

# Using Spirit 2.3 : Qi and Karma

## Get Possessed

Michael Caisse

Object Modeling Designs

[www.objectmodelingdesigns.com](http://www.objectmodelingdesigns.com)

Copyright © 2010

BoostCon, 2010

# Outline

- Part I: Tutorial
- Part II: Examples

# Outline of Part I

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Outline

- Part I: Tutorial
- Part II: Examples

# Outline of Part II

- 4 Protocol Translator
  - The Problem
  - The Solution
  
- 5 HTTP Request
  - The Request
  - The URI
  
- 6 XML
  - What is in a name?

# Part I

## Tutorial

Spirit is made up of:

- ***Qi*** - Parsing Library
- ***Karma*** - Generating Library
- ***Lex*** - A Lexer
- ***Classic*** - The Old Parsing Library

Spirit is made up of:

- ***Qi*** - Parsing Library
- ***Karma*** - Generating Library
- ***Lex*** - A Lexer
- ***Classic*** - The Old Parsing Library



# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

## Temptation to create ad-hoc solutions are high

- Your favorite *lex/flex/yacc/bison* are for big jobs
- I bet I can write a simple regex to parse this
- I don't want additional libraries to link
- A one-line `scanf/istream` will handle it
- A one-line `printf/ostream` will handle it
- `std::string` and `boost::lexical_cast` are my best friends

# Ad-hoc Parsing

```
std::string::const_iterator iter = argument.begin();
std::string::const_iterator iter_end = argument.end();
while( iter != iter_end )
{
    if( *iter == '+' )
    {
        if( building_key ){ key += ' '; }
        else { value += ' '; }
    }
    else if( *iter == '=' )
    {
        building_key = false;
    }
    else if( *iter == '&' )
    {
        argument_map[ key ] = value;
        key = "";
        value = "";
        building_key = true;
    }
    else if( *iter == '?' )
    {}
    else
    {
        if( building_key ){ key += *iter; }
        else { value += *iter; }
    }

    ++iter;
}
```

# Ad-hoc Parsing and Generating

```
boost::regex expression( "(request_firmware_version)|(calibrate_sensor_gain)|(calibrate_sensor_offset)" );
boost::smatch match;

if( boost::regex_search( product_data, match, expression ) )
{
    if( match[ 1 ].matched )
    {
        message_to_send += char( STX );
        message_to_send += char( 0x11 );
        message_to_send += char( ETX );
    }
    else if( match[ 2 ].matched )
    {
        message_to_send += char( STX );
        message_to_send += char( 0x12 );
        message_to_send += char( ETX );
    }
    else if( match[ 3 ].matched )
    {
        boost::regex expression( "calibrate_sensor (\\d+) (\\d+)" );
        if( boost::regex_search( product_data, match, expression ) )
        {
            try
            {
                message_to_send += char( STX );
                message_to_send += char( 0x13 );
                message_to_send += char( boost::lexical_cast<int>( match[ 1 ] ) + 0x10 );
                message_to_send += char( boost::lexical_cast<int>( match[ 2 ] ) + 0x10 );
                message_to_send += char( ETX );
            }
            catch( ... )
            {
                message_to_send.clear();
            }
        }
    }
}
```

# Ad-hoc Generating

```

std::vector< boost::any >::iterator product_iter = all_products.begin();
std::vector< boost::any >::iterator product_iter_end = all_products.end();
while( product_iter != product_iter_end )
{
    try
    {
        if( contains< GenericStorageType >( *product_iter ) )
        {
            // unsafe_any_cast is not part of the public interface. Our 'contains' method knows
            // have the right type. The standard any_cast uses type_info compares which won't work
            // library boundaries
            const GenericStorageType* p_product = boost::unsafe_any_cast< GenericStorageType >( *product_iter );

            send_packet += "<option value=\"";
            send_packet += p_product->first;
            send_packet += "\">";
            send_packet += p_product->first;
            send_packet += " - ";
            send_packet += p_product->second;
            send_packet += "</option>";

        }
    }
    catch( ... )
    {}

    ++product_iter;
}

```

# Outline

## 1 Motivation

- Ad-hoc Solutions
- **The Spirit Way**

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Loading a Table with Qi

## A 2 A.M. hack to load calibration data.

```
typedef std::vector< std::pair< int, int > > table_container_t;
struct cal_table_t
{
    int settle_time;
    table_container_t table;
};

bool load_calibration_table( const std::string& filename )
{
    std::ifstream stream( filename.c_str() );
    stream.unsetf( std::ios::skipws );
    spirit::istream_iterator begin_iter( stream );
    spirit::istream_iterator end_iter;

    return( qi::phrase_parse( begin_iter, end_iter,

                               qi::int_ >> *( qi::int_ >> qi::int_ )

                               , spirit::ascii::space
                               , cal_table ) );
}
```

# Is This Syntax Valid?

```
start =  
    lit( '/' )  
    >> -( +( ~char_( "/" ) ) )  
    >> -( '/' >> +( ~char_( "?" ) ) )  
    >> -( '?' >> ( query_pair % '&' ) )  
    ;  
  
query_pair =  
    +( ~char_( '=' ) )  
    >> '='  
    >> +( ~char_( '&' ) )  
    ;
```



# Wisdom

“Sometimes I think I’ll never really know C++”

*Eric Niebler*

# Power of Expression Templates

Spirit implements a ***Domain Specific Embedded Language*** for parsing and generating.

# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- **Parsers**
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Parser?

Data Stream  $\rightarrow$  Qi  $\rightarrow$  Abstract Syntax Tree (AST)

# A First, Simple Example

A parser for integers is simply:

Example (Integer Parser)

```
int_
```

A parser for doubles:

Example (Double Parser)

```
double_
```

A literal string parser:

Example (Parse literal string 'foo')

```
lit( "foo" )
```

# A First, Simple Example

We can use the parser with the `qi::parse` API.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ );
```

# A First, Simple Example

We can use the parser with the `qi::parse` API.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ );
```

# A First, Simple Example

We can use the parser with the `qi::parse` API.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
        int_ );
```



# A First, Simple Example

We can use the parser with the `qi::parse` API.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ );
```

# A First, Simple Example

We can use the parser with the `qi::parse` API.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ );
```

# A First, Simple Example

Parsing the double in just as simple.

```
std::string input( "1234.56" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       double_ );
```

# Some of the Available Parsers

Type	Parser	Example
signed	short_, int_, long_, long_long, int_(-42)	578, -1865, 99301
unsigned	bin, oct, hex, ushort_, ulong_, uint_, ulong_long, uint_(82)	01101, 24, 7af2, 243
real	float_, double_, long_double, double_(123.5)	-1.9023, 9328.11928
boolean	bool_, true_, false_	true, false
binary	byte_, word, dword, qword, word(0xface)	
big endian	big_word, big_dword, big_qword, big_dword(0xdeadbeef)	
litte endian	litte_word, litte_dword, litte_qword, little_dword(0xefbeadde)	

# Some of the Available Parsers

Type	Parser	Example
signed	short_, <b>int_</b> , long_, long_long, int_(-42)	578, -1865, 99301
unsigned	bin, oct, hex, ushort_, ulong_, <b>uint_</b> , ulong_long, uint_(82)	01101, 24, 7af2, 243
real	float_, <b>double_</b> , long_double, double_(123.5)	-1.9023, 9328.11928
boolean	<b>bool_</b> , true_, false_	true, false
binary	byte_, <b>word</b> , dword, qword, word(0xface)	
big endian	big_word, <b>big_dword</b> , big_qword, big_dword(0xdeadbeef)	
litte endian	litte_word, <b>litte_dword</b> , litte_qword, little_dword(0xefbeadde)	

# Some of the Available Parsers

Type	Parser	Example
signed	short_, int_, long_, long_long, <b>int_(-42)</b>	578, -1865, 99301
unsigned	bin, oct, hex, ushort_, ulong_, uint_, ulong_long, <b>uint_(82)</b>	01101, 24, 7af2, 243
real	float_, double_, long_double, <b>double_(123.5)</b>	-1.9023, 9328.11928
boolean	bool_, <b>true_</b> , false_	true, false
binary	byte_, word, dword, qword, <b>word(0xface)</b>	
big endian	big_word, big_dword, big_qword, <b>big_dword(0xdeadbeef)</b>	
little endian	litte_word, litte_dword, litte_qword, <b>little_dword(0xefbeadde)</b>	

# Some of the Available Parsers

Type	Parser	Example
character	<code>char_</code> , <code>char_('x')</code> , <code>char_(x)</code> , <code>char_('a','z')</code> , <code>char_("a-z8A-Z")</code> , <code>~char_('a')</code>	<code>a b e \$ 1 }</code>
	<code>lit('a')</code> , <code>'a'</code>	<code>a</code>
string	<code>string("foo")</code> , <code>string(s)</code> , <code>lit("bar")</code> , <code>"bar"</code> , <code>lit(s)</code>	
classification	<code>alnum</code> , <code>alpha</code> , <code>blank</code> , <code>cntrl</code> , <code>digit</code> , <code>graph</code> , <code>lower</code> , <code>print</code> , <code>punct</code> , <code>space</code> , <code>upper</code> , <code>xdigit</code>	

# Some of the Available Parsers

Type	Parser	Example
character	<b>char_</b> , char_('x'), char_(x), char_('a','z'), char_("a-z8A-Z"), ~char_('a')	a b e \$ 1 }
	lit('a'), 'a'	a
string	string("foo"), string(s), lit("bar"), "bar", lit(s)	
classification	alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit	



# Some of the Available Parsers

Type	Parser	Example
character	<code>char_</code> , <code>char_('x')</code> , <code>char_(x)</code> , <code>char_('a','z')</code> , <code>char_("a-z8A-Z")</code> , <code>~char_('a')</code>	<code>a b e \$ 1 }</code>
	<code>lit('a')</code> , <code>'a'</code>	<code>a</code>
string	<code>string("foo")</code> , <code>string(s)</code> , <code>lit("bar")</code> , <code>"bar"</code> , <code>lit(s)</code>	
classification	<code>alnum</code> , <code>alpha</code> , <code>blank</code> , <code>cntrl</code> , <code>digit</code> , <code>graph</code> , <code>lower</code> , <code>print</code> , <code>punct</code> , <code>space</code> , <code>upper</code> , <code>xdigit</code>	

# Some of the Available Parsers

Type	Parser	Example
character	<code>char_</code> , <code>char_('x')</code> , <code>char_(x)</code> , <b><code>char_('a','z')</code></b> , <code>char_("a-zA-Z")</code> , <code>~char_('a')</code>	<code>a b e \$ 1 }</code>
	<code>lit('a')</code> , <code>'a'</code>	<code>a</code>
string	<code>string("foo")</code> , <code>string(s)</code> , <code>lit("bar")</code> , <code>"bar"</code> , <code>lit(s)</code>	
classification	<code>alnum</code> , <code>alpha</code> , <code>blank</code> , <code>cntrl</code> , <code>digit</code> , <code>graph</code> , <code>lower</code> , <code>print</code> , <code>punct</code> , <code>space</code> , <code>upper</code> , <code>xdigit</code>	

# Some of the Available Parsers

Type	Parser	Example
character	<code>char_</code> , <code>char_('x')</code> , <code>char_(x)</code> , <code>char_('a','z')</code> , <b><code>char_("a-zA-Z")</code></b> , <code>~char_('a')</code>	<code>a b e \$ 1 }</code>
	<code>lit('a')</code> , <code>'a'</code>	<code>a</code>
string	<code>string("foo")</code> , <code>string(s)</code> , <code>lit("bar")</code> , <code>"bar"</code> , <code>lit(s)</code>	
classification	<code>alnum</code> , <code>alpha</code> , <code>blank</code> , <code>cntrl</code> , <code>digit</code> , <code>graph</code> , <code>lower</code> , <code>print</code> , <code>punct</code> , <code>space</code> , <code>upper</code> , <code>xdigit</code>	

# Some of the Available Parsers

Type	Parser	Example
character	<code>char_</code> , <code>char_('x')</code> , <code>char_(x)</code> , <code>char_('a','z')</code> , <code>char_("a-zA-Z")</code> , <b><code>~char_('a')</code></b>	<code>a b e \$ 1 }</code>
	<code>lit('a')</code> , <code>'a'</code>	<code>a</code>
string	<code>string("foo")</code> , <code>string(s)</code> , <code>lit("bar")</code> , <code>"bar"</code> , <code>lit(s)</code>	
classification	<code>alnum</code> , <code>alpha</code> , <code>blank</code> , <code>cntrl</code> , <code>digit</code> , <code>graph</code> , <code>lower</code> , <code>print</code> , <code>punct</code> , <code>space</code> , <code>upper</code> , <code>xdigit</code>	

# Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ >> ' ' >> double_ );
```

# Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ >> ' ' >> double_ );
```

# Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ >> ' ' >> double_ );
```

# Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ >> ' ' >> double_ );
```



# Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
parse( iter, end_iter,  
       int_ >> ' ' >> double_ );
```

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

# Operators

Description	PEG	Spirit Qi
<b>Sequence</b>	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

Read as *a* is followed by *b*

```
int_ >> ' ' >> double_  
"42 -89.3"
```

```
char_ >> ':' >> int_  
"a:19"
```

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
<b>Alternative</b>	<code>a   b</code>	<b><code>a   b</code></b>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

Either *a* **or** *b* are allowed.  
Evaluated in listed order.

**alpha | digit | punct**  
`"a"`  
`"9"`  
`";"`  
`"+"` *fails to parse*

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
<b>Zero of more (Kleene)</b>	<code>a*</code>	<b><code>*a</code></b>
<b>One or more (Plus)</b>	<code>a+</code>	<b><code>+a</code></b>
<b>Optional</b>	<code>a?</code>	<b><code>-a</code></b>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

```
*alpha >> int_  
"z86"  
"abcde99"  
"99"
```

```
+alpha >> int_  
"z86"  
"abcde99"  
"99" parse fails
```

```
-alpha >> int_  
"z86"  
"abcde99" parse fails  
"99"
```

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
<b>And-predicate</b>	<code>&amp;a</code>	<b><code>&amp;a</code></b>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

And-predicate can provide basic look-ahead. It matches *a* without consuming *a*.

```
int_ >> &char_(' ; ')
```

```
"86;"
```

*"-99" fails to parse*

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
<b>Not-predicate</b>	<code>!a</code>	<b><code>!a</code></b>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

Not-predicate can provide basic look-ahead. If *a* does match the parse is successful without consuming *a*.

```
"for" >> !(alnum|' _')
```

```
"for()"
```

*"forty" fails to parse*

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
<b>Difference</b>		<b><code>a - b</code></b>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

Match *a* but not *b*.

```
"/*"
>> *(char_ - "*/")
>> "*/"
```

```
"/* comment */"
```

Always fails.

```
lit("obiwatanabe") -
"obiwa"
```



# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
<b>Expectation</b>		<b><code>a &gt; b</code></b>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

*a* must be followed by *b*. No backtracking allowed. A Sequence returns no-match, an Expectation throws `expectation_failure<iter>`

```
char_('o')
> char_('k')
```

"ok"

"ox" *throws exception*

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
<b>List</b>		<b><code>a % b</code></b>
Permutation		<code>a ^ b</code>
Sequential Or		<code>a    b</code>

Shortcut for:

```
a >> *( b >> a )
```

```
int_ % ', '
```

```
"9,2,42,-187,76"
```

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero of more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
<b>Permutation</b>		<b><code>a ^ b</code></b>
Sequential Or		<code>a    b</code>

Parse *a*, *b*, *c*, ... in any order.  
Each element can occur 0:1 times.

```
char_('a') ^ 'b' ^ 'c'
"abc"
"ba"
"cba"
"bb" second b fails
```

```
*( char_('A') ^ 'C'
    ^ 'T' ^ 'G' )
"ACTGGCTAGACT"
```

# Operators

Description	PEG	Spirit Qi
Sequence	<code>a b</code>	<code>a &gt;&gt; b</code>
Alternative	<code>a   b</code>	<code>a   b</code>
Zero or more (Kleene)	<code>a*</code>	<code>*a</code>
One or more (Plus)	<code>a+</code>	<code>+a</code>
Optional	<code>a?</code>	<code>-a</code>
And-predicate	<code>&amp;a</code>	<code>&amp;a</code>
Not-predicate	<code>!a</code>	<code>!a</code>
Difference		<code>a - b</code>
Expectation		<code>a &gt; b</code>
List		<code>a % b</code>
Permutation		<code>a ^ b</code>
<b>Sequential Or</b>		<b><code>a    b</code></b>

Shortcut for:

`a >> -b | b`

Mind your order!

