Boost Spirit V2

A cookbook style guide to parsing and output generation in C++

Joel de Guzman (joel@boost-consulting.com)
Hartmut Kaiser (hartmut.kaiser@gmail.com)

Outline

- Introduction:
  - What‘s Spirit
  - Spirit Components
  - General features,
    - What’s new, what’s different
  - PEG compared to EBNF, semantic actions
- Cookbook Guide
  - Parsing
  - Lexing
  - Output generation
- Conclusions, Questions, Discussion
  - Compile time issues

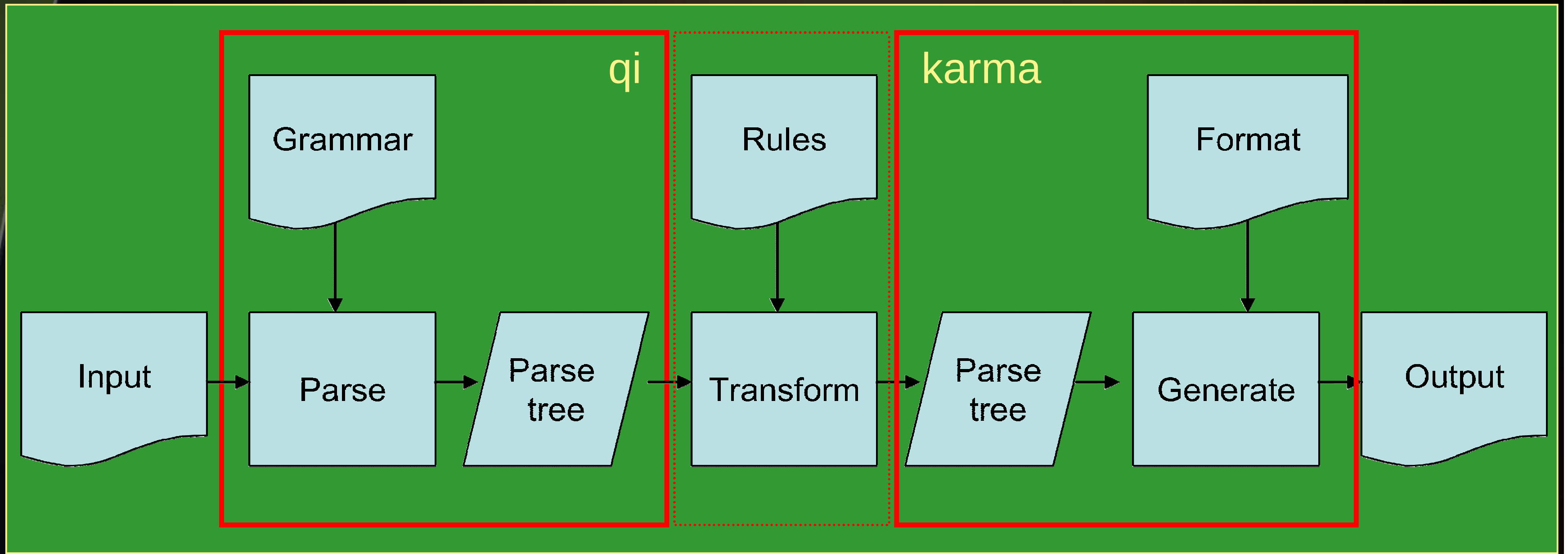
Where to get the stuff

- Spirit2:
  - Snapshot:
    - http://spirit.sf.net/dl_more/spirit2.zip
  - https://svn.sourceforge.net/svnroot/spirit/trunk/final/
  - Boost CVS::HEAD (we rely on the latest Fusion and Proto libraries)
- Mailing lists:
  - http://sourceforge.net/mail/?group_id=28447

What's Spirit

- A object oriented, recursive-descent parser and output generation framework 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 tool to compile grammar
  - Seamless integration with other C++ code
  - Immediately executable
- Domain Specific Embedded Language for
  - Token definition (`lex`)
  - Parsing (`qi`)
  - Output generation (`karma`)

