>
format_manip<Expr, Attribute>
  format(Expr const& xpr, Attribute& attr);

int i = 42;
os << format(int_, i);    // outputs: "42"
```

The Stream based Generator API

- Generating output using delimiters

```
template <typename Expr, typename Delimiter>
format_manip<Expr>
  format_delimited(Expr const& xpr, Delimiter const& d);

template <typename Expr, typename Attribute,
  typename Delimiter>
format_manip<Expr, Attribute>
  format_delimited(Expr const& xpr, Attribute& attr,
    Delimiter const& d);

int i = 42;
os << format_delimited(int_, i, space);   // "42 "
```

Generator Types and their Attributes

	Karma generator types	AttributeType
Primitive components	• int_, char_, double_, ... • bin, oct, hex • byte, word, dword, qword, ... • stream	• int, char, double, ... • int • uint8_t, uint16_t, uint32_t, uint64_t, ... • spirit::hold_any (~ boost::any)
Non-terminals	• rule<void(A)>, grammar<void(A)>	• Explicitely specified (A)
Operators	• *a (kleene) • +a (one or more) • -a (optional) • a % b (list) • a << b (sequence) • a | b (alternative)	• std::vector<A> (std container) • std::vector<A> (std container) • boost::optional<A> • std::vector<A> (std container) • fusion::vector<A, B> (sequence) • boost::variant<A, B>
Directives	• verbatim[a], delimit(...)[a] • lower[a], upper[a] • left_align[a], center[a], right_align[a] • wrap[a], indent[a] (TBD)	• A • A • A • A
Semantic action	• a[f]	• A

Different Output Grammars

Different output formats for:	`std::vector<int>`
*int_ '[' << *int_ << ']'	Without any separation
int_ % ", " '[' << (int_ % ", ") << ']'	Comma separated
'[' << (('(' << int_ << ')') % ", ") << ']'	Comma separated, enclosed in parenthesis
" " << *verbatim[" " << int_ << " "] << " "	HTML bullet list
"|" << right_align[int_] << "|"	Right aligned in a column

Different Data Structures

Different data structures for: `'[' << (stream % ", ") << ']'`	
`int i[4];`	C style arrays
`std::vector<int>`	STL containers
`std::list<char>`	
`std::vector<boost::gregorian::date>`	
`std::string, std::wstring`	Strings
`boost::iterator_range<…>`	Boost data structures
`boost::array<long, 20>`	

Semantic Actions

- Construct allowing to attach code to a generator component
  - Gets executed *before* the invocation of the generator
  - May *provide* values for the generators to output
  - May use local variables or rule arguments
- Syntax similar to parser semantic actions

```
int i = 4;
int_[_1 = ref(i)]
```

- Easiest way to write semantic actions is phoenix
  - `_1, _2, …` refer to elements of generator
  - `_a, _b, …` refer to locals (for `rule<>`'s)
  - `_r1, _r2, …` refer to arguments (for `rule<>`'s))
  - `pass` allows to make match fail (by assigning false)

Semantic Actions

- Any function or function object can be called

```
void f(Attribute&, Context&, bool&);
void f(Attribute&, Context&);
void f(Attribute&);
void f();
```

- Attribute
  - Simple generator: just the attribute value
  - Compound generator: `fusion::vector<A1, A2, …>` (AN: attributes of generators)
- Context
  - Normally `unused_type` (except for `rule<>`'s and `grammar<>`'s, where this is a complex data structure)
  - Can be used to access rule's locals and attributes
- bool
  - Allows to make the generator fail (by assigning `false`)

Semantic Actions

- Plain function:

```
void read(int& i) { i = 42; }
int_[&read]
```

- But it's possible to use `boost::lambda`…

```
using boost::lambda::_1;
int_[_1 = 42]
```

- … and `boost::bind` as well

```
void read(int& i) { i = 42; }
int_[boost::bind(&read, _1)]
```

Formatting directives

- Alignment
  - Using spaces (default width: `BOOST_KARMA_DEFAULT_FIELD_LENGTH`):