`int_ || ('.' >> int_)`

`"123.456"`

`".456"`

`"123"`

# Combining Parsers - Parse key/value pairs

```

1  std::string input( "foo  : bar "
2                        "gorp : smart "
3                        "falcou : \"crazy frenchman\" "
4                        "arm8 : risc " );
5
6  std::string::iterator iter = input.begin();
7  std::string::iterator iter_end = input.end();
8
9  phrase_parse( iter, iter_end,
10               // ----- start parser -----
11               *( (alpha >> *alnum)
12                 >> ':'
13                 >> ( '"' >> *( ~char_( '"' ) ) >> '"' )
14                   |
15                   (alpha >> *alnum)
16                 )
17               // ----- end parser -----
18               , space );

```

# Combining Parsers - Parse key/value pairs refined

```
1  std::string input( "foo  : bar "  
2                        "gorp : smart "  
3                        "falcou : \"crazy frenchman\" "  
4                        "arm8 : risc " );  
5  
6  std::string::iterator iter = input.begin();  
7  std::string::iterator iter_end = input.end();  
8  
9  phrase_parse( iter, iter_end,  
10              // ----- start parser -----  
11  
12  
13              *( name >> ':' >> ( quote | name ) )  
14  
15  
16  
17              // ----- end parser -----  
18              , space );
```

# Combining Parsers - Rules

Rules allow us to organize parsers into named units. They provide a few facilities:

- Allows us to name parsers
- Specify synthesized attribute type
- Specify inherited attribute types
- Specify local variables

# Combining Parsers - Rules

Assign our parsers to rules.

```
qi::rule<iter_t, space_type> name;  
name = alpha >> *alnum;
```

```
qi::rule<iter_t, space_type> quote;  
quote = '\"' >> *( ~char_('"') ) >> '\"';
```



# Combining Parsers - Rules

Assign our parsers to rules.

```
qi::rule<iter_t, space_type> name;  
name = alpha >> *alnum;
```

```
qi::rule<iter_t, space_type> quote;  
quote = '\"' >> *( ~char_('"') ) >> '\"';
```

# Combining Parsers - Rules

The iterator type to be used by the rule.

```
qi::rule<iter_t, space_type> name;  
name = alpha >> *alnum;
```

```
qi::rule<iter_t, space_type> quote;  
quote = '\"' >> *( ~char_('"') ) >> '\"';
```

# Combining Parsers - Rules

The skipper type to be used by the rule.

```
qi::rule<iter_t, space_type> name;  
name = alpha >> *alnum;
```

```
qi::rule<iter_t, space_type> quote;  
quote = '\"' >> *( ~char_('"') ) >> '\"';
```

# Combining Parsers - Parse key/value pairs refined

```
1 std::string input( "foo : bar "  
2                   "gorp : smart "  
3                   "falcou : \"crazy frenchman\" "  
4                   "arm8 : risc " );  
5  
6 typedef std::string::iterator iter_t;  
7 iter_t iter = input.begin();  
8 iter_t iter_end = input.end();  
9  
10 rule<iter_t, space_type> name  = alpha >> *alnum;  
11 rule<iter_t, space_type> quote =      ' "  
12                                     >> * (~char_( ' " ' ) )  
13                                     >>      ' " ' ;  
14  
15 phrase_parse( iter, iter_end,  
16              // ----- start parser -----  
17              * ( name >> ' :' >> ( quote | name ) )  
18              // ----- end parser -----  
19              , space );
```

# Combining Parsers - Grammars

Grammars:

- Group rules into higher level abstractions
- Expose an attribute
- Are just structures
- Specify local variables

# Combining Parsers - Grammars

```
1 struct key_value_list : grammar
2 {
3     key_value_list()
4     {
5         // rule assignment here
6     };
7
8     // rule definitions here
9     rule start;
10    rule item;
11    rule key;
12    rule value;
13 };
```

# Combining Parsers - Grammars

```
1 template <typename Iter>
2 struct key_value_list : grammar<Iter, space_type>
3 {
4     key_value_list() : key_value_list::base_type(start)
5     {
6         start = *item;
7         item = key >> ':' >> value;
8         key = alpha >> *alnum;
9         value = ('"' >> * (~char_('"')) >> '"')
10                |
11                *alnum;
12     };
13
14     rule<Iter, space_type> start;
15     rule<Iter, space_type> item;
16     rule<Iter, space_type> key;
17     rule<Iter, space_type> value;
18 };
```

# Combining Parsers - Parse key/value with grammar

```
1 std::string input( "foo  : bar "  
2                  "gorp : smart "  
3                  "falcou : \"crazy frenchman\" "  
4                  "arm8 : risc " );  
5  
6 typedef std::string::iterator iter_t;  
7 iter_t iter = input.begin();  
8 iter_t iter_end = input.end();  
9  
10 key_value_list<iter_t> list_grammar;  
11  
12 phrase_parse( iter, iter_end,  
13             list_grammar,  
14             space );
```



# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- **Attributes and Actions**
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Getting Parse Results

How do we get at the parsed results?

```
1 std::string input( "foo  : bar "  
2                   "corp : smart "  
3                   "falcou : \"crazy frenchman\" "  
4                   "arm8 : risc " );  
5  
6 std::map< std::string, std::string > key_value_map;  
7  
8 // Do something clever here ??????????
```

# Parsers Expose Attributes - Synthesized Attributes

	Qi Parser Type	Attribute Type
Literals	'a', "abc", int_(42), ...	No attribute
Primitives	int_, char_, double_, ...	int, char, double, ...
	bin, oct, hex	unsigned
	byte_, word, dword, ...	uint8_t, uint16_t, uint32_t, ...
	string("abc")	"abc"
	symbol<A, B>	specified (B)
Non-terminal	rule<A()>, grammar<A()>	specified (A)
Operators	a >> b (sequence)	fusion::vector<A, B>
	a   b (alternative)	boost::variant<A,B>
	*a (zero or more)	std::vector<A>
	+a (one or more)	std::vector<A>
	-a (optional, zero or one)	boost::optional<A>
	&a, !a (predicates)	No attribute
	a % b (list)	std::vector<A>
	a ^ b (permutation)	fusion::vector<optional<A>, optional<B> >

# A First Attribute Example

We can simply provide a reference to the parse API and get the ***Synthesized Attribute***.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
int result;  
parse( iter, end_iter,  
       int_,  
       result );
```

# A First Attribute Example

We can simply provide a reference to the parse API and get the ***Synthesized Attribute***.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
int result;  
parse( iter, end_iter,  
      int_,  
      result );
```

# A First Attribute Example

We can simply provide a reference to the parse API and get the ***Synthesized Attribute***.

```
std::string input( "1234" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
int result;  
parse( iter, end_iter,  
       int_,  
       result );
```

# Parse a string into a `std::string`

Attribute parsing can produce *compatible attributes*

```
std::string input( "pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::string result;  
parse( iter, end_iter,  
      *char_,  
      result );
```

`std::string` is compatible with `std::vector<char>`  
attribute of the `*char_` parser.

# Parse a string into a `std::string`

Attribute parsing can produce *compatible attributes*

```
std::string input( "pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::string result;  
parse( iter, end_iter,  
      *char_,  
      result );
```

`std::string` is compatible with `std::vector<char>`  
attribute of the `*char_` parser.



# Parse a string into a `std::string`

Attribute parsing can produce *compatible attributes*

```
std::string input( "pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::string result;  
parse( iter, end_iter,  
       *char_,  
       result );
```

`std::string` is compatible with `std::vector<char>`  
attribute of the `*char_` parser.

# Attribute Parsing - Sequence Parse API

```
std::string input( "cosmic pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::string result1;  
std::string result2;  
  
parse( iter, end_iter,  
      ~(~char_(' ')) >> ' ' >> *char_,  
      result1,  
      result2 );
```

# Attribute Parsing - Sequence Parse API

```
std::string input( "cosmic pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::string result1;  
std::string result2;  
  
parse( iter, end_iter,  
      *(&char_(' ')) >> ' ' >> &char_,  
      result1,  
      result2 );
```

# Attribute Parsing - Sequence Parse API

```
std::string input( "cosmic pizza" );
std::string::iterator iter = input.begin();
std::string::iterator end_iter = input.end();
std::string result1;
std::string result2;

parse( iter, end_iter,
      *(~char_(' ')) >> ' ' >> *char_,
      result1,
      result2 );
```

# Attribute Parsing - Sequence Parse API

## Compatible attributes to the rescue!

```
std::string input( "cosmic pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::pair<std::string, std::string> result;  
  
parse( iter, end_iter,  
      *(~char_(' ')) >> ' ' >> *char_,  
      result );
```

# Attribute Parsing - Sequence Parse API

Compatible attributes to the rescue!

```
std::string input( "cosmic pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::pair<std::string, std::string> result;  
  
parse( iter, end_iter,  
      *(&char_(' ')) >> ' ' >> &char_,  
      result );
```

# Attribute Parsing - Sequence Parse API

Compatible attributes to the rescue!

```
std::string input( "cosmic pizza" );  
std::string::iterator iter = input.begin();  
std::string::iterator end_iter = input.end();  
std::pair<std::string, std::string> result;  
  
parse( iter, end_iter,  
      *(~char_(' ')) >> ' ' >> *char_,  
      result );
```

# Attribute Parsing - Compatibility

Attribute parsing is where the Spirit *Magic* lives.

```
std::string input( "foo    : bar ,"
                  "gorp    : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ', ',
              space,
              key_value_map );
```



# Attribute Parsing - Compatibility

Synthesized attributes are formulated as C++ function types.

```
std::string input( "foo    : bar ,"
                  "gorp    : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '"'
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ',',
              space,
              key_value_map );
```

# Attribute Parsing - Compatibility

**a: char**, b: std::vector<char>  $\rightarrow$  ( a >> b ): std::vector<char>

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '"'
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ', ',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

a: char, **b: std::vector<char>**  $\rightarrow$  ( a >> b ): std::vector<char>

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '"'
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ', ',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

a: char, b: std::vector<char>  $\rightarrow$  ( a >> b ): **std::vector<char>**

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '"'
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ',',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

**a: unused**, b: vector<char>, **c: unused** → ( a >> b >> c): std::vector<char>

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =      ' "'
>> lexeme[ *(~char_('\"')) ]
>> ' "';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ', ',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

a: unused, **b: vector<char>**, c: unused  $\rightarrow$  ( a >> b >> c): std::vector<char>

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =      ' "'
>> lexeme[ * (~char_(' ')) ]
>> ' ';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ', ',
              space,
              key_value_map );
```

# Attribute Parsing - Compatibility

a: unused, b: vector<char>, c: unused  $\rightarrow$  ( a >> b >> c): **std::vector<char>**

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =      "'"
    >> lexeme[ *(~char_('"')) ]
    >> "'";

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ', ',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

**a: string**, b: string  $\rightarrow$  ( a | b ): variant<string, string>  $\rightarrow$  string

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '""'
    >> lexeme[ *(~char_('"')) ]
    >> '""';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ', ',
              space,
              key_value_map );
```



# Attribute Parsing - Compatibility

a: string, **b: string**  $\rightarrow$  ( a | b ): variant<string, string>  $\rightarrow$  string

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '""'
    >> lexeme[ *(~char_('"')) ]
    >> '""';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ',',
              space,
              key_value_map );
```

# Attribute Parsing - Compatibility

$a: \text{string}, b: \text{string} \rightarrow (a \mid b): \text{variant}\langle \text{string}, \text{string} \rangle \rightarrow \text{string}$

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '"'
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ',',
              space,
              key_value_map );
```

# Attribute Parsing - Compatibility

**a: string**, b: unused, **c: string**  $\rightarrow$  ( a >> b >> c): tuple<string, string>

```
std::string input( "foo      : bar ,",
                  "gorp      : smart ,",
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ',',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

a: string, **b: unused**, c: string  $\rightarrow$  ( a >> b >> c): tuple<string, string>

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ',',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

a: string, b: unused, c: string  $\rightarrow$  ( a >> b >> c): **tuple<string, string>**

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    >> lexeme[ *(~char_('"')) ]
    >> '"';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
             item % ',',
             space,
             key_value_map );
```

# Attribute Parsing - Compatibility

**a: std::pair<string, string> → ( a % b ): vector< std::pair<string, string> >**

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '""'
    >> lexeme[ *(~char_('\"')) ]
    >> '""';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ', ',
              space,
              key_value_map );
```

# Attribute Parsing - Compatibility

a: std::pair<string, string>  $\rightarrow$  ( a % b ): **vector< std::pair<string, string> >**

```
std::string input( "foo      : bar ,"
                  "gorp      : smart ,"
                  "falcou : \"crazy frenchman\" " );

typedef std::string::iterator iter_t;
iter_t iter = input.begin();
iter_t iter_end = input.end();

rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
rule<iter_t, std::string(), space_type> quote =
    '""'
    >> lexeme[ *(~char_('\"')) ]
    >> '""';

rule<iter_t, std::pair<std::string, std::string>(), space_type>
    item = name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;

phrase_parse( iter, iter_end,
              item % ', ',
              space,
              key_value_map );
```

# Complete Example

```

1  int main()
2  {
3      std::string input( "foo    : bar ,",
4                          "gorp   : smart ,",
5                          "falcou : \"crazy frenchman\" " );
6      iter_t iter = input.begin();
7      iter_t iter_end = input.end();
8
9      qi::rule<iter_t, std::string(), space_type> name = alpha >> *alnum;
10
11     qi::rule<iter_t, std::string(), space_type>
12         quote = "'" >> qi::lexeme[ ~(char_('"')) ] >> "'";
13
14     qi::rule<iter_t, std::pair<std::string, std::string>(), space_type>
15         item =
16             name
17             >> ':'
18             >> ( quote | name );
19
20     std::map< std::string, std::string > key_value_map;
21
22     qi::phrase_parse( iter, iter_end,
23                       item % ',',
24                       qi::ascii::space,
25                       key_value_map );
26
27     std::for_each( key_value_map.begin(), key_value_map.end(),
28                   std::cout << phx::at_c<0>(arg1) << " : "
29                   << phx::at_c<1>(arg1) << '\n' );
30
31     return 0;
32 }

```

## Output

```

falcou : crazy frenchman
foo : bar
gorp : smart

```



# Attribute Parsing : Fusion Adaption