What’s Spirit

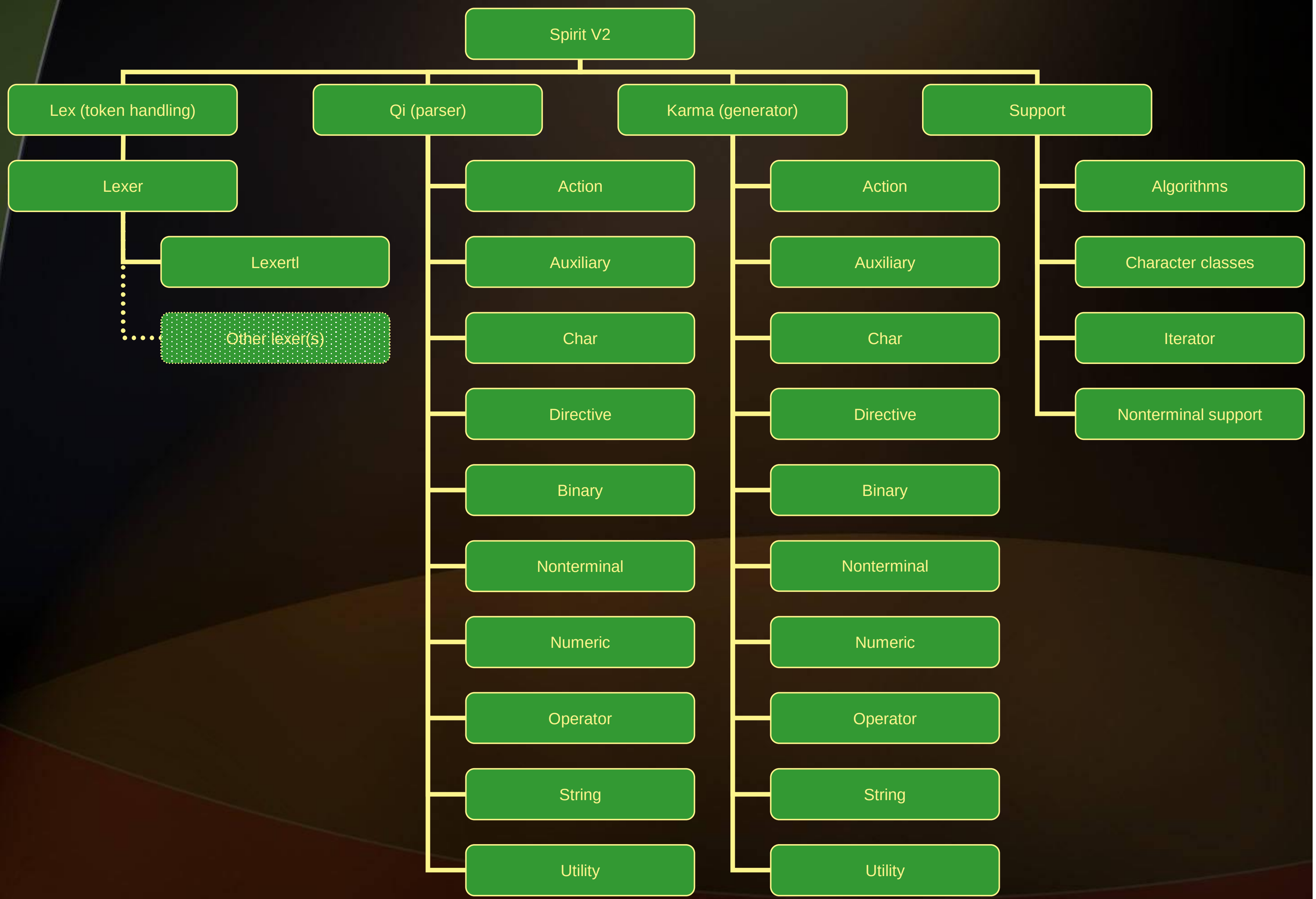


- Provides two main components of the text processing transformation chain:
  - Parsing (`spirit::qi`, `spirit::lex`)
  - Output generation (`spirit::karma`)
- Both parts are independent, but well integrated

