Using X3

Using Spirit X3 to Write Parsers

ciere consulting

Michael Caisse

michael.caisse@ciere.com

ciere.com

Part I

Introduction

ciere.com

Outline

1. Introduction
   - Spirit X3
   - Concepts
2. Elements
   - Parsers
   - Rules
   - Grammars
   - Attributes

ciere.com

Spirit X3

- Next generation of Spirit
- Modern C++14 language features
- Hackable, simpler internal design.
- Minimal code base and dependencies
- Compiles faster and runs faster
- Better error handling
- Optimized attribute processing

Spirit X3

- Next generation of Spirit
- Modern C++14 language features
- Hackable, simpler internal design.
- Minimal code base and dependencies
- Compiles faster and runs faster
- Better error handling
- Optimized attribute processing

Spirit X3

- Next generation of Spirit
- Modern C++14 language features
- Hackable, simpler internal design.
- Minimal code base and dependencies
- Compiles faster and runs faster
- Better error handling
- Optimized attribute processing

ciere.com

Spirit X3

- Next generation of Spirit
- Modern C++14 language features
- Hackable, simpler internal design.
- Minimal code base and dependencies
- Compiles faster and runs faster
- Better error handling
- Optimized attribute processing

Spirit X3

- Next generation of Spirit
- Modern C++14 language features
- Hackable, simpler internal design.
- Minimal code base and dependencies
- Compiles faster and runs faster
- Better error handling
- Optimized attribute processing

Spirit X3

- Next generation of Spirit
- Modern C++14 language features
- Hackable, simpler internal design.
- Minimal code base and dependencies
- Compiles faster and runs faster
- Better error handling
- Optimized attribute processing

ciere.com

Spirit X3

- Next generation of Spirit
- Modern C++14 language features
- Hackable, simpler internal design.
- Minimal code base and dependencies
- Compiles faster and runs faster
- Better error handling
- Optimized attribute processing

Outline

1. Introduction
   - Spirit X3
   - Concepts
2. Elements
   - Parsers
   - Rules
   - Grammars
   - Attributes

Spirit X3

Domain Specific Embedded Language

Spirit X3

Domain Specific Embedded Language

Parsing

Spirit X3

Domain Specific **Embedded** Language

C++ via *Expression Templates*

Spirit X3

Domain Specific Embedded **Language**

PEG - Parsing Expression Grammar

Ad-hoc Parsing

```cpp
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
```

Ad-hoc Parsing and Generating

```
boost::regex expression( "(request_firmware_version)|(calibrat
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
```

Concepts

PEG grammar Email (*not really*)

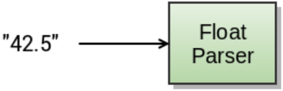
```
name     <-  [a-z]+ ("." [a-z]+)*
host     <-  [a-z]+ "." ("com" / "org" / "net")
email    <-  name "@" host
```

```
auto name  = +char_("a-z") >> *('.' >> +char_("a-z"));
auto