```
1 struct boost_author{
2     boost::optional<std::string> name;
3     boost::optional<std::string> email;
4     boost::optional<std::vector< std::string > > libraries; };
5
6 BOOST_FUSION_ADAPT_STRUCT(
7     boost_author,
8     (boost::optional<std::string>, name)
9     (boost::optional<std::vector<std::string> >, libraries)
10 )
11
12 int main()
13 {
14     typedef std::string::iterator iter_t;
15     std::vector< boost_author > result;
16
17     std::string input( "{(name: Hartmut Kaiser)(libraries: spirit,wave)}"
18                       "{(libraries: spirit,phoenix,fusion,phoenix)(name: Joel de Guzman)}"
19                       "{(libraries: units)(name: Steven Watanabe)}" );
20     iter_t iter = input.begin();
21     iter_t end = input.end();
22
23     qi::rule<iter_t, std::string>, space_type>
24         name = lit('(') >> "name" >> ':' >> lexeme[ *(~char_('(')) ] >> ')';
25
26     qi::rule<iter_t, std::vector<std::string>(), space_type>
27         libraries = lit('(') >> "libraries" >> ':' >> (*(~char_(", "))) % ', ' >> ')';
28
29     qi::phrase_parse( iter, end,
30         *(' ' >> (name ^ libraries) >> ')'),
31         space,
32         result );
33
34     return 0;
35 }
```

# Attribute Parsing : Top - Down

```

1  int main()
2  {
3      typedef std::string::iterator iter_t;
4
5      std::string input( "foo bar: kaaal gorp$" );
6      iter_t iter = input.begin();
7      iter_t end = input.end();
8
9      qi::rule<iter_t, std::vector<char>>()> next_rule = *(~char_('$'));
10     qi::rule<iter_t, std::vector<char>>()> top_rule  = *(~char_(':')) >> ':' >> next_rule;
11
12     std::vector< char > result;
13
14     qi::parse( iter, end,
15               top_rule,
16               result );
17
18     std::cout << karma::format( karma::string, result );
19
20     return 1;
21 }

```

## Output

foo bar kaaal gorp

# Semantic Actions

When more control is required... Semantic Actions

- Can be attached to any non-terminal in the grammar
- Executes after a successful parse
- Provides access to:
  - Synthesized attribute value
  - Inherited attribute values
  - Local variables
  - Ability to force parser to fail

# Semantic Actions - Example 1

The rule synthesizes one int but parses two ints

```
std::string input( "12 * 8" );
```

```
rule<iter_t, int() , space_type> mult =
```

```
    int_      [ _val = _1 ]  
>> ' * '  
>> int_      [ _val *= _1 ]  
;
```

```
int result = 100;
```

```
phrase_parse( iter, end,  
             mult,  
             space,  
             result );
```

# Semantic Actions - Example 1

Actions are attached to non-terminals with **[ ]**

```
std::string input( "12 * 8" );

rule<iter_t, int(), space_type> mult =

    int_      [ _val = _1 ]
  >> ' * '
  >> int_      [ _val *= _1 ]
  ;

int result = 100;

phrase_parse( iter, end,
             mult,
             space,
             result );
```

# Semantic Actions - Example 1

Phoenix placeholder for rule's Synthesized Attribute

```
std::string input( "12 * 8" );

rule<iter_t, int(), space_type> mult =
    int_      [ _val = _1 ]
  >> ' * '
  >> int_      [ _val *= _1 ]
  ;

int result = 100;

phrase_parse( iter, end,
             mult,
             space,
             result );
```

# Semantic Actions - Example 1

Phoenix placeholder for attached parser's attribute

```
std::string input( "12 * 8" );

rule<iter_t, int(), space_type> mult =

    int_      [ _val = 1 ]
>>  ' * '
>>  int_      [ _val *= 1 ]
;

int result = 100;

phrase_parse( iter, end,
             mult,
             space,
             result );
```

# Semantic Actions - Example 1

After semantic action `result` is 12

```
std::string input( "12 * 8" );
```

```
rule<iter_t, int(), space_type> mult =
```

```
    int_      [ _val = _1 ]
>> ' * '
>> int_      [ _val *= _1 ]
;
```

```
int result = 100;
```

```
phrase_parse( iter, end,
              mult,
              space,
              result );
```



# Semantic Actions - Example 1

After semantic action `result` is 96

```
std::string input( "12 * 8" );
```

```
rule<iter_t, int(), space_type> mult =
```

```
    int_      [ _val = _1 ]  
>> ' * '  
>> int_      [ _val *= _1 ]  
    ;
```

```
int result = 100;
```

```
phrase_parse( iter, end,  
             mult,  
             space,  
             result );
```

# Semantic Actions - Example 1

What if fails after parsing the first int: `result = 12`

```
std::string input( "12 + 8" );
```

```
rule<iter_t, int(), space_type> mult =
```

```
    int_      [ _val = _1 ]
>> ' * '
>> int_      [ _val *= _1 ]
;
```

```
int result = 100;
```

```
phrase_parse( iter, end,
              mult,
              space,
              result );
```

# Semantic Actions - Example 2

We can introduce phoenix local variables to a rule

```
std::string input( "12 * 8" );
```

```
rule<iter_t, int(), locals<int>, space_type> mult =
```

```

    int_      [ _a = _1 ]
>>  ' * '
>>  int_      [ _val = _a * _1 ]
;
```

```
int result = 100;
```

```
phrase_parse( iter, end,
              mult,
              space,
              result );
```

# Semantic Actions - Example 2

After semantic action `result` is 100

```
std::string input( "12 * 8" );
```

```
rule<iter_t, int(), locals<int>, space_type> mult =
```

```

    int_      [ _a = _1 ]
>>  ' * '
>>  int_      [ _val = _a * _1 ]
;
```

```
int result = 100;
```

```
phrase_parse( iter, end,
              mult,
              space,
              result );
```

# Semantic Actions - Example 2

After semantic action `result` is 96

```
std::string input( "12 * 8" );
```

```
rule<iter_t, int(), locals<int>, space_type> mult =
```

```

    int_      [ _a = _1 ]
>>  ' * '
>>  int_      [ _val = _a * _1 ]
    ;

```

```
int result = 100;
```

```
phrase_parse( iter, end,
              mult,
              space,
              result );
```

# Semantic Actions - Inherited Attributes -Example 3

**Inherited Attributes:** the arguments in the C++ function type

```
int result = 100;
std::string input( "12 - 8" );

rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;

mult =  '*' >> int_    [ _val = _r1 * _1 ] ;
div  =  '/' >> int_    [ _val = _r1 / _1 ] ;
add  =  '+' >> int_    [ _val = _r1 + _1 ] ;
sub  =  '-' >> int_    [ _val = _r1 - _1 ] ;

binary_op =          int_                [ _a = _1 ]
              >> (    add(_a)
                    | sub(_a)
                    | mult(_a)
                    | div(_a) )        [ _val = _1 ];

phrase_parse( iter, end,
              binary_op,
              space, result );
```

# Semantic Actions - Inherited Attributes -Example 3

**Synthesized Attribute:** return part of the C++ function type

```
int result = 100;
std::string input( "12 - 8" );

rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;

mult =  '*' >> int_    [ _val = _r1 * _1 ] ;
div  =  '/' >> int_    [ _val = _r1 / _1 ] ;
add  =  '+' >> int_    [ _val = _r1 + _1 ] ;
sub  =  '-' >> int_    [ _val = _r1 - _1 ] ;

binary_op =      int_                [ _a = _1 ]
              >> (    add(_a)
                    | sub(_a)
                    | mult(_a)
                    | div(_a) )      [ _val = _1 ] ;

phrase_parse( iter, end,
              binary_op,
              space, result );
```

# Semantic Actions - Inherited Attributes -Example 3

Passed as if an argument to the rule.

```
int result = 100;
std::string input( "12 - 8" );

rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;

mult =  '*' >> int_    [ _val = _r1 * _1 ] ;
div  =  '/' >> int_    [ _val = _r1 / _1 ] ;
add  =  '+' >> int_    [ _val = _r1 + _1 ] ;
sub  =  '-' >> int_    [ _val = _r1 - _1 ] ;

binary_op =      int_                [ _a = _1 ]
              >> (    add(_a)
                    | sub(_a)
                    | mult(_a)
                    | div(_a) )      [ _val = _1 ] ;

phrase_parse( iter, end,
              binary_op,
              space, result );
```



# Semantic Actions - Inherited Attributes -Example 3

Use Phoenix placeholder `_rN` to access inherited attribute

```
int result = 100;
std::string input( "12 - 8" );

rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;

mult =  '*' >> int_    [ _val = r1 * _1 ] ;
div  =  '/' >> int_    [ _val = r1 / _1 ] ;
add  =  '+' >> int_    [ _val = r1 + _1 ] ;
sub  =  '-' >> int_    [ _val = r1 - _1 ] ;

binary_op =          int_                [ _a = _1 ]
            >> (      add(_a)
                  | sub(_a)
                  | mult(_a)
                  | div(_a) )    [ _val = _1 ];

phrase_parse( iter, end,
              binary_op,
              space, result );
```

# Semantic Actions - Inherited Attributes -Example 3

Let's parse....

```
int result = 100;
std::string input( "12 - 8" );
```

```
rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;
```

```
mult = '*' >> int_ [ _val = _r1 * _1 ] ;
div  = '/' >> int_ [ _val = _r1 / _1 ] ;
add  = '+' >> int_ [ _val = _r1 + _1 ] ;
sub  = '-' >> int_ [ _val = _r1 - _1 ] ;
```

```
binary_op =      int_      [ _a = _1 ]
                >> (      add(_a)
                      | sub(_a)
                      | mult(_a)
                      | div(_a) ) [ _val = _1 ];
```

```
phrase_parse( iter, end,
              binary_op,
              space, result );
```

# Semantic Actions - Inherited Attributes -Example 3

Let's parse....

```
int result = 100;
std::string input( "12 - 8" );
```

```
rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;
```

```
mult =  '*' >> int_    [ _val = _r1 * _1 ] ;
div  =  '/' >> int_    [ _val = _r1 / _1 ] ;
add  =  '+' >> int_    [ _val = _r1 + _1 ] ;
sub  =  '-' >> int_    [ _val = _r1 - _1 ] ;
```

```
binary_op =          int_                [ _a = _1 ]
              >> (    add(_a)
                    |  sub(_a)
                    | mult(_a)
                    | div(_a) )    [ _val = _1 ];
```

```
phrase_parse( iter, end,
              binary_op,
              space, result );
```

# Semantic Actions - Inherited Attributes -Example 3

Let's parse....

```
int result = 100;
std::string input( "12 - 8" );
```

```
rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;
```

```
mult = '*' >> int_ [ _val = _r1 * _1 ] ;
div  = '/' >> int_ [ _val = _r1 / _1 ] ;
add  = '+' >> int_ [ _val = _r1 + _1 ] ;
sub  = '-' >> int_ [ _val = _r1 - _1 ] ;
```

```
binary_op =          int_          [ _a = _1 ]
            >> (      add(_a)
                  | sub(_a)
                  | mult(_a)
                  | div(_a) )      [ _val = _1 ];
```

```
phrase_parse( iter, end,
              binary_op,
              space, result );
```

# Semantic Actions - Inherited Attributes -Example 3

The value of `result` is 4 after semantic action

```
int result = 100;
std::string input( "12 - 8" );

rule<iter_t, int(int), space_type> mult, div, add, sub;
rule<iter_t, int(), locals<int>, space_type> binary_op;

mult =  '*' >> int_    [ _val = _r1 * _1 ] ;
div  =  '/' >> int_    [ _val = _r1 / _1 ] ;
add  =  '+' >> int_    [ _val = _r1 + _1 ] ;
sub  =  '-' >> int_    [ _val = _r1 - _1 ] ;

binary_op =          int_                [ _a = _1 ]
            >> (      add(_a)
                  | sub(_a)
                  | mult(_a)
                  | div(_a) )    [ _val = _1 ];

phrase_parse( iter, end,
              binary_op,
              space, result );
```

# Semantic Actions - Auto Rule - Example 4

Parse string of characters and print the string and count

```
int count = 0;
std::string result;
std::string input( "I love Boost.Phoenix" );
iter_t iter = input.begin();
iter_t end = input.end();

qi::rule<iter_t, std::string>
    count_rule = *( char_[ ++ref(count) ] ) ;

qi::parse( iter, end,
          count_rule,
          result );

std::cout >> result >> " has "
          >> count >> " characters" >> std::endl;
```

# Semantic Actions - Auto Rule - Example 4

Parse string of characters and print the string and count

```
int count = 0;
std::string result;
std::string input( "I love Boost.Phoenix" );
iter_t iter = input.begin();
iter_t end = input.end();

qi::rule<iter_t, std::string()>
    count_rule = *( char_[ ++ref(count) ] ) ;

qi::parse( iter, end,
           count_rule,
           result );

std::cout >> result >> " has "
           >> count >> " characters" >> std::endl;
```

# Semantic Actions - Auto Rule - Example 4

Parse string of characters and print the string and count

```
int count = 0;
std::string result;
std::string input( "I love Boost.Phoenix" );
iter_t iter = input.begin();
iter_t end = input.end();

qi::rule<iter_t, std::string>
    count_rule = *( char_[ ++ref(count) ] ) ;

qi::parse( iter, end,
          count_rule,
          result );

std::cout >> result >> " has "
          >> count >> " characters" >> std::endl;
```



# Semantic Actions - Auto Rule - Example 4

Parse string of characters and print the string and count

```
int count = 0;
std::string result;
std::string input( "I love Boost.Phoenix" );
iter_t iter = input.begin();
iter_t end = input.end();

qi::rule<iter_t, std::string()>
    count_rule = *( char_[ ++ref(count) ] ) ;

qi::parse( iter, end,
    count_rule,
    result );

std::cout >> result >> " has "
    >> count >> " characters" >> std::endl;
```

# Semantic Actions - Auto Rule - Example 4

Output: “ has 20 characters”... *Gack!*

```
int count = 0;
std::string result;
std::string input( "I love Boost.Phoenix" );
iter_t iter = input.begin();
iter_t end = input.end();

qi::rule<iter_t, std::string()>
    count_rule = *( char_[ ++ref(count) ] ) ;

qi::parse( iter, end,
    count_rule,
    result );

std::cout >> result >> " has "
    >> count >> " characters" >> std::endl;
```

# Semantic Actions - Auto Rule - Example 4

Rules with semantic actions require explicit **auto rule**

```
int count = 0;
std::string result;
std::string input( "I love Boost.Phoenix" );
iter_t iter = input.begin();
iter_t end = input.end();

qi::rule<iter_t, std::string()>
    count_rule %= *( char_[ ++ref(count) ] ) ;

qi::parse( iter, end,
           count_rule,
           result );

std::cout >> result >> " has "
           >> count >> " characters" >> std::endl;
```

# Semantic Actions - Auto Rule - Example 4

Output: "I love Boost.Phoenix has 20 characters"

```
int count = 0;
std::string result;
std::string input( "I love Boost.Phoenix" );
iter_t iter = input.begin();
iter_t end = input.end();

qi::rule<iter_t, std::string()>
    count_rule %= *( char_[ ++ref(count) ] ) ;

qi::parse( iter, end,
    count_rule,
    result );

std::cout >> result >> " has "
    >> count >> " characters" >> std::endl;
```

# Phoenix Place Holders in Qi

Placeholder	Note
<code>_1, _2, ...</code>	Nth attribute of the parser.
<code>_val</code>	The enclosing rule's synthesized attribute.
<code>_r1, _r2, ...</code>	The enclosing rule's Nth inherited attribute.
<code>_a, _b, ..., _j</code>	The enclosing rule's local variables.
<code>_pass</code>	Assign <i>false</i> to force parser failure.

# Semantic Actions - Final bit of advice

- Favor attribute parsing
- Learn Boost.Phoenix
- Auto rule *compatibility* is your friend

```
rule    = some_cool_expression
      [
          _val = _1
      ];
```

`_val` assignment is *not* the same as auto attribute propagation

```
rule %= some_cool_expression;
rule = some_cool_expression;
```

# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- **To Skip or Not To Skip**
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Parser API

	No Skip	Skip
Iterator Based	<code>parse</code>	<code>phrase_parse</code>
Stream Based	<code>match</code>	<code>phrase_match</code>



# Provided Skippers

- A skipper can be any parser
- Two most common skippers used:

space	0x09(HT), 0x0a(NL), 0x0b(VT), 0x0c(NP), 0x0d(CR), 0x20(SP)
blank	0x09(HT), 0x20(SP)

# Custom Skipper Example - List of ints

Parse a list of integers.

## Example

1, 2,3 ,4,5

## Example

1, 2,3 ,  
4,5

## Grammar