Spirit Components

- Parsing (`spirit::qi`)
  - Token definition (`spirit::lex`)
  - Grammar specification
    - Token sequence definition
    - Semantic actions, i.e. attaching code to matched sequences
    - Parsing expression grammar
    - Attribute grammar
    - Error handling
- Output generation (`spirit::karma`)
  - Grammar specification
    - Same as above
  - Formatting directives
    - Alignment, whitespace delimiting, line wrapping, indentation

Spirits Modular Structure



The Spirit Parser components

Parsing expression grammar

- Represents a recursive descent parser
- Similar to REs (regular expressions) and the Extended Backus-Naur Form
- Different Interpretation
  - Greedy Loops
  - First come first serve alternates
- Does not require a tokenization stage

PEG Operators

`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

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

Primitives

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

Directives

- Lexeme[]
- omit[]
- nocase[]
- raw[]

Auxiliary

- eps[]
- functor
- lazy

The Direct Parse 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);
```

- 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 Parse 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);
```

- 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

<table>
<tr><th><</th><th>Qi parser types</th><th>Attribute Type</th></tr>
<tr><td>Primitive components</td><td>• int_, char_, double_, …
• bin, oct, hex
• byte, word, dword, qword, …

• stream
• typed_stream<A>
• symbol<A></td><td>• int, char, double, …
• int
• uint8_t, uint16_t, uint32_t, uint64_t, …
• spirit::hold_any (~ boost::any)
• Explicitely specified (A)
• Explicitely specified (A)</td></tr>
<tr><td>Non-terminals</td><td>• rule<A()>, grammar<A()></td><td>• Explicitely specified (A)</td></tr>
<tr><td>Operators</td><td>• *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)</td><td>• 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>
></td></tr>
<tr><td>Directives</td><td>• lexeme[a], omit[a], nocase[a] …
• raw[]</td><td>• A
• boost::iterator_range<Iterator></td></tr>
<tr><td>Semantic action</td><td>• a[f]</td><td>• A</td></tr>
</table>

The Qi Cookbook

The Qi Cookbook

- Simple examples
  - Sum
  - Number lists
  - Complex number
  - Roman numerals
  - Mini XML
- Let’s build a Mini-C Interpreter
  - Expression evaluator
  - With semantic actions
  - Error handling and reporting
  - Virtual Machine
  - Variables and assignment
  - Control statements
  - Not a calculator anymore

Sum

```
double_
    >> *(',' >> double_)
```

Sum

```
double_[ref(n) = _1]
    >> *(',' >> double_[ref(n) += _1])
```

Sum

```
double_[ref(n) = _1]
    >> *(',' >> double_[ref(n) += _1])
```

Semantic Actions

Sum

```
double_[ref(n) = _1]
    >> *(',' >> double_[ref(n) += _1])
```

_1: Placeholder
(result of the parser)

Number List

```
double_[push_back(ref(v), _1)]
>> *(',' >> double_[push_back(ref(v), _1)])
```

Number List

```
double_[push_back(ref(v), _1)] % ','
```

Number List

```
double_ % ','
```

Attribute

std::vector<double>

Complex Number

```
    '(' >> double_
        >> -(',' >> double_) >> ')'
|   double_
```

Complex Number

```
  '(' >> double_[ref(rN) = _1]
      >> -(',' >> double_[ref(iN) = _1]) >> ')'
| double_[ref(rN) = _1]
```

Complex Number

```
template <typename Iterator>
bool parse_complex(Iterator first, Iterator last, std::complex<double>& c)
{
    double rN = 0.0;
    double iN = 0.0;
    bool r = phrase_parse(first, last,

        //  Begin grammar
        (
                '(' >> double_[ref(rN) = _1]
                    >> -(',' >> double_[ref(iN) = _1]) >> ')'
            |   double_[ref(rN) = _1]
        )
        ,
        //  End grammar

        space);

    if (first != last)    // fail if we did not get a full match
        return false;
    return r;
}
```