```
left_align[…some generator…]
right_align[…some generator…]
center[…some generator…]

right_align[int_(12345)]                    // '     12345'
```

  - Using any generator, any width:

```
left_align(…)[…some generator…]
left_align(width, …)[…some generator…]

right_align(char_('*'))[int_(12345)]    // '*****12345'
right_align(6, char_('*'))[int_(12345)] // '*12345'
```

Formatting directives

- Switch between delimited and non-delimited modes

```
delimit[…some generator…]      // uses spaces
delimit(…)[…some generator…]   // uses given generator

verbatim[…some generator…]     // no delimiting

delimit[int_(12345)]                      // '12345 '
delimit(char_(','))[int_(12345)]          // '12345,'

vector<int> v = { 1, 2, 3 };
(*delimit(char_(','))[int_])
    [_1 = ref(v)]                         // '1,2,3,'
```

Formatting directives

- Case management

```
upper_case[…some generator…]  // upper case
lower_case[…some generator…]  // lower case

// not only obvious conversions, but also
upper_case[double_(1.8e20)]   // 1.8E20
lower_case[double_(1.8e20)]   // 1.8e20

upper_case[hex(0xaBcD)]       // ABCD
lower_case[hex(0xaBcD)]       // abcd

// influences nan, inf as well (NAN, INF)
```

Doing something useful

- Generating output from an AST produced by a calculator parser:

```
expression =
            term                            [_val = _1]
            >> *(   ('+' >> term            [_val += _1])
                |   ('-' >> term            [_val -= _1])
                )
            ;

        term =
            factor                          [_val = _1]
            >> *(   ('*' >> factor          [_val *= _1])
                |   ('/' >> factor          [_val /= _1])
                )
            ;

        factor =
            uint_                           [_val = _1]
            |   '(' >> expression           [_val = _1] >> ')'
            |   ('-' >> factor              [_val = neg(_1)])
            |   ('+' >> factor              [_val = pos(_1)])
            ;
```

Doing something useful

- Here is the AST:

```
struct expression_ast
{
    typedef
        boost::variant<
            int,
            boost::recursive_wrapper<binary_op>,
            boost::recursive_wrapper<unary_op>
        >
    type;

    type expr;
};
```

Doing something useful

- Output formatting using a grammar:

```
ast_node %=
        int_        [_1 = _int(_r0)]
    |   binary_node [_1 = _bin_op(_r0)]
    |   unary_node  [_1 = _unary_op(_r0)]
    ;

binary_node =
        ('(' << ast_node << char_ << ast_node << ')')
        [
            _1 = _left(_r0), _2 = _op(_r0), _3 = _right(_r0)
        ]
    ;

unary_node =
        ('(' << char_ << ast_node << ')')
        [
            _1 = _op(_r0), _2 = _right(_r0)
        ]
    ;
```

Doing something useful

- Output formatting using a grammar:

```
ast_node %=
        int_        [_1 = _int(_r0)]
    |   binary_node [_1 = _bin_op(_r0)]
    |   unary_node  [_1 = _unary_op(_r0)]
    ;

binary_node =
        (ast_node << ast_node << char_)
        [
            _1 = _left(_r0), _2 = _right(_r0), _3 = _op(_r0)
        ]
    ;

unary_node =
        verbatim['(' << ast_node << char_ << ')']
        [
            _1 = _right(_r0), _2 = _op(_r0)
        ]
    ;
```

Future Directions

- Spirit.Qi
  - Deferred actions
  - Packrat Parsing (memoization)
  - LL1 deterministic parsing
  - Transduction Parsing (micro-spirit)
- Spirit.Karma
  - More output formatting
  - Integration with layout engines
- Alchemy:
  - parse tree transformation based on Dan Marsden's Traversal library