```
phrase_parse( iter, end,  
              int_ % ',','  
              space );
```

# Custom Skipper Example - List of ints

Parse a list of integers.

## Example

1, 2,3 ,4,5

## Example

1, 2,3 ,  
4,5

## Grammar

```
phrase_parse( iter, end,  
              int_ % ',','  
              space );
```

# Custom Skipper Example - List of ints w/ comments

Parse a list of integers out of the comments.

## Example

```
| spirit is awesome  
|4,  
1,| joel de guzman  
2,3| and hartmut kaiser  
| perform  
,4,5 | magic
```

## Grammar

```
phrase_parse( iter, end,  
              int_ % ', ',  
              ?????? );
```

# Custom Skipper Example - List of ints w/ comments

Parse a list of integers out of the comments.

## Example

```
| spirit is awesome  
|4,  
1,| joel de guzman  
2,3| and hartmut kaiser  
| perform  
,4,5 | magic
```

## Grammar

```
phrase_parse( iter, end,  
              int_ % ', ',  
              ?????? );
```

# Custom Skipper Example - List of ints w/ comments

Parse a list of integers out of the comments.

## Example

```
| spirit is awesome
|4,
1,| joel de guzman
2,3| and hartmut kaiser
| perform
,4,5 | magic
```

## Grammar

```
phrase_parse( iter, end,
               int_ % ', ',
               ????? );
```

# Custom Skipper Example - List of ints w/ comments

Parse a list of integers out of the comments.

## Example

```
| spirit is awesome
|4,
1,| joel de guzman
2,3| and hartmut kaiser
| perform
,4,5 | magic
```

## Grammar

```
phrase_parse( iter, end,
               int_ % ' , ',
               ?????? );
```

# Custom Skipper Example - List of ints w/ comments

Parse a list of integers out of the comments.

## Example

```
| spirit is awesome  
|4,  
1,| joel de guzman  
2,3| and hartmut kaiser  
| perform  
,4,5 | magic
```

## Grammar

```
phrase_parse( iter, end,  
              int_ % ' ','',  
              ????? );
```



# Custom Skipper Example - Skipping Grammar

```
template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    skipper() : skipper::base_type(skip_it)
    {
        comment =    '|'
                    >> *( char_ - eol )
                    >> eol ;

        skip_it =
            comment
            | char_( " \x09\x0a\x0d" );
    }

    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
};
```

# Custom Skipper Example - Skipping Grammar

```
template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    skipper() : skipper::base_type(skip_it)
    {
        comment =    ' | '
                    >> *( char_ - eol )
                    >> eol ;

        skip_it =
            comment
            | char_( " \x09\x0a\x0d" );
    }

    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
};
```

# Custom Skipper Example - Skipping Grammar

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    skipper() : skipper::base_type(skip_it)
    {
        comment =      ' | '
                      >> *( char_ - eol )
                      >> eol ;

        skip_it =
            comment
            | char_( " \x09\x0a\x0d" );
    }

    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
};

```

# Custom Skipper Example - Skipping Grammar

```
template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    skipper() : skipper::base_type(skip_it)
    {
        comment =      ' | '
                      >> *( char_ - eol )
                      >> eol ;

        skip_it =
            comment
            | char_( " \x09\x0a\x0d" ) ;
    }

    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
};
```

# Custom Skipper Example - Skipping Grammar

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    skipper() : skipper::base_type(skip_it)
    {
        comment =
            ' | '
            >> *( char_ - eol )
            >> eol ;

        skip_it =
            comment
            | char_( " \x09\x0a\x0d" );
    }

    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
};

```

# Custom Skipper Example - Using Grammar

```
1 typedef std::string::iterator iter_t;
2 std::string input( "| spirit is awesome\n"
3                   "|4,\n"
4                   "1,| joel de guzman\n"
5                   " 2,3| and hartmut kaiser\n"
6                   "| perform\n"
7                   ",4,5 | magic\n" );
8
9 iter_t iter = input.begin();
10 iter_t end  = input.end();
11
12 std::vector<int> result;
13
14 phrase_parse( iter, end,
15              int_ % ',',
16              skipper<iter_t>(),
17              result );
```

# Custom Skipper Example - Changing Comment Start

Parse a list of integers out of the comments.

The start character for comments can change with a directive.

## Example

```
| spirit is awesome  
|4,  
1,| joel de guzman  
[Comment Char] %_char 2,3% and hartmut kaiser  
% perform  
,4,5 % magic
```

# Custom Skipper Example - Changing Comment Start

Parse a list of integers out of the comments.

The start character for comments can change with a directive.

## Example

```
| spirit is awesome
```

```
|4,
```

```
1,| joel de guzman
```

```
[Comment Char] %_char 2,3% and hartmut kaiser
```

```
% perform
```

```
,4,5 % magic
```



# Custom Skipper Example - Changing Comment Start

Parse a list of integers out of the comments.

The start character for comments can change with a directive.

## Example

```
| spirit is awesome  
|4,  
1,| joel de guzman  
[Comment Char] %_char 2,3% and hartmut kaiser  
% perform  
,4,5 % magic
```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
            >> *( char_ - eol )
            >> eol ;

        comment_directive = "[Comment Char] "
            > char_[ phx::bind( &my_type::set_comment_char, this,
                               qi::_1 ) ]
            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
            >> *( char_ - eol )
            >> eol ;

        comment_directive = "[Comment Char] "
            > char_[ phx::bind( &my_type::set_comment_char, this,
                               qi::_1 ) ]
            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
            >> *( char_ - eol )
            >> eol ;

        comment_directive = "[Comment Char] "
            > char_[ phx::bind( &my_type::set_comment_char, this,
                               qi::_1 ) ]
            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
                    >> *( char_ - eol )
                    >> eol ;

        comment_directive = "[Comment Char] "
                            > char_[ phx::bind( &my_type::set_comment_char, this,
                                                  qi::_1 ) ]
                            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
                    >> *( char_ - eol )
                    >> eol ;

        comment_directive = "[Comment Char] "
                            > char_[ phx::bind( &my_type::set_comment_char, this,
                                                  qi::_1 ) ]
                            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
            >> *( char_ - eol )
            >> eol ;

        comment_directive = "[Comment Char] "
            > char_[ phx::bind( &my_type::set_comment_char, this,
                               qi::_1 ) ]
            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
                    >> *( char_ - eol )
                    >> eol ;

        comment_directive = "[Comment Char] "
                            > char_[ phx::bind( &my_type::set_comment_char, this,
                                                qi::_1 ) ]
                            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```



# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
                    >> *( char_ - eol )
                    >> eol ;

        comment_directive = "[Comment Char] "
                            > char_[ phx::bind( &my_type::set_comment_char, this,
                                                  qi::_1 ) ]
                            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Adaptable Comment Start

```

template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = char_( phx::ref( comment_start_char ) )
            >> *( char_ - eol )
            >> eol ;

        comment_directive = "[Comment Char] "
            > char_[ phx::bind( &my_type::set_comment_char, this,
                               qi::_1 ) ]
            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it;
    qi::rule<Iterator> comment;
    qi::rule<Iterator> comment_directive;
};

```

# Custom Skipper Example - Using Grammar

```
1 typedef const char* iter_t;
2 iter_t iter =
3     "| spirit is awesome\n"
4     "|4,\n"
5     "1,| joel de guzman\n"
6     "[Comment Char] %_char 2,3% and hartmut kaiser\n"
7     "% perform\n"
8     ",4,5 % magic\n" ;
9
10 iter_t end = iter + std::strlen(iter);
11
12 std::vector<int> result;
13 phrase_parse( iter, end,
14             int_ % ',',
15             skipper<iter_t>('|'),
16             result );
```

# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- **Tid-bits**

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Expectation Exceptions

In the adaptable skipper we had:

```
1 | comment_directive =  
2 |     "[Comment Char] "  
3 |     > char_[ phx::bind( &my_type::set_comment_char, this,  
4 |                       qi::_1 ) ]  
5 |     > "_char" ;
```

What if our input is (notice *\_chr*):

```
1 | iter_t iter =  
2 |     "| coffee\n"  
3 |     "1,| and\n"  
4 |     "[Comment Char] !_chr 2,3 \n"  
5 |     "!coddng cookies\n"  
6 |     "! make the world go\n"  
7 |     ;
```

# Expectation Exception - Output

```
terminate called after throwing an instance of  
'boost::exception_detail::clone_impl<  
boost::exception_detail::error_info_injector<  
boost::spirit::qi::expectation_failure<  
char const*> > >'  
    what():  boost::spirit::qi::expectation_failure  
Aborted
```

# Expectation Exception - Add Error Handler

```
template <typename Iterator>
struct skipper : qi::grammar< Iterator >
{
    typedef skipper<Iterator> my_type;

    skipper(char comment_char) : skipper::base_type(skip_it),
                                comment_start_char(comment_char)
    {
        comment = lexeme[ char_( phx::ref( comment_start_char ) )
                            >> *( char_ - eol )
                            >> eol ] ;

        comment_directive = "[Comment Char] "
                            > char_[ phx::bind( &my_type::set_comment_char, this, qi::_1 ) ]
                            > "_char" ;

        skip_it =
            comment_directive
            | comment
            | char_( " \x09\x0a\x0d" );

        comment_directive.name("comment_directive");

        qi::on_error<qi::fail>( comment_directive,
                                std::cout << val("Error! Expecting ") << _4
                                << val(" here: \"") << construct<std::string>(_3,_2)
                                << val("\"\\n") );
    }

    void set_comment_char( char value ){ comment_start_char = value; }

    char comment_start_char;
    qi::rule<Iterator> skip_it, comment, comment_directive;
};
```

# Expectation Exception - Error Handler Output

Much more useful output.

```
Error! Expecting "_char" here: "_chr 2,3  
!coddling cookies  
! make the world go  
"
```



# Expectation Exception - Error Handler - Closer Look

## Registering an error handler.

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

Action	Description
fail	Quit and fail. Returns no_match.
retry	Attempt error recovery, possibly moving the iterator position.
accept	Force success, moving the iterator position appropriately.
rethrow	Rethrow the error.

# Expectation Exception - Error Handler - Closer Look

Call `qi::on_error<Action>(rule, handler)`

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

Action	Description
fail	Quit and fail. Returns <code>no_match</code> .
retry	Attempt error recovery, possibly moving the iterator position.
accept	Force success, moving the iterator position appropriately.
rethrow	Rethrow the error.

# Expectation Exception - Error Handler - Closer Look

Call `qi::on_error<Action>(rule, handler)`

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

Action	Description
fail	Quit and fail. Returns <code>no_match</code> .
retry	Attempt error recovery, possibly moving the iterator position.
accept	Force success, moving the iterator position appropriately.
rethrow	Rethrow the error.

# Expectation Exception - Error Handler - Closer Look

Call `qi::on_error<Action>(rule, handler)`

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

Argument	Description
first	The position of the iterator when the rule was entered.
last	The end of input.
error-pos	The actual position of the iterator where the error occurred.
what	What failed: a string describing the failure.

# Expectation Exception - Error Handler - Closer Look

Call `qi::on_error<Action>(rule, handler)`

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

Argument	Description
first	The position of the iterator when the rule was entered.
last	The end of input.
error-pos	The actual position of the iterator where the error occurred.
<b>what</b>	What failed: a string describing the failure.

# Expectation Exception - Error Handler - Closer Look

Call `qi::on_error<Action>(rule, handler)`

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

Argument	Description
first	The position of the iterator when the rule was entered.
<b>last</b>	The end of input.
<b>error-pos</b>	The actual position of the iterator where the error occurred.
what	What failed: a string describing the failure.

# Profit !

- 1 Learn Spirit
- 2 ???????
- 3 Profit !!!!

# Profit !

- 1 Learn Spirit
- 2 **Key / Value Grammar**
- 3 Profit !!!!



# Debugging - Key / Value Revisited

```
1  int main()
2  {
3      typedef const char* iter_t;
4      iter_t iter = "foo      : bar,"
5                   "falcou : 'crazy frenchman' " ;
6
7      iter_t iter_end = iter + std::strlen(iter);
8      std::map< std::string, std::string > key_value_map;
9      key_value_grammar<iter_t> grammar;
10
11     phrase_parse( iter, iter_end,
12                  grammar,
13                  qi::ascii::space,
14                  key_value_map );
15
16     std::for_each( key_value_map.begin(), key_value_map.end(),
17                   std::cout << phx::at_c<0>(arg1) << " : "
18                   << phx::at_c<1>(arg1) << '\n' );
19
20     return 1;
21 }
```

## Output:

```
foo : bar
```

# Debugging - Key / Value as a Grammar

```

1  template <typename Iterator>
2  struct key_value_grammar
3      : qi::grammar<Iterator, std::map<std::string, std::string>(), space_type>
4  {
5      key_value_grammar() : key_value_grammar::base_type(start)
6      {
7          start = item % "','';
8
9          item = name >> ':' >> ( quote | name );
10
11         name  = alpha >> *alnum;
12
13         quote %=      omit[ char_("\"'\" )[_a = _1] ]
14                     >> lexeme[ *( char_ - char_( _a ) ) ]
15                     >> omit[ char_( _a ) ];
16     }
17
18     qi::rule<Iterator, std::map<std::string, std::string>(), space_type> start;
19     qi::rule<Iterator, std::pair<std::string, std::string>(), space_type> item;
20     qi::rule<Iterator, std::string(), locals<char>, space_type> quote;
21     qi::rule<Iterator, std::string(), space_type> name;
22 };

```

# Debugging - Key / Value as a Grammar - Add Debug

```

1  template <typename Iterator>
2  struct key_value_grammar
3  : qi::grammar<Iterator, std::map<std::string, std::string>(), space_type>
4  {
5      key_value_grammar() : key_value_grammar::base_type( start )
6      {
7          start = item % "','';
8
9          item = name >> ':' >> ( quote | name );
10
11         name  = alpha >> *alnum;
12
13         quote %=      omit[ char_( "\"" )[_a = _1] ]
14                     >> lexeme[ *( char_ - char_( _a ) ) ]
15                     >> omit[ char_( _a ) ];
16
17         BOOST_SPIRIT_DEBUG_NODE( start );
18         BOOST_SPIRIT_DEBUG_NODE( item );
19         BOOST_SPIRIT_DEBUG_NODE( name );
20         BOOST_SPIRIT_DEBUG_NODE( quote );
21     }
22
23     qi::rule<Iterator, std::map<std::string, std::string>(), space_type> start;
24     qi::rule<Iterator, std::pair<std::string, std::string>(), space_type> item;
25     qi::rule<Iterator, std::string(), locals<char>, space_type> quote;
26     qi::rule<Iterator, std::string(), space_type> name;
27 };

```

# Debugging - Debug Support Requirements

## Enable debug support

```
1 | #define BOOST_SPIRIT_DEBUG // before including Spirit
```

## Ensure we can stream the synthesized attributes.