Complex Number

```
template <typename Iterator>
bool parse_complex(
    Iterator first
  , Iterator last
  , std::complex<double>& c)
```

Complex Number

```
double rN = 0.0;
double iN = 0.0;
bool r = phrase_parse(first, last,

    //  Begin grammar
    (
            '(' >> double_[ref(rN) = _1]
                >> -(',' >> double_[ref(iN) = _1]) >> ')'
        |   double_[ref(rN) = _1]
    )
    ,
    //  End grammar

    space);
```

Complex Number

```
double rN = 0.0;
double iN = 0.0;
bool r = phrase_parse(first, last,

    //  Begin grammar
    (
            '(' >> double_[ref(rN) = _1]
                >> -(',' >> double_[ref(iN) = _1]) >> ')'
        |   double_[ref(rN) = _1]
    )
    ,
    //  End grammar

    space);
```

Complex Number

```
double rN = 0.0;
double iN = 0.0;
bool r = phrase_parse(first, last,

    //  Begin grammar
    (
            '(' >> double_[ref(rN) = _1]
                >> -(',' >> double_[ref(iN) = _1]) >> ')'
        |   double_[ref(rN) = _1]
    )
    ,
    //  End grammar

    space);
```

Complex Number

```
double rN = 0.0;
double iN = 0.0;
bool r = phrase_parse(first, last,

    //  Begin grammar
    (
            '(' >> double_[ref(rN) = _1]
                >> -(',' >> double_[ref(iN) = _1]) >> ')'
        |   double_[ref(rN) = _1]
    )
    ,
    //  End grammar

    space);
```

Complex Number

```
double rN = 0.0;
double iN = 0.0;
bool r = phrase_parse(first, last,

    //  Begin grammar
    (
            '(' >> double_[ref(rN) = _1]
                >> -(',' >> double_[ref(iN) = _1]) >> ')'
        |   double_[ref(rN) = _1]
    )
    ,
    //  End grammar

    space);
```

Roman Numerals

Demonstrates the

- symbol table
- rules
- grammar

Roman Numerals

```
struct hundreds_ : symbols<char, unsigned>
{
    hundreds_()
    {
        add
            ("C"    , 100)
            ("CC"   , 200)
            ("CCC"  , 300)
            ("CD"   , 400)
            ("D"    , 500)
            ("DC"   , 600)
            ("DCC"  , 700)
            ("DCCC" , 800)
            ("CM"   , 900)
        ;
    }
} hundreds;
```

Roman Numerals

```
struct tens_ : symbols<char, unsigned>
{
    tens_()
    {
        add
            ("X"    , 10)
            ("XX"   , 20)
            ("XXX"  , 30)
            ("XL"   , 40)
            ("L"    , 50)
            ("LX"   , 60)
            ("LXX"  , 70)
            ("LXXX" , 80)
            ("XC"   , 90)
        ;
    }
} tens;
```

Roman Numerals

```
struct ones_ : symbols<char, unsigned>
{
    ones_()
    {
        add
            ("I"    , 1)
            ("II"   , 2)
            ("III"  , 3)
            ("IV"   , 4)
            ("V"    , 5)
            ("VI"   , 6)
            ("VII"  , 7)
            ("VIII" , 8)
            ("IX"   , 9)
        ;
    }
} ones;
```

Roman Numerals

```
template <typename Iterator>
struct roman : grammar_def<Iterator, unsigned()>
{
    roman()
    {
        start
            =   +char_('M') [_val += 1000]
            ||  hundreds    [_val += _1]
            ||  tens        [_val += _1]
            ||  ones        [_val += _1];
    }

    rule<Iterator, unsigned()> start;
};
```