```
1 | // provided in std namespace for ADL
2 | namespace std
3 | {
4 |     template< typename T1, typename T2 >
5 |     std::ostream& operator<<( std::ostream& stream, const std::pair<T1,T2> & value )
6 |     {
7 |         stream << '(' << value.first << ":" << value.second << ')';
8 |         return stream;
9 |     }
10 |
11 |
12 |     template< typename T1, typename T2 >
13 |     std::ostream& operator<<( std::ostream& stream, const std::map<T1,T2> & value )
14 |     {
15 |         stream << '{';
16 |         std::for_each( value.begin(), value.end(),
17 |             stream << phx::val('(') << phx::at_c<0>(arg1)
18 |                 << ':' << phx::at_c<1>(arg1) << ')' );
19 |         stream << '}';
20 |         return stream;
21 |     }
22 | }
```

# Debugging - Failure Output

## Debug Output

```

1 <start>
2   <try>foo      : bar,falcou </try>
3 <item>
4   <try>foo      : bar,falcou </try>
5 <name>
6   <try>foo      : bar,falcou </try>
7   <success>     : bar,falcou : `</success>
8   <attributes>(foo)</attributes>
9 </name>
10 <quote>
11   <try> bar,falcou : `crazy</try>
12   <fail/>
13 </quote>
14 <name>
15   <try> bar,falcou : `crazy</try>
16   <success>,falcou : `crazy fre</success>
17   <attributes>(bar)</attributes>
18 </name>
19   <success>,falcou : `crazy fre</success>
20   <attributes>((foo:bar))</attributes>
21 </item>

```

## Grammar Rules

```

1 start = item % ' ';
2 item = name >> ':' >> ( quote | name );
3 name  = alpha >> *alnum;
4 quote %=
5     omit[ char_( "\"" )[_a = _1] ]
6     >> lexeme[ *( char_ - char_( _a ) ) ]
7     >> omit[ char_( _a ) ];

```

# Debugging - Failure Output *Continued*

## Debug Output *Continued*

```

1  <item>
2    <try>falcou : 'crazy fren</try>
3  <name>
4    <try>falcou : 'crazy fren</try>
5    <success> : 'crazy frenchman'</success>
6    <attributes>(falcou)</attributes>
7  </name>
8  <quote>
9    <try> 'crazy frenchman' </try>
10   <fail/>
11 </quote>
12 <name>
13   <try> 'crazy frenchman' </try>
14   <fail/>
15 </name>
16 <fail/>
17 </item>
18 <success>,falcou : 'crazy fre</success>
19 <attributes>({ (foo:bar) })</attributes>
20 </start>

```

## Grammar Rules

```

1  start = item % ' ';
2  item = name >> ':' >> ( quote | name );
3  name  = alpha >> *alnum;
4  quote %=
5      omit[ char_( "\"" )[_a = _1] ]
6      >> lexeme[ *( char_ - char_( _a ) ) ]
7      >> omit[ char_( _a ) ];

```

# Debugging - Success Output

## Debug Output *Fixed*

```

1 <item>
2   <try>falcou : 'crazy fren</try>
3 <name>
4   <try>falcou : 'crazy fren</try>
5   <success> : 'crazy frenchman'</success>
6   <attributes>(falcou)</attributes>
7 </name>
8 <quote>
9   <try> 'crazy frenchman' </try>
10  <success> </success>
11  <attributes>(crazy frenchman)</attributes><locals>(')</locals>
12 </quote>
13 <success> </success>
14 <attributes>((falcou:crazy frenchman))</attributes>
15 </item>
16 <success> </success>
17 <attributes>((falcou:crazy frenchman) (foo:bar))</attributes>
18 </start>

```

## Grammar Rules

```

1 start = item % ' ';
2 item = name >> ':' >> ( quote | name );
3 name = alpha >> *alnum;
4 quote %=
5   omit[ char_( "\"" )[_a = _1] ]
6   >> lexeme[ *( char_ - char_( _a ) ) ]
7   >> omit[ char_( _a ) ];

```

# Compile Errors

On occasion you may encounter a compilation error

- Rewarded with hundreds of thousands of lines of giberish
- Log onto IRC and see if Hartmut is in the room
- Check for compile time asserts (\*\*\*\*\*)
- Look for the first occurrence of your source file(s), follow to the line of the spirit file just above.



# Compile Errors

On occasion you may encounter a compilation error

- Rewarded with hundreds of thousands of lines of giberish
- Log onto IRC and see if Hartmut is in the room
- Check for compile time asserts (\*\*\*\*\*)
- Look for the first occurrence of your source file(s), follow to the line of the spirit file just above.

# Compile Errors

On occasion you may encounter a compilation error

- Rewarded with hundreds of thousands of lines of giberish
- Log onto IRC and see if Hartmut is in the room
- Check for compile time asserts (\*\*\*\*\*)
- Look for the first occurrence of your source file(s), follow to the line of the spirit file just above.

# Compile Errors

On occasion you may encounter a compilation error

- Rewarded with hundreds of thousands of lines of giberish
- Log onto IRC and see if Hartmut is in the room
- Check for compile time asserts (\*\*\*\*\*)
- Look for the first occurrence of your source file(s), follow to the line of the spirit file just above.

# Compile Errors

On occassion you may encounter a compilation error

- Rewarded with hundreds of thousands of lines of giberish
- Log onto IRC and see if Hartmut is in the room
- Check for compile time asserts (\*\*\*\*\*)
- Look for the first occurrence of your source file(s), follow to the line of the spirit file just above.

# Compile Errors - Compile-time Asserts

While changing over to the grammar had this:

```
1 | quote %=      omit[ char_( "\"" ) [_a = _1] ]
2 |             >> lexeme[ *( ~char_( _a ) ]
3 |             >> omit[ char_( _a ) ];
```

Searched for my filename:

```
parsing_strings_map.cpp:62:   instantiated from key_value_grammar<Iterator>::key_value_grammar
parsing_strings_map.cpp:88:   instantiated from here
../boost/spirit/home/qi/char/char_parser.hpp:141: error: no matching function for call to
assertion_failed(mpl_::failed***** (boost::spirit::qi::make_composite
...

```

In char\_parser.hpp

```
1 | template <typename Elements, typename Modifiers>
2 | struct make_composite<proto::tag::complement, Elements, Modifiers>
3 | {
4 |     typedef typename
5 |         fusion::result_of::value_at_c<Elements, 0>::type
6 |         subject;
7 |
8 |     BOOST_SPIRIT_ASSERT_MSG((
9 |         traits::is_char_parser<subject>::value
10 |     ), subject_is_not_negatable, (subject));
```

# Compile Errors - Follow the Error

```
1 | template <typename Expr>
2 | rule& operator=(Expr const& expr)
3 | {
4 |     // Report invalid expression error as early as possible.
5 |     // If you got an error_invalid_expression error message here,
6 |     // then the expression (expr) is not a valid spirit qi expression.
7 |     BOOST_SPIRIT_ASSERT_MATCH(qi::domain, Expr);
8 |
9 |     f = detail::bind_parser<mpl::false_>(compile<qi::domain>(expr));
10 |    return *this;
11 | }
```

```
1 | // If you are seeing a compilation error here, you are probably
2 | // trying to use a rule or a grammar which has inherited
3 | // attributes, without passing values for them.
4 | context_type context(attr_);
```

```
1 | // If you are seeing a compilation error here stating that the
2 | // forth parameter cant be converted to a qi::reference
3 | // then you are probably trying to use a rule or a grammar with
4 | // an incompatible skipper type.
5 | if (f(first, last, context, skipper))
```

# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- **Getting Started**
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Generator?

AST  $\rightarrow$  Karma  $\rightarrow$  Data Stream



# Everything you need to know

Qi	Karma
Consumes streams and generates attributes	Consumes attributes and generates streams
Uses >> to tie parsers together	Uses << to tie generators together
Skippers	Dilimeters
Executes semantic actions after successful parse	Executes semantic actions before generation

# Everything you need to know

Qi	Karma
Consumes streams and generates attributes	Consumes attributes and generates streams
Uses >> to tie parsers together	Uses << to tie generators together
Skippers	Dilimeters
Executes semantic actions after successful parse	Executes semantic actions before generation

# Everything you need to know

Qi	Karma
Consumes streams and generates attributes	Consumes attributes and generates streams
Uses >> to tie parsers together	Uses << to tie generators together
Skippers	Dilimeters
Executes semantic actions after successful parse	Executes semantic actions before generation

# Everything you need to know

Qi	Karma
Consumes streams and generates attributes	Consumes attributes and generates streams
Uses >> to tie parsers together	Uses << to tie generators together
Skippers	Dilimeters
Executes semantic actions after successful parse	Executes semantic actions before generation

# Motivation - List of ints

```
1  iterator_t iter =
2      "| coffee\n"
3      "1,| and\n"
4      "[Comment Char] !_char 2,3 !coddling cookies\n"
5      "! make the world go\n" ;
6
7  iterator_t end  = iter + std::strlen(iter);
8  std::vector<int> result;
9
10 bool r = phrase_parse( iter, end,
11                        qi::int_ % ',',
12                        skipper<iterator_t>( '|' ),
13                        result);
14
15 std::cout << karma::format( karma::int_ % ',',
16                             , result )
17           << std::endl;
```

## Output

1, 2, 3

# Motivation - Boost Authors - Classic

## Boost Library Author AST

```
1 struct boost_author{
2     boost::optional<std::string> name;
3     boost::optional<std::string> email;
4     boost::optional<std::vector< std::string > > libraries;
5 };
6
7 std::vector< boost_author > result;
```

## Generate via Classical Iterating over Containers

```
1 std::vector< boost_author >::iterator authors_iter = result.begin();
2 std::vector< boost_author >::iterator authors_iter_end = result.end();
3 while( authors_iter != authors_iter_end )
4 {
5     std::cout << "-----\nname: ";
6     if( authors_iter->name ){ std::cout << *(authors_iter->name); }
7     std::cout << "\nlibraries: ";
8     if( authors_iter->libraries )
9     {
10         std::vector<std::string>::iterator lib_iter = (authors_iter->libraries)->begin();
11         std::vector<std::string>::iterator lib_iter_end = (authors_iter->libraries)->end();
12         while( lib_iter != lib_iter_end )
13         {
14             std::cout << *lib_iter;
15             if( ++lib_iter != lib_iter_end ){ std::cout << ", "; }
16         }
17     }
18     std::cout << "\n";
19     ++authors_iter;
20 }
```

# Motivation - Boost Authors - Classic

## Output

```
-----  
name:  Hartmut Kaiser  
libraries:  spirit, wave  
-----  
name:  Joel de Guzman  
libraries:  spirit, phoenix, fusion, phoenix  
-----  
name:  Steven Watanabe  
libraries:  units
```

```
1 struct boost_author{  
2     boost::optional<std::string> name;  
3     boost::optional<std::string> email;  
4     boost::optional<std::vector< std::string > > libraries;  
5 };  
6  
7 std::vector< boost_author > result;
```

## Generate via Classical Iterating over Containers

```
1 std::vector< boost_author >::iterator authors_iter = result.begin();  
2 std::vector< boost_author >::iterator authors_iter_end = result.end();  
3 while( authors_iter != authors_iter_end )  
4 {  
5     std::cout << "-----\nname: ";  
6     if( authors_iter->name ){ std::cout << *(authors_iter->name); }  
7     std::cout << "\nlibraries: ";  
8     if( authors_iter->libraries ){  
9         for( auto& lib : *authors_iter->libraries )  
10             std::cout << lib << ", ";  
11     }  
12     std::cout << "\n";  
13     authors_iter++;  
14 }
```

# Motivation - Boost Authors - Karma

## Boost Library Author AST

```
1 | struct boost_author{
2 |     boost::optional<std::string> name;
3 |     boost::optional<std::string> email;
4 |     boost::optional<std::vector< std::string > > libraries;
5 | };
6 |
7 | BOOST_FUSION_ADAPT_STRUCT(
8 |     boost_author,
9 |     (boost::optional<std::string>, name)
10 |    (boost::optional<std::vector<std::string> >, libraries)
11 | )
12 |
13 | std::vector< boost_author > result;
```

## Karma Approach

```
1 | std::cout << karma::format( *(      karma::lit("-----\n")
2 |                               << "name: " << -karma::string << '\n'
3 |                               << "libraries: " << -( karma::string % ", " ) << '\n'
4 |                               )
5 |                               , result );
```



# Motivation - Boost Authors - Karma

## Output

```
-----  
name:  Hartmut Kaiser  
libraries:  spirit, wave  
-----  
name:  Joel de Guzman  
libraries:  spirit, phoenix, fusion, phoenix  
-----  
name:  Steven Watanabe  
libraries:  units
```

```
1 | struct boost_author{  
2 |     boost::optional<std::string> name;  
3 |     boost::optional<std::string> email;  
4 |     boost::optional<std::vector< std::string > > libraries;  
5 | };  
6 |  
7 | BOOST_FUSION_ADAPT_STRUCT(  
8 |     boost_author,  
9 |     (boost::optional<std::string>, name)  
10 |     (boost::optional<std::vector<std::string> >, libraries)  
11 | )  
12 |  
13 | std::vector< boost_author > result;
```

## Karma Approach

```
1 | std::cout << karma::format( *( karma::lit( "-----\n") )  
2 |
```

# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- **Generators Types and Attributes**
- Semantic Actions
- Delimiters / No-delimiters

# Some of the Available Generators

Numeric/Binary Generators look like Qi Parsers

Type	Generator
signed	<code>lit(num), short_, int_, long_, long_long,</code> <code>int_(-42)</code>
unsigned	<code>lit(num), bin, oct, hex, ushort_, ulong_,</code> <code>uint_, ulong_long, uint_(82)</code>
real	<code>lit(num), float_, double_, long_double,</code> <code>double_(123.5)</code>
boolean	<code>lit(b), bool_, bool_(b), true_, false_</code>
binary	<code>byte_, word, dword, qword, word(0xface)</code>
big endian	<code>big_word, big_dword, big_qword,</code> <code>big_dword(0xdeadbeef)</code>
litte endian	<code>litte_word, litte_dword, litte_qword,</code> <code>little_dword(0xefbeadde)</code>

# Some of the Available Generators

Generators without arguments consume a compatible attribute while generating.

Type	Generator
signed	<code>lit(num)</code> , <code>short_</code> , <code>int_</code> , <code>long_</code> , <code>long_long</code> , <code>int_(-42)</code>
unsigned	<code>lit(num)</code> , <code>bin</code> , <code>oct</code> , <code>hex</code> , <code>ushort_</code> , <code>ulong_</code> , <code>uint_</code> , <code>ulong_long</code> , <code>uint_(82)</code>
real	<code>lit(num)</code> , <code>float_</code> , <code>double_</code> , <code>long_double</code> , <code>double_(123.5)</code>
boolean	<code>lit(b)</code> , <code>bool_</code> , <code>bool_(b)</code> , <code>true_</code> , <code>false_</code>
binary	<code>byte_</code> , <code>word</code> , <code>dword</code> , <code>qword</code> , <code>word(0xface)</code>
big endian	<code>big_word</code> , <code>big_dword</code> , <code>big_qword</code> , <code>big_dword(0xdeadbeef)</code>
little endian	<code>litte_word</code> , <code>litte_dword</code> , <code>litte_qword</code> , <code>little_dword(0xefbeadde)</code>

# Some of the Available Generators

Numeric Generators with arguments consume compatible attributes while generating. The attribute value must match the generator's argument.

Type	Generator
signed	<code>lit(num), short_, int_, long_, long_long,</code> <code><b>int_(-42)</b></code>
unsigned	<code>lit(num), bin, oct, hex, ushort_, ulong_,</code> <code>uint_, ulong_long, <b>uint_(82)</b></code>
real	<code>lit(num), float_, double_, long_double,</code> <code><b>double_(123.5)</b></code>
boolean	<code>lit(b), bool_, bool_(b), <b>true_</b>, false_</code>
binary	<code>byte_, word, dword, qword, word(0xface)</code>
big endian	<code>big_word, big_dword, big_qword,</code> <code>big_dword(0xdeadbeef)</code>
little endian	<code>litte_word, litte_dword, litte_qword,</code> <code>little_dword(0xefbeadde)</code>

# Some of the Available Generators

Use `lit` to generate literal values for Numeric Generators. Binary Generators that take an argument will produce the literal.

Type	Generator
signed	<code>lit(num)</code> , <code>short_</code> , <code>int_</code> , <code>long_</code> , <code>long_long</code> , <code>int_(-42)</code>
unsigned	<code>lit(num)</code> , <code>bin</code> , <code>oct</code> , <code>hex</code> , <code>ushort_</code> , <code>ulong_</code> , <code>uint_</code> , <code>ulong_long</code> , <code>uint_(82)</code>
real	<code>lit(num)</code> , <code>float_</code> , <code>double_</code> , <code>long_double</code> , <code>double_(123.5)</code>
boolean	<code>lit(b)</code> , <code>bool_</code> , <code>bool_(b)</code> , <code>true_</code> , <code>false_</code>
binary	<code>byte_</code> , <code>word</code> , <code>dword</code> , <code>qword</code> , <code>word(0xface)</code>
big endian	<code>big_word</code> , <code>big_dword</code> , <code>big_qword</code> , <code>big_dword(0xdeadbeef)</code>
little endian	<code>litte_word</code> , <code>litte_dword</code> , <code>litte_qword</code> , <code>little_dword(0xefbeadde)</code>

# Some of the Available Generators

Type	Generator
character	<code>char_, char_('x'), char_(a),</code> <code>char_('a','z'), char_("a-z8A-Z"),</code> <code>~char_('a')</code> <code>lit('a'), 'a'</code>
string	<code>string("foo"), string(s), lit("bar"), "bar",</code> <code>lit(s)</code>
classification	<code>alnum, alpha, blank, cntrl, digit, graph,</code> <code>lower, print, punct, space, upper, xdigit</code>

# Some of the Available Generators

Generate any character while consuming the compatible attribute.

Type	Generator
character	<b>char_</b> , char_('x'), char(_a), char_('a','z'), char("a-z8A-Z"), ~char_('a') lit('a'), 'a'
string	string("foo"), string(s), lit("bar"), "bar", lit(s)
classification	alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit



# Some of the Available Generators

Only generate the matching character while consuming the compatible attribute.

Type	Generator
character	<code>char_, <b>char_('x')</b>, char_(a), char_('a','z'), char_("a-zA-Z"), ~char_('a')</code>
	<code>lit('a'), 'a'</code>
string	<code>string("foo"), string(s), lit("bar"), "bar", lit(s)</code>
classification	<code>alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit</code>

# Some of the Available Generators

Generate characters that are within the range while consuming the compatible attribute.

Type	Generator
character	<code>char_, char_('x'), char_(a),</code> <code><b>char_('a','z')</b>, char_("a-z8A-Z"),</code> <code>~char_('a')</code> <code>lit('a'), 'a'</code>
string	<code>string("foo"), string(s), lit("bar"), "bar",</code> <code>lit(s)</code>
classification	<code>alnum, alpha, blank, cntrl, digit, graph,</code> <code>lower, print, punct, space, upper, xdigit</code>

# Some of the Available Generators

Generate characters that match the character set definition while consuming the compatible attribute.

Type	Generator
character	<code>char_, char_('x'), char_(a),</code> <code>char_('a','z'), <b>char_("a-z8A-Z"),</b></code> <code>~char_('a')</code> <code>lit('a'), 'a'</code>
string	<code>string("foo"), string(s), lit("bar"), "bar",</code> <code>lit(s)</code>
classification	<code>alnum, alpha, blank, cntrl, digit, graph,</code> <code>lower, print, punct, space, upper, xdigit</code>

# Some of the Available Generators

Negating the character generator's test condition works in Karma too.

Type	Generator
character	<code>char_, char_('x'), char_(a),</code> <code>char_('a','z'), char_("a-zA-Z"),</code> <code><b>~char_('a')</b></code>
	<code>lit('a'), 'a'</code>
string	<code>string("foo"), string(s), lit("bar"), "bar",</code> <code>lit(s)</code>
classification	<code>alnum, alpha, blank, cntrl, digit, graph,</code> <code>lower, print, punct, space, upper, xdigit</code>

# Some of the Available Generators

Literals are generated via `lit`.

Type	Generator
character	<code>char_, char_('x'), char(_a),</code> <code>char_('a','z'), char_("a-z8A-Z"),</code> <code>~char_('a')</code> <code><b>lit('a'), 'a'</b></code>
string	<code>string("foo"), string(s), lit("bar"), <b>"bar"</b>,</code> <code>lit(s)</code>
classification	<code>alnum, alpha, blank, cntrl, digit, graph,</code> <code>lower, print, punct, space, upper, xdigit</code>

# Some Simple Examples

Using a character set definition.

```
1 | std::string value( "foo bar" );  
2 | std::cout << karma::format( *( karma::char_( "boa" ) )  
3 |                               , value )  
4 |                               << std::endl;
```

Output

ooba

# Some Simple Examples

## A negated character generator

```
1 | std::string value( "foo bar" );  
2 | std::cout << karma::format( *( ~karma::char_( 'o' ) )  
3 |                               , value )  
4 |                               << std::endl;
```

### Output

f bar

# Some Simple Examples

## An integer generator with immediate value

```
1  std::string generated;  
2  std::back_inserter_iterator<std::string> sink(generated);  
3  
4  int value = 40;  
5  karma::generate( sink,  
6                  karma::int_(42),  
7                  value );  
8  
9  std::cout << " " << generated << " " << std::endl;
```

### Output

```
  \ /
```



# Some Simple Examples

## Using a binary generator

```
1 | std::cout << " "
2 |           << karma::format( karma::byte_(0x30) )
3 |           << " " << std::endl;
```

### Output

```
'0'
```

# Generators Consume - Consumed Attributes

	Karma Generator Type	Attribute Type
Literals	'a', "abc", lit_(42), ...	No attribute
Primitives	int_, char_, double_, ...	int, char, double, ...
	bin, oct, hex	unsigned
	byte_, word, dword, ...	uint8_t, uint16_t, uint32_t, ...
	int_(42), char_('a'), string("abc")	attribute with specified value
	string	std::string
	symbol<A, B>	specified (A)
Non-terminal	rule<A()>, grammar<A()>	specified (A)
Operators	a << b (sequence)	fusion::vector<A, B>
	a   b (alternative)	boost::variant<A,B>
	*a (zero or more)	std::vector<A>
	+a (one or more)	std::vector<A>
	-a (optional, zero or one)	boost::optional<A>
	&a, !a (predicates)	No attribute
	a % b (list)	std::vector<A>

# Generator Examples

```
1 fusion::vector<int, std::string>
2     fusion_magic( 42, " IS the number" );
3 std::cout << karma::format( karma::int_ << karma::string
4                             , fusion_magic )
5     << std::endl;
```

42 IS the number

```
1 std::vector<int> numbers;
2 numbers.push_back(1);
3 numbers.push_back(8);
4 numbers.push_back(16);
5 std::cout << karma::format( int_ % ' - '
6                             , fusion_fun )
7     << std::endl;
```

1-8-16

# Directives

Directive	Note
<code>left_align[]</code> , <code>center[]</code> , <code>right_alight[]</code>	Aligns output from generator expression within column.
<code>repeat[]</code>	Repeats an generator expression with optional lower and upper counts.
<code>verbatim[]</code>	Disable automatic delimiting for embedded generator. Performs post delimiting.
<code>no_delimit[]</code>	Disable automatic delimiting for embedded generator.
<code>delimit[]</code>	Enable automatic delimiting for the embedded generator. Allows specification of the delimiting generator.
<code>upper[]</code> , <code>lower[]</code>	Force generation as upper or lower case.
<code>maxwidth[]</code>	Limit the overall length of the emmitted output.
<code>buffer[]</code>	The embedded generator is invoked but the output is buffered. If the embedded generator fails the buffer will be discarded, otherwise it will be emitted.
<code>omit[]</code>	The embedded generator is invoked and attributes consumed but no output is generated. Always succeeds.
<code>columns[]</code>	Seperates the ouput of the embedded generator into columns.

# Directive examples

```
1 | std::cout << format( ' | ' << right_align["boost"] << ' | ' );
```

```
| boost |
```

```
1 | std::cout << format( maxwidth(5) ["boostcon"] );
```

```
boost
```

```
1 | std::string city("Aspen");  
2 | std::cout << format( repeat(2,4) [char_  
3 |             , city )  
4 |             << std::endl;
```

```
Aspe
```

# Karma Does Rules

```
1 typedef std::back_insert_iterator<std::string> iter_t;
2 std::string generated;
3 iter_t sink(generated);
4
5 karma::rule<iter_t, std::pair<std::string, std::string>()> item;
6 item = *char_ << " : " << *char_;
7
8 std::pair<std::string, std::string> value = std::make_pair( "foo", "bar" );
9
10 karma::generate( sink,
11                 item,
12                 value );
13
14 std::cout << "' ' << generated << "' ' << std::endl;
```

## Output

```
'foo : bar'
```

# Karma Does Grammars

```
1  template <typename Iter>
2  struct key_value_generator
3  : karma::grammar<Iter, std::map<std::string, std::string>(), karma::space_type>
4  {
5      key_value_generator() : key_value_generator::base_type(start)
6      {
7          start = item % ',';
8
9          item = karma::string << ':' << karma::string;
10     }
11
12     karma::rule<Iter, std::map<std::string, std::string>(), karma::space_type> start;
13     karma::rule<Iter, std::pair<std::string, std::string>(), karma::space_type> item;
14 };
```

# Karma Does Grammars - Output

```
1 std::map< std::string, std::string > key_value_map;
2 key_value_map[ "foo" ]    = "bar";
3 key_value_map[ "quark" ]  = "floop";
4
5 typedef std::back_insert_iterator<std::string> iter_t;
6 std::string generated;
7 iter_t sink(generated);
8
9 key_value_generator<iter_t> generator;
10
11 karma::generate_delimited( sink,
12                             generator,
13                             karma::space,
14                             key_value_map );
15
16 std::cout << generated << std::endl;
```

## Output

```
foo : bar , quark : flop
```



# Example of Key/Value Output

Generate the key/value AST back for profit !

```
1 | iter_t iter = "foo      : bar,"
2 |               "gorp      : snork, "
3 |               "falcou    : 'crazy frenchman' " ;
4 | ...
5 |
6 | std::map< std::string, std::string > key_value_map;
7 |
8 | // magic profitable parsing
9 | ...
```

# Example of Key/Value Output - Generator

```
1 template <typename Iter>
2 struct key_value_generator
3     : karma::grammar<Iter, std::map<std::string, std::string>(), space_type>
4 {
5     key_value_generator() : key_value_generator::base_type(start)
6     {
7         start = item % ',';
8
9         item = karma::string << ':' << ( name | quoted );
10
11         name = karma::verbatim[ karma::strict[ alpha << *alnum ] ];
12
13         quoted = karma::verbatim[ '"' << karma::string << '"' ];
14     }
15
16     karma::rule<Iter, std::map<std::string, std::string>(), space_type> start;
17     karma::rule<Iter, std::pair<std::string, std::string>(), space_type> item;
18     karma::rule<Iter, std::string(), space_type> quoted;
19     karma::rule<Iter, std::string(), space_type> name;
20 };
```

# Example of Key/Value Output - Generator

```

1  template <typename Iter>
2  struct key_value_generator
3      : karma::grammar<Iter, std::map<std::string, std::string>(), space_type>
4  {
5      key_value_generator() : key_value_generator::base_type(start)
6      {
7          start = item % "','';
8
9          item = karma::string << ':' << ( name | quoted );
10
11         name = karma::verbatim[ karma::strict[ alpha << *alnum ] ];
12
13         quoted = karma::verbatim[ "'" << karma::string << "'" ];
14     }
15
16     karma::rule<Iter, std::map<std::string, std::string>(), space_type> start;
17     karma::rule<Iter, std::pair<std::string, std::string>(), space_type> item;
18     karma::rule<Iter, std::string(), space_type> quoted;
19     karma::rule<Iter, std::string(), space_type> name;
20 };

```

## Output

```

falcou : "crazy frenchman" , foo : bar ,
gorp :  snork

```

# Fusion Adapted

Can only adapt once... need something else to help

```

1 struct boost_author{
2     boost::optional<std::string> name;
3     boost::optional<std::string> email;
4     boost::optional<std::vector< std::string > > libraries;
5 };
6
7 BOOST_FUSION_ADAPT_STRUCT(
8     boost_author,
9     (boost::optional<std::string>, name)
10    (boost::optional<std::vector<std::string> >, libraries)
11 )

```

```

1 karma::rule< iter_t, boost_author() > author_libs_generator;
2 author_libs_generator = karma::lit("-----\n")
3                         << "name: " << -karma::string << '\n'
4                         << "libraries: " << -( karma::string % " " ) << '\n';
5
6
7 karma::rule< iter_t, boost_author() > author_email_generator;
8 author_email_generator = -karma::string
9                         << " < "
10                        << ( karma::string
11                            |
12                            "no email on record"
13                        )
14                        << " >\n";

```

# Fusion Adapted - Named

Give your adaptions unique names.

```

1 BOOST_FUSION_ADAPT_STRUCT_NAMED(
2     boost_author const, boost_author_libs_view,
3     (boost::optional<std::string>, name)
4     (boost::optional<std::vector<std::string> >, libraries)
5 )
6
7 BOOST_FUSION_ADAPT_STRUCT_NAMED(
8     boost_author const, boost_author_email_view,
9     (boost::optional<std::string>, name)
10    (boost::optional<std::string>, email)
11 )

1 karma::rule< iter_t, boost::fusion::adapted::boost_author_libs_view() >
2     author_libs_generator = karma::lit("-----\n")
3                             << "name: " << -karma::string << '\n'
4                             << "libraries: " << -( karma::string % " " ) << '\n';
5
6
7 karma::rule< iter_t, boost::fusion::adapted::boost_author_email_view() >
8     author_email_generator = -karma::string
9                             << " < "
10                             << ( karma::string
11                                 |
12                                 "no email on record"
13                             )
14                             << " >\n";

```

# Fusion Adapted - Class Named

## Deep Magic

```
1  class secrete_storage
2  {
3      public:
4          secrete_storage( int value ) : value_( value ) {}
5          int  get() const { return value_; }
6          void set( int value ){ value_ = value; }
7      private:
8          int value_;
9  };
10
11 BOOST_FUSION_ADAPT_CLASS_NAMED(
12     secrete_storage const, secrete_view,
13     (int, int, obj.obj.get(), obj.obj.set(val) )
14 )
15
16 int main()
17 {
18     typedef std::back_inserter<std::string> iter_t;
19     std::string generated;
20     iter_t sink(generated);
21
22     secrete_storage value( 42 );
23
24     karma::rule<iter_t, boost::fusion::adapted::secrete_view()> s_rule = karma::int_;
25
26     karma::generate( sink,
27                     s_rule,
28                     value );
29
30     std::cout << "the secrete is: " << generated << std::endl;
31     return 1;
32 }
```

# Fusion Adapted - Class Named

## Output

the secrete is: 42

```
1  class secrete_storage
2  {
3      public:
4          secrete_storage( int value ) : value_( value ) {}
5          int  get() const { return value_; }
6          void set( int value ){ value_ = value; }
7      private:
8          int value_;
9  };
10
11  BOOST_FUSION_ADAPT_CLASS_NAMED(
12      secrete_storage const, secrete_view,
13      (int, int, obj.obj.get(), obj.obj.set(val) )
14  )
15
16  int main()
17  {
18      typedef std::back_insert_iterator<std::string> iter_t;
19      std::string generated;
20      iter_t sink(generated);
21
22      secrete_storage value( 42 );
23
24      karma::rule<iter_t, boost::fusion::adapted::secrete_view()> s_rule = karma::int_;
25
26      karma::generate( sink,
27                      s_rule,
28                      value );
29
30      std::cout << "the secrete is: " << generated << std::endl;
```

# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- **Semantic Actions**
- Delimiters / No-delimiters



# Usage

Provides control *before* generation

- Can be attached to any non-terminal in the grammar
- Executes before generation
- Provides access to:
  - Generated attribute value
  - Inherited attribute values
  - Local variables
  - Ability to force generator failure

# Phoenix Place Holders in Karma

Placeholder	Note
<code>_1, _2, ...</code>	Nth attribute of the generator.
<code>_val</code>	The enclosing rule's synthesized attribute.
<code>_r1, _r2, ...</code>	The enclosing rule's Nth inherited attribute.
<code>_a, _b, ..., _j</code>	The enclosing rule's local variables.
<code>_pass</code>	Assign <i>false</i> to force generator failure.

VeXocide: urgh, we have too many `_1`'s

*a.k.a. Jeroen Habraken*

# Example - Generate Even Numbers

If a value is odd, increment it to be even.