Roman Numerals

```
template <typename Iterator>
struct roman : grammar_def<Iterator, unsigned()>
{
    roman()
    {
        start
            =   +char_('M') [_val += 1000]
            ||  hundreds    [_val += _1]
            ||  tens        [_val += _1]
            ||  ones        [_val += _1];
    }

    rule<Iterator, unsigned()> start;
};
```

Signature

Roman Numerals

```
template <typename Iterator>
struct roman : grammar_def<Iterator, unsigned()>
{
    roman()
    {
        start
            =   +char_('M') [_val += 1000]
            ||  hundreds    [_val += _1]
            ||  tens        [_val += _1]
            ||  ones        [_val += _1];
    }

    rule<Iterator, unsigned()> start;
};
```

val_: Synthesized-Attribute

(return type)

MiniXML

Some Basics:

```
text = lexeme[+(char_ - '<')    [_val += _1]];
node = (xml | text)             [_val = _1];
```

MiniXML

```
start_tag =
        '<'
    >>  lexeme[+(char_ - '>') [_val += _1]]
    >>  '>'
;

end_tag =
        "</"
    >>  lit(_r1)
    >>  '>'
;
```

MiniXML

```
start_tag =
        '<'
    >>  lexeme[+(char_ - '>')        [_val += _1]]
    >>  '>'
;

end_tag =
        "</"
    >>  lit(_r1)
    >>  '>'
;
```

_r1: Inherited-Attribute

(first function argument)

```
rule<Iterator, void(std::string), space_type> end_tag;
```

MiniXML

```
end_tag =
            "</"
      >>    lit(_r1)
      >>    '>'
;

xml =
            start_tag   [at_c<0>(_val) = _1]
      >>    *node       [push_back(at_c<1>(_val), _1)]
      >>    end_tag(at_c<0>(_val))
;
```

MiniXML

```
end_tag =
        "</"
    >>  lit(_r1)
    >>  '>'
;

xml =
        start_tag  [at_c<0>(_val) = _1]
    >>  *node      [push_back(at_c<1>(_val), _1)]
    >>  end_tag(at_c<0>(_val))
;
```

MiniXML

```
end_tag =
        "</"
    >>  lit(_r1)
    >>  '>'
;

xml =
        start_tag  [at_c<0>(_val) = _1]
    >>  *node      [push_back(at_c<1>(_val), _1)]
    >>  end_tag(at_c<0>(_val))
;
```

MiniXML

```
rule<Iterator, mini_xml(), space_type> xml;
```

MiniXML

```
struct mini_xml;

typedef
    boost::variant<
        boost::recursive_wrapper<mini_xml>
      , std::string
    >
mini_xml_node;

struct mini_xml
{
    std::string name;                          // tag name
    std::vector<mini_xml_node> children;    // children
};
```

MiniXML

```
// We need to tell fusion about our mini_xml struct
// to make it a first-class fusion citizen
BOOST_FUSION_ADAPT_STRUCT(
    mini_xml,
    (std::string, name)
    (std::vector<mini_xml_node>, children)
)
```

MiniXML Auto-AST generation

The basics, revisited:

```
text %= lexeme[+(char_ - '<')];
node %= xml | text;
```

MiniXML Auto-AST generation

The basics, new strategy:

```
text %= lexeme[+(char_ - '<')];
node %= xml | text;
```

MiniXML Auto-AST generation

```
start_tag %=
            '<'
      >>  lexeme[+(char_ - '>')]
      >>  '>'
;

end_tag =
            "</"
      >>  lit(_r1)
      >>  '>'
;

xml %=
            start_tag[_a = _1]
      >>  *node
      >>  end_tag(_a)
;
```

MiniXML Auto-AST generation

```
xml %=
        start_tag[_a = _1]
    >>  *node
    >>  end_tag(_a)
;


rule<
    Iterator
  , mini_xml()
  , locals<std::string>
  , space_type>
xml;
```

Local Variable

Calculator (Parser only)

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

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

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

Calculator (Parser only)

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

Calculator (Parser only)

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

Calculator (Parser only)

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

Calculator (Parser only)