```
1  std::vector<int> value;
2  value.push_back( 1 );
3  value.push_back( 4 );
4  value.push_back( 171 );
5  value.push_back( 192 );
6
7  cout << format_delimited( *( int_[ if_( _1 % 2 == 1 )
8                               [
9                               _1 = _1 + 1
10                              ]
11                             ] )
12                               , boost::spirit::ascii::space
13                               , value )
14      << std::endl;
```

## Output

2 4 172 192

# Outline

## 1 Motivation

- Ad-hoc Solutions
- The Spirit Way

## 2 Qi 101

- Parsers
- Attributes and Actions
- To Skip or Not To Skip
- Tid-bits

## 3 Karma 101

- Getting Started
- Generators Types and Attributes
- Semantic Actions
- Delimiters / No-delimiters

# Generator API

	No Delimination	Deliminate
Iterator Based	<code>generate</code>	<code>generate_delimited</code>
Stream Based	<code>format</code>	<code>format_generated</code>

# Delimiters Are Just Karma Expressions

```
1 std::vector<int> value;  
2 value.push_back( 1 );  
3 value.push_back( 4 );  
4 value.push_back( 171 );  
5 value.push_back( 192 );  
6  
7 std::cout << karma::format_delimited  
8             ( *karma::int_  
9               , karma::lit(':')  
10              , value )  
11             << std::endl;
```

## Output

```
1:4:171:192:
```

## Part II

# Examples

# Outline

- 4 Protocol Translator
  - The Problem
  - The Solution
- 5 HTTP Request
  - The Request
  - The URI
- 6 XML
  - What is in a name?



# Translator

Data Type A  $\rightarrow$  Data Type B

# The Incoming Protocol

0xbabe      0xfb0c      0x1 0x4 0x3      0xdead  
Start of Message    Command      Data Payload      End of Message

# The Outgoing Protocol

```
1 <update>
2   <product path='version'>1.4.3</product>
3 </update>
```

- XML format
- *path* attribute is the command
- command specific data in *product* node data

# Outline

- 4 Protocol Translator
  - The Problem
  - The Solution
- 5 HTTP Request
  - The Request
  - The URI
- 6 XML
  - What is in a name?

# Translator

Data Type A  $\rightarrow$  AST  $\rightarrow$  Data Type B

# Translator

Data Type A  $\rightarrow$  **Qi**  $\rightarrow$  AST  $\rightarrow$  **Karma**  $\rightarrow$  Data Type B

# Attribute

Start of Message	Command	Data Payload	End of Message
0xbabe	0xfa05	0x01 0x54	0xdead

```
struct message_t
{
    uint16_t command;
    std::vector< uint8_t > data;
};

BOOST_FUSION_ADAPT_STRUCT(
    message_t,
    (uint16_t,  command)
    (std::vector< uint8_t >, data)
)
```

# Attribute

Start of Message	<b>Command</b>	Data Payload	End of Message
0xbabe	0xfa05	0x01 0x54	0xdead

```
struct message_t
{
    uint16_t command;
    std::vector< uint8_t > data;
};

BOOST_FUSION_ADAPT_STRUCT(
    message_t,
    (uint16_t,  command)
    (std::vector< uint8_t >, data)
)
```



# Attribute

Start of Message	Command	Data Payload	End of Message
0xbabe	0xfa05	0x01 0x54	0xdead

```
struct message_t
{
    uint16_t command;
    std::vector< uint8_t > data;
};

BOOST_FUSION_ADAPT_STRUCT(
    message_t,
    (uint16_t,  command)
    (std::vector< uint8_t >, data)
)
```

# Attribute

Start of Message	Command	Data Payload	End of Message
0xbabe	0xfa05	0x01 0x54	0xdead

```
struct message_t
{
    uint16_t command;
    std::vector< uint8_t > data;
};

BOOST_FUSION_ADAPT_STRUCT(
    message_t,
    (uint16_t,    command)
    (std::vector< uint8_t >, data)
)
```

# Attribute

Start of Message	Command	Data Payload	End of Message
0xbabe	0xfa05	0x01 0x54	0xdead

```
struct message_t
{
    uint16_t command;
    std::vector< uint8_t > data;
};

BOOST_FUSION_ADAPT_STRUCT(
    message_t,
    (uint16_t,    command)
    (std::vector< uint8_t >, data)
)
```

# Attribute

Start of Message	Command	Data Payload	End of Message
0xbabe	0xfa05	0x01 0x54	0xdead

```
struct message_t
{
    uint16_t command;
    std::vector< uint8_t > data;
};

BOOST_FUSION_ADAPT_STRUCT(
    message_t,
    (uint16_t,  command)
    (std::vector< uint8_t >, data)
)
```

# Attribute

Start of Message	Command	Data Payload	End of Message
0xbabe	0xfa05	0x01 0x54	0xdead

```
struct message_t
{
    uint16_t command;
    std::vector< uint8_t > data;
};

BOOST_FUSION_ADAPT_STRUCT(
    message_t,
    (uint16_t,    command)
    (std::vector< uint8_t >, data)
)
```

# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	uint16_t	std::vector<uint8_t>	0xdead

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;  
  
parse_rule_t read_rule =  
    omit[ *( !big_word(0xbabe) » byte_ ) ]  
    » big_word(0xbabe)  
    » little_word  
    » *( byte_ - big_word( 0xdead ) )  
    » big_word( 0xdead )  
    ;
```

# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	uint16_t	std::vector<uint8_t>	0xdead

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;
```

```
parse_rule_t read_rule =
    omit[ *( !big_word(0xbabe) » byte_ ) ]
    » big_word(0xbabe)
    » little_word
    » *( byte_ - big_word( 0xdead ) )
    » big_word( 0xdead )
    ;
```

# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	<b>uint16_t</b>	<b>std::vector&lt;uint8_t&gt;</b>	0xdead

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;
```

```
parse_rule_t read_rule =
    omit[ *( !big_word(0xbabe) » byte_ ) ]
    » big_word(0xbabe)
    » little_word
    » *( byte_ - big_word( 0xdead ) )
    » big_word( 0xdead )
    ;
```



# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	uint16_t	std::vector<uint8_t>	0xdead

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;  
  
parse_rule_t read_rule =  
    omit[ *( !big_word(0xbabe) » byte_ ) ]  
    » big_word(0xbabe)  
    » little_word  
    » *( byte_ - big_word( 0xdead ) )  
    » big_word( 0xdead )  
    ;
```

# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	<b>uint16_t</b>	std::vector<uint8_t>	0xdead

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;

parse_rule_t read_rule =
    omit[ *( !big_word(0xbabe) » byte_ ) ]
    » big_word(0xbabe)
    » little_word
    » *( byte_ - big_word( 0xdead ) )
    » big_word( 0xdead )
    ;
```

# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	uint16_t	<b>std::vector&lt;uint8_t&gt;</b>	0xdead

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;
```

```
parse_rule_t read_rule =
    omit[ *( !big_word(0xbabe) » byte_ ) ]
    » big_word(0xbabe)
    » little_word
    » *( byte_ - big_word( 0xdead ) )
    » big_word( 0xdead )
    ;
```

# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	uint16_t	std::vector<uint8_t>	<b>0xdead</b>

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;
```

```
parse_rule_t read_rule =
    omit[ *( !big_word(0xbabe) » byte_ ) ]
    » big_word(0xbabe)
    » little_word
    » *( byte_ - big_word( 0xdead ) )
    » big_word( 0xdead )
    ;
```

# Qi : Parsing Rule

SOM	Command	Data Payload	EOM
0xbabe	uint16_t	std::vector<uint8_t>	0xdead

```
typedef qi::rule< iter_t, message_t() > parse_rule_t;
```

```
parse_rule_t read_rule =
    omit[ *( !big_word(0xbabe) » byte_ ) ]
    » big_word(0xbabe)
    » little_word
    » *( byte_ - big_word( 0xdead ) )
    » big_word( 0xdead )
    ;
```

# Karma : Generating Warm-up

SOM	Command	Data Payload	EOM
	uint16_t	std::vector<uint8_t>	
0xbabe	0xfa05	0x01 0x54	0xdead

```
typedef karma::rule< iter_t, message_t() > write_rule_t;
```

```
write_rule_t write_rule =
    big_word(0xbabe)
    » little_word
    » *byte_
    » big_word( 0xdead )
    ;
```

# Karma : Generating Warm-up

SOM	Command	Data Payload	EOM
	uint16_t	std::vector<uint8_t>	
0xbabe	0xfa05	0x01 0x54	0xdead

```
typedef karma::rule< iter_t, message_t() > write_rule_t;
```

```
write_rule_t write_rule =
    big_word(0xbabe)
    » little_word
    » *byte_
    » big_word( 0xdead )
    ;
```

# Karma : Generating Warm-up

SOM	Command	Data Payload	EOM
	<code>uint16_t</code>	<code>std::vector&lt;uint8_t&gt;</code>	
0xbabe	0xfa05	0x01 0x54	0xdead

```
typedef karma::rule< iter_t, message_t() > write_rule_t;
```

```
write_rule_t write_rule =
    big_word(0xbabe)
    » little_word
    » *byte_
    » big_word( 0xdead )
    ;
```



# Karma : Generating Warm-up

SOM	Command	Data Payload	EOM
	uint16_t	std::vector<uint8_t>	
0xbabe	0xfa05	0x01 0x54	0xdead

```
typedef karma::rule< iter_t, message_t() > write_rule_t;
```

```
write_rule_t write_rule =
```

```
    big_word(0xbabe)
    » little_word
    » *byte_
    » big_word( 0xdead )
    ;
```

# Karma : Generating Warm-up

SOM	Command	Data Payload	EOM
	<b>uint16_t</b>	std::vector<uint8_t>	
0xbabe	<b>0xfa05</b>	0x01 0x54	0xdead

```
typedef karma::rule< iter_t, message_t() > write_rule_t;
```

```
write_rule_t write_rule =
    big_word(0xbabe)
    » little_word
    » *byte_
    » big_word( 0xdead )
    ;
```

# Karma : Generating Warm-up

SOM	Command	Data Payload	EOM
	uint16_t	<code>std::vector&lt;uint8_t&gt;</code>	
0xbabe	0xfa05	0x01 0x54	0xdead

```
typedef karma::rule< iter_t, message_t() > write_rule_t;
```

```
write_rule_t write_rule =
    big_word(0xbabe)
    » little_word
    » *byte_
    » big_word( 0xdead )
    ;
```

# Karma : Generating Warm-up

SOM	Command	Data Payload	EOM
	uint16_t	std::vector<uint8_t>	
0xbabe	0xfa05	0x01 0x54	<b>0xdead</b>

```
typedef karma::rule< iter_t, message_t() > write_rule_t;
```

```
write_rule_t write_rule =
    big_word(0xbabe)
    » little_word
    » *byte_
    » big_word( 0xdead )
    ;
```

## The Ying and the Yang

### Qi : Parser

```
using qi::omit;  
using qi::big_word;  
using qi::little_word;  
using qi::byte_;  
  
read_rule_t read_rule =  
    omit[*(!big_word(0xbabe) » byte_)]  
    » big_word(0xbabe)  
    » little_word  
    » *( byte_ - big_word(0xdead) )  
    » big_word(0xdead)  
    ;
```

### Karma : Generator

```
using karma::big_word;  
using karma::little_word;  
using karma::byte_;  
  
write_rule_t write_rule =  
    big_word(0xbabe)  
    « little_word  
    « *byte_  
    « big_word(0xdead)  
    ;
```

# Karma : Generation Rule

```
struct{ uint16_t command; vector<uint8_t> data; }
```

## Simple Response

```
<update>  
  <product path='set_cal'>ok</product>  
</update>
```

## Version

```
<update>  
  <product path='version'>1.4.3</product>  
</update>
```

## Serial Number

```
<update>  
  <product path='serial_number'>826</product>  
</update>
```

# Karma : Generation Rule

```
struct{ uint16_t command; vector<uint8_t> data; }
```

## Simple Response

```
<update>  
  <product path='set_cal'>ok</product>  
</update>
```

## Version

```
<update>  
  <product path='version'>1.4.3</product>  
</update>
```

## Serial Number

```
<update>  
  <product path='serial_number'>826</product>  
</update>
```

# Karma : Generation Rule

```
struct{ uint16_t command; vector<uint8_t> data; }
```

## Simple Response

```
<update>  
  <product path='set_cal'>ok</product>  
</update>
```

## Version

```
<update>  
  <product path='version'>1.4.3</product>  
</update>
```

## Serial Number

```
<update>  
  <product path='serial_number'>826</product>  
</update>
```



# Karma : Generation Rule : response

```
1 struct message_t {  
2     uint16_t command;  
3     std::vector<uint8_t> data; };
```

```
<update>  
<product path="set_cal">ok</product>  
</update>
```

```
karma::rule< Iterator, message_t() > response;
```

```
response =  
    lit( "<update><product path=\"\" )  
    « (  
        simple_response  
        | version  
        | serial_number  
    )  
    « "</product></update>"  
    ;
```

# Karma : Generation Rule : response

```
1 struct message_t {  
2     uint16_t command;  
3     std::vector<uint8_t> data; };
```

```
<update>  
<product path="set_cal">ok</product>  
</update>
```

```
karma::rule< Iterator, message_t () > response;
```

```
response =  
    lit( "<update><product path=\"\" )  
    « (  
        simple_response  
        | version  
        | serial_number  
    )  
    « "</product></update>"  
    ;
```

# Karma : Generation Rule : response

```
1 struct message_t {  
2     uint16_t command;  
3     std::vector<uint8_t> data; };
```

```
<update>  
<product path="set_cal">ok</product>  
</update>
```

```
karma::rule< Iterator, message_t() > response;
```

```
response =
```

```
    lit( "<update><product path=\"\" )  
    « (  
        simple_response  
        | version  
        | serial_number  
    )  
    « "</product></update>"  
    ;
```

# Karma : Generation Rule : response

```
1 struct message_t {  
2     uint16_t command;  
3     std::vector<uint8_t> data; };
```

```
<update>  
<product path="set_cal">ok</product>  
</update>
```

```
karma::rule< Iterator, message_t() > response;
```

```
response =  
    lit( "<update><product path=\"\" )  
    « (  
        simple_response  
        | version  
        | serial_number  
    )  
    « "</product></update>"  
    ;
```

# Karma : Generation Rule : response

```
1 struct message_t {  
2     uint16_t command;  
3     std::vector<uint8_t> data; };
```

```
<update>  
  <product path="version">1.4.3</product>  
</update>
```

```
karma::rule< Iterator, message_t() > response;
```

```
response =  
    lit( "<update><product path=\"\" )  
    « (  
        simple_response  
        | version  
        | serial_number  
    )  
    « "</product></update>"  
    ;
```