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

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

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

Calculator With Actions

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

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

factor =
    uint_                               [bind(&do_int, _1)]
    |   '(' >> expression >> ')'
    |   ('-' >> factor                  [bind(&do_neg)])
    |   ('+' >> factor)
    ;
```

Calculator With Actions

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

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

Full Calculator

```
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])
    ;
```

Full Calculator

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

Full Calculator

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

Synthesized-Attribute

```
rule<Iterator, int(), space_type> expression
```

Error Handling

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

Error Handling

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

Error Handling

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

What failed?

Error Handling

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

iterators to error-position
and end of input

Error Handling

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

Fail parsing

Error Handling

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

Hard Expectation:

- **A deterministic point**
- **No backtracking**

Error Handling

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

Hard Expectation

Error Handling

```
123 * (456 + 789] / 20
```

Ooops!

Error! Expecting ')' here: "] / 20"

Error Handling

123 + blah

Ooops!

Error! Expecting term here: " blah"

Statements, Variables and Assignment… (calc6)

- A strategy for a grander scheme to come ;-)
- Demonstrates grammar modularization. Here you will see how expressions and statements are built as modular grammars.
- Breaks down the program into smaller, more manageable parts

Boolean expressions, compound statements and more... (calc7)

A little bit more, but still a calculator

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

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

Boolean expressions, compound statements and more... (calc7)

A little bit more, but still a calculator

```
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))
                     ]
    ;
```

Boolean expressions, compound statements and more... (calc7)

A little bit more, but still a calculator

```
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))
                     ]
    ;
```

Boolean expressions, compound statements and more… (calc7)

```
[
    _a = size(ref(code)) // mark our position
]
```

Local (temporary) variables

```
[
    _a = size(ref(code)) // mark our position
]
```

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

Mini-C: Not a calculator anymore, right? :-)

```
/* The factorial */

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

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

The Spirit Lexer components

The Spirit Lexer components

- Wrapper for lexer library (such as Ben Hansons lexertl, www.benhanson.net)
  - Exposes an iterator based interface for easy integration with Spirit parsers:

```
std::string str (…input…);
lexer<example1_tokens> lex(tokens);
grammar<example1_grammar> calc(def);
parse(lex.begin(str.begin(), str.end()), lex.end(), calc);
```

- Facilities allowing to define tokens based on regular expression strings
  - Classes: `token_def`, `token_set`, `lexer`

```
token_def<> identifier = "[a-zA-Z_][a-zA-Z0-9_]*";
self = identifier | ',' | '{' | '}';
```

- Lexer related components are at the same time parser components, allowing for tight integration

```
start = '{' >> *(tok.identifier >> -char_(',')) >> '}';
```

The Spirit Lexer Components

- Advantage:
  - Avoid re-scanning of input stream during backtracking
  - Simpler grammars for input language
  - Token values are evaluated once
  - Pattern (token) recognition is fast (uses regexs and DFAs)
- Disadvantages:
  - Additional overhead for token construction (especially if no backtracking occurs)

The Spirit Lexer Components

- Parsing using a lexer is fully token based (even single characters are tokens)
- Every token may have its own (typed) value

```
token_def<std::string> identifier =
    "[a-zA-Z_][a-zA-Z0-9_]*";
```

- During parsing this value is available as the tokens (parser components) ‘return value’

```
std::vector<std::string> names;
start = '{'
  >> *(tok.identifier >> -char_(',')) [ref(names)]
  >> '}';
```

- Token values are evaluated once and only on demand **à** no performance loss
- Tokens without a value ‘return’ iterator pair

Lexer Example 1: Token definition

- Separate class allows for encapsulated token definitions:

```
// template parameter 'Lexer' specifies the underlying lexer (library) to use
template <typename Lexer>
struct example1_tokens : lexer_def<Lexer>
{
    // the 'def()' function gets passed a reference to a lexer interface object
    template <typename Self>
    void def (Self& lex)
    {
        // define tokens and associate them with the lexer
        identifier = "[a-zA-Z_][a-zA-Z0-9_]*";
        lex = token_def<>(',') | '{' | '}' | identifier;

        // any token definition to be used as the skip parser during parsing
        // has to be associated with a separate lexer state (here 'WS')
        white_space = "[ \\t\\n]+";
        lex("WS") = white_space;
    }

    // every 'named' token has to be defined explicitly
    token_def<> identifier, white_space;
};
```

Lexer Example 1: Grammar definition

- Grammar definition takes token definition as parameter:

```
// template parameter 'Iterator' specifies the iterator this grammar is based on
// Note: token_def<> is used as the skip parser type
template <typename Iterator>
struct example1_grammar : grammar_def<Iterator, token_def<> >
{
    // parameter 'TokenDef' is a reference to the token definition class
    template <typename TokenDef>
    example1_grammar(TokenDef const& tok)
    {
        // Note: we use the 'identifier' token directly as a parser component
        start = '{' >> *(tok.identifier >> -char_(',')) >> '}';
    }

    // usual rule declarations, token_def<> is skip parser (as for grammar)
    rule<Iterator, token_def<> > start;
};
```

Lexer Example 1: Pulling it together

```
// iterator type used to expose the underlying input stream
    typedef std::string::const_iterator base_iterator_type;

// This is the lexer type to use to tokenize the input.
// We use the lexertl based lexer engine.
    typedef lexertl_lexer<base_iterator_type> lexer_type;

// This is the token definition type (derived from the given lexer type).
    typedef example1_tokens<lexer_type> example1_tokens;

// This is the iterator type exposed by the lexer
    typedef lexer<example1_tokens>::iterator_type iterator_type;

// This is the type of the grammar to parse
    typedef example1_grammar<iterator_type> example1_grammar;

// Now we use the types defined above to create the lexer and grammar
// object instances needed to invoke the parsing process
    example1_tokens tokens;                         // Our token definition
    example1_grammar def (tokens);                  // Our grammar definition

    lexer<example1_tokens> lex(tokens);             // Our lexer
    grammar<example1_grammar> calc(def);            // Our parser

// At this point we generate the iterator pair used to expose the tokenized input stream.
    std::string str (read_from_file("example1.input"));
    iterator_type iter = lex.begin(str.begin(), str.end());
    iterator_type end = lex.end();

// Parsing is done based on the the token stream, not the character stream read from the input.
// Note, how we use the token_def defined above as the skip parser.
    bool r = phrase_parse(iter, end, calc, tokens.white_space);
```

The Spirit Generator components

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
  - 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 resembling EBNF

Output generation

- DSEL was modeled after EBNF (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 values with formating rules:
  - Using literals (int_(10))
  - Semantic actions (int_[_1 = val(10)])
  - Explicit passing to API functions

The Direct Generator API

- Generating without delimiting

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

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

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

- Generating with delimiting

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

```
template <typename OutputIterator, typename Expr, typename Delimiter, typename
Parameter>
bool OutputIterator & first, Iterator last, Expr const& xpr,
   Parameter const& param, Delimiter const& delim);
```

```
int i = 42;
generate_delimited (sink, int_, i, space);   // outputs: “42 “
```

The Stream based Generator API

- Generating without delimiting

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

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

- Generating with delimiting

`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 = 42;
os << format_delimited(int_, i, space);    // outputs: “42 “
```

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[] (TBD)
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 …	Parameters: _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 and parameters	• Return type (attribute) is the type generated by the parser component, 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	• Parameter is the type expected by the generator component, i.e. the provided value must be convertible to this type. • Parameters are passed down. • Parameters are passed as const& • Generator components need always a ‚source‘ value: either literal or parameter

Generator Types and their Parameters

	Karma generator types	Parameter Type
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

Demo, Examples

Future Directions

- Qi
  - Deferred actions
  - Packrat Parsing (memoization)
  - LL1 deterministic parsing
  - Transduction Parsing (micro-spirit)
- Karma
  - More output formatting
  - Integration with layout engines
- Alchemy: parse tree transformation framework