# Karma : Generation Rule : response

```
1 struct message_t {  
2     uint16_t command;  
3     std::vector<uint8_t> data; };
```

```
<update>  
<product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator, message_t() > response;
```

```
response =  
    lit( "<update><product path=\"\" )  
    « (  
        simple_response  
        | version  
        | serial_number  
    )  
    « "</product></update>"  
    ;
```

# Karma : Generation Rule : response

```
1 struct message_t {  
2     uint16_t command;  
3     std::vector<uint8_t> data; };
```

```
<update>  
<product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator, message_t() > response;
```

```
response =  
    lit( "<update><product path=\"\" )  
    « (  
        simple_response  
        | version  
        | serial_number  
    )  
    « "</product></update>"  
    ;
```

# Karma : Generation Rule : simple\_response

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };  

```

```
<update><product path="set_cal">fail</product></update>
```

```
karma::rule< Iterator, message_t() > simple_response;
```

```
simple_response =  
(  
    ( &uint_( 0xfb01 ) « "set_cal\">ok" )  
| ( &uint_( 0xfb02 ) « "set_cal\">fail" )  
| ( &uint_( 0xfb07 ) « "store_table\">ok" )  
| ( &uint_( 0xfb08 ) « "store_table\">fail" )  
| ( &uint_( 0xfb0b ) « "ping\">" )  
)  
« omit[ *uint_ ]  
;
```



# Karma : Generation Rule : simple\_response

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update><product path="set_cal">fail</product></update>
```

```
karma::rule< Iterator, message_t () > simple_response;
```

```
simple_response =  
(  
    ( &uint_( 0xfb01 ) « "set_cal\">ok" )  
| ( &uint_( 0xfb02 ) « "set_cal\">fail" )  
| ( &uint_( 0xfb07 ) « "store_table\">ok" )  
| ( &uint_( 0xfb08 ) « "store_table\">fail" )  
| ( &uint_( 0xfb0b ) « "ping\">" )  
)  
« omit[ *uint_ ]  
;
```

# Karma : Generation Rule : simple\_response

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update><product path="set_cal">fail</product></update>
```

```
karma::rule< Iterator, message_t() > simple_response;
```

```
simple_response =  
(  
    ( &uint_( 0xfb01 ) << "set_cal\">ok" )  
    | ( &uint_( 0xfb02 ) << "set_cal\">fail" )  
    | ( &uint_( 0xfb07 ) << "store_table\">ok" )  
    | ( &uint_( 0xfb08 ) << "store_table\">fail" )  
    | ( &uint_( 0xfb0b ) << "ping\">" )  
)  
<< omit[ *uint_ ]  
;
```

# Karma : Generation Rule : simple\_response

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update><product path="set_cal">fail</product></update>
```

```
karma::rule< Iterator, message_t() > simple_response;
```

```
simple_response =  
(  
    ( &uint_( 0xfb01 ) « "set_cal\">ok" )  
    | ( &uint_( 0xfb02 ) « "set_cal\">fail" )  
    | ( &uint_( 0xfb07 ) « "store_table\">ok" )  
    | ( &uint_( 0xfb08 ) « "store_table\">fail" )  
    | ( &uint_( 0xfb0b ) « "ping\">" )  
)  
« omit[ *uint_ ]  
;
```

# Karma : Generation Rule : simple\_response

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update><product path="set_cal">fail</product></update>
```

```
karma::rule< Iterator, message_t() > simple_response;
```

```
simple_response =  
(  
    ( &uint_( 0xfb01 ) << "set_cal\">ok" )  
    | ( &uint_( 0xfb02 ) << "set_cal\">fail" )  
    | ( &uint_( 0xfb07 ) << "store_table\">ok" )  
    | ( &uint_( 0xfb08 ) << "store_table\">fail" )  
    | ( &uint_( 0xfb0b ) << "ping\">" )  
)  
<< omit[ *uint_ ]  
;
```

# Karma : Generation Rule : simple\_response

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;    };
```

```
<update><product path="set_cal">fail</product></update>
```

```
karma::rule< Iterator, message_t() > simple_response;
```

```
simple_response =  
(  
    ( &uint_( 0xfb01 ) « "set_cal\">ok" )  
| ( &uint_( 0xfb02 ) « "set_cal\">fail" )  
| ( &uint_( 0xfb07 ) « "store_table\">ok" )  
| ( &uint_( 0xfb08 ) « "store_table\">fail" )  
| ( &uint_( 0xfb0b ) « "ping\">" )  
)  
« omit[ *uint_ ]  
;
```

# Karma : Generation Rule : version

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
  <product path="version">1.4.3</product>  
</update>
```

```
karma::rule< Iterator, message_t() > version;
```

```
version =  
    &uint_(0xfb0c)  
    « "version\">  
    « ( uint_ % "." )  
    ;
```

# Karma : Generation Rule : version

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
  <product path="version">1.4.3</product>  
</update>
```

```
karma::rule< Iterator, message_t () > version;
```

```
version =  
    &uint_(0xfb0c)  
    « "version\">  
    « ( uint_ % "." )  
    ;
```

# Karma : Generation Rule : version

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
  <product path="version">1.4.3</product>  
</update>
```

```
karma::rule< Iterator, message_t() > version;
```

```
version =  
    &uint_(0xfb0c)  
    « "version\">"  
    « ( uint_ % "." )  
    ;
```



# Karma : Generation Rule : version

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
    <product path="version">1.4.3</product>  
</update>
```

```
karma::rule< Iterator, message_t() > version;
```

```
version =  
    &uint_(0xfb0c)  
    « "version\">>"  
    « ( uint_ % "." )  
    ;
```

# Karma : Generation Rule : version

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;    };  

```

```
<update>  
    <product path="version">1.4.3</product>  
</update>
```

```
karma::rule< Iterator, message_t() > version;
```

```
version =  
    &uint_(0xfb0c)  
    « "version\">"  
    « ( uint_ % "." )  
    ;
```

# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;    };
```

```
<update>  
    <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    « "serial_number\">  
    « omit[      eps      [_a = val(0)]  
        « *(int_      [_a = _a * val(256) + _1] )  
        ]  
    « int_      [_1 = _a]  
    ;
```

# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
  <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    « "serial_number\">  
    « omit[      eps      [_a = val(0)]  
        « *(int_      [_a = _a * val(256) + _1] )  
        ]  
    « int_      [_1 = _a]  
    ;
```

# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;    };
```

```
<update>  
  <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    « "serial_number\">  
    « omit[      eps      [_a = val(0)]  
        « *(int_      [_a = _a * val(256) + _1] )  
        ]  
    « int_      [_1 = _a]  
    ;
```

# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
  <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    « "serial_number\">  
    « omit[      eps      [_a = val(0)]  
        « *(int_      [_a = _a * val(256) + _1] )  
        ]  
    « int_      [_1 = _a]  
    ;
```

# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
  <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    « "serial_number\">  
    « omit[      eps      [_a = val(0)]  
        « *(int_      [_a = _a * val(256) + _1] )  
        ]  
    « int_      [_1 = _a]  
    ;
```

# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;    };
```

```
<update>  
    <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    « "serial_number\">  
    « omit[      eps      [_a = val(0)]  
        « *(int_      [_a = _a * val(256) + _1] )  
        ]  
    « int_      [_1 = _a]  
    ;
```



# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;    };
```

```
<update>  
    <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    << "serial_number\">  
    << omit[      eps      [_a = val(0)]  
        << *(int_      [_a = _a * val(256) + _1] )  
        ]  
    << int_      [_1 = _a]  
    ;
```

# Karma : Generation Rule : serial\_number

```
struct message_t{  
    uint16_t command; vector<uint8_t> data;  };
```

```
<update>  
  <product path="serial_number">826</product>  
</update>
```

```
karma::rule< Iterator,  
            locals<int>, message_t() > serial_number;
```

```
serial_number %=  
    &uint_(0xfb03)  
    « "serial_number\">  
    « omit[      eps      [_a = val(0)]  
        « *(int_      [_a = _a * val(256) + _1] )  
    ]  
    « int_      [_1 = _a]  
    ;
```

# Karma : Generation Grammar

```
1  template <typename Iterator>
2  struct xml_message_grammar : karma::grammar< Iterator, message_t() >
3  {
4      xml_message_grammar() : xml_message_grammar::base_type( response )
5      {
6          simple_response =
7              ( ( &uint_(0xfb01) << "set_cal\">ok" )
8                | ( &uint_(0xfb02) << "set_cal\">fail" )
9                | ( &uint_(0xfb07) << "store_table\">ok" )
10               | ( &uint_(0xfb08) << "store_table\">fail" )
11               | ( &uint_(0xfb0b) << "ping\">" )
12             )
13             << omit[ *uint_ ] ;
14
15          cal_value =
16              &uint_( 0xfb03 )
17              << "cal_value\">"
18              << ( uint_ % " " ) ;
19
20          version =
21              &uint_( 0xfb0c )
22              << "version\">"
23              << ( uint_ % "." ) ;
24
25          response =
26              lit( "<update><product path=\"" )
27              << ( simple_response
28                  | cal_value
29                  | version
30                )
31              << "</product></update>" ;
32      }
33
34      karma::rule< Iterator, message_t() > simple_response, cal_value, version, response;
35  };
```

# Translator : Pulling It Together

```
1  int main()
2  {
3      parse_iter_t input_iter = "\x43\x12\xba\xad"    // trash to be flushed
4                                  "\xba\xbe"           // start of message
5                                  "\x0c\xfb"           // command is little endian
6                                  "\x01\x04\x03"        // data
7                                  "\xde\xad" ;          // end of message
8
9      parse_iter_t input_iter_end = input_iter + std::strlen(input_iter);
10
11     parse_rule_t read_rule =
12         qi::omit[ *( !big_word(0xbabe) >> byte_ ) ]
13         >> big_word(0xbabe)
14         >> little_word
15         >> *( byte_ - big_word(0xdead) )
16         >> big_word(0xdead)
17         ;
18
19     message_t message;
20
21     qi::parse( input_iter, input_iter_end, read_rule, message );
22
23     std::string xml_message;
24     gen_iter_t sink( xml_message);
25
26     xml_message_grammar<gen_iter_t> output_grammar;
27
28     if( karma::generate( sink, output_grammar, message ) )
29     {
30         std::cout << "generated: " << xml_message << std::endl;
31     }
32
33     return 0;
34 }
```

# Outline

## 4 Protocol Translator

- The Problem
- The Solution

## 5 HTTP Request

- The Request
- The URI

## 6 XML

- What is in a name?

# Our Data Structure

```
1 namespace omd{ namespace http{ namespace request{
2
3 enum method_t
4 {
5     REQUEST_OPTIONS,
6     REQUEST_GET,
7     REQUEST_HEAD,
8     REQUEST_POST,
9     REQUEST_PUT,
10    REQUEST_DELETE,
11    REQUEST_TRACE,
12    REQUEST_CONNECT
13 };
14
15 struct request_line_t
16 {
17     method_t    method;
18     std::string uri;
19     std::string version;
20 };
21
22 struct message
23 {
24     typedef std::map< std::string, std::string > headers_t;
25
26     request_line_t request;
27     headers_t      headers;
28 };
29 }}}
```

# The Rules

```
1 message =
2     request_line
3     >> *header_pair
4     >> crlf
5     ;
6
7 request_line =
8     method_symbol >> ' '
9     >> uri >> ' '
10    >> http_version
11    >> crlf
12    ;
13
14 crlf = lexeme[ lit( '\x0d' ) >> lit( '\x0a' ) ];
15
16 uri = +( ~char_( ' ' ) );
17
18 http_version = lexeme[ "HTTP/" >> raw[ int_ >> '.' >> int_ ] ];
19
20 header_pair = token >> ':' >> lws >> field_value >> crlf ;
21
22 field_value = *( char_ - crlf );
23
24 lws = omit[ -crlf >> *char_( " \x09" ) ] ;
25
26 token = +( ~char_( " ()<>@,;:\\""/[]?={ } \x09" ) );
```

# Parsing the Request Line

```
1 qi::rule< Iterator, omd::http::request::request_line_t() > request_line;  
2 request_line =  
3     method_symbol >> ' ' '  
4     >> uri >> ' ' '  
5     >> http_version  
6     >> crlf  
7     ;
```

```
1 enum method_t  
2 {  
3     REQUEST_OPTIONS, REQUEST_GET, REQUEST_HEAD, REQUEST_POST,  
4     REQUEST_PUT, REQUEST_DELETE, REQUEST_TRACE, REQUEST_CONNECT  
5 };  
6  
7 struct request_line_t  
8 {  
9     method_t    method;  
10    std::string uri;  
11    std::string version;  
12 };
```



# The Symbol Table

```
1  /**
2   * symbol table to describe the valid request methods
3   */
4   struct method_symbol_ : qi::symbols< char, omd::http::request::method_t >
5   {
6   method_symbol_()
7   {
8       add
9       ( "OPTIONS", omd::http::request::REQUEST_OPTIONS )
10      ( "GET", omd::http::request::REQUEST_GET )
11      ( "HEAD", omd::http::request::REQUEST_HEAD )
12      ( "POST", omd::http::request::REQUEST_POST )
13      ( "PUT", omd::http::request::REQUEST_PUT )
14      ( "DELETE", omd::http::request::REQUEST_DELETE )
15      ( "TRACE", omd::http::request::REQUEST_TRACE )
16      ( "CONNECT", omd::http::request::REQUEST_CONNECT )
17      ;
18  }
19
20  } method_symbol;
```

# Adapting the Structures

```
1 BOOST_FUSION_ADAPT_STRUCT(  
2     omd::http::request::request_line_t,  
3     (omd::http::request::method_t, method)  
4     (std::string, uri)  
5     (std::string, version)  
6 )  
7  
8  
9 BOOST_FUSION_ADAPT_STRUCT(  
10    omd::http::request::message,  
11    (omd::http::request::request_line_t, request)  
12    (omd::http::request::message::headers_t, headers)  
13 )  
  
1 qi::rule< Iterator, std::pair<std::string, std::string>() > header_pair;  
2 qi::rule< Iterator, omd::http::request::request_line_t() > request_line;  
3  
4 ...  
5  
6 message =  
7     request_line  
8     >> *header_pair  
9     >> crlf  
10 ;
```

# Outline

## 4 Protocol Translator

- The Problem
- The Solution

## 5 HTTP Request

- The Request
- The URI

## 6 XML

- What is in a name?

# Our Data Structure

```
1 namespace omd{ namespace http{ namespace request{
2
3     struct uri_parts
4     {
5         typedef std::map< std::string, std::string > query_t;
6         boost::optional< std::string > root;
7         boost::optional< std::string > hierarchy;
8         boost::optional< query_t > queries;
9     };
10
11 }}}
```

  

```
1 qi::rule< Iterator, omd::http::request::uri_parts() > start;
2 qi::rule< Iterator, std::pair<std::string, std::string>() > query_pair;
3
4 start =
5     lit( '/' )
6     >> -( +( ~char_( "/" ) ) )
7     >> -( '/' >> +( ~char_( "?" ) ) )
8     >> -( '?' >> ( query_pair % '&' ) )
9     ;
10
11 query_pair = +( ~char_( '=' ) ) >> '=' >> +( ~char_( '&' ) );
```

# Outline

## 4 Protocol Translator

- The Problem
- The Solution

## 5 HTTP Request

- The Request
- The URI

## 6 XML

- What is in a name?

# ?ML Example

## Example

$$\pi \cong 3$$

## True

This is an XML Example.

## Heresy

Let's call it an MXL Example (*Michael's eXchange Language*)

# ?ML Example

## Example

$$\pi \cong 3$$

## True

This is an XML Example.

## Heresy

Let's call it an MXL Example (*Michael's eXchange Language*)

# ?ML Example

## Example

$$\pi \cong 3$$

## True

This is an XML Example.

## Heresy

Let's call it an MXL Example (*Michael's eXchange Language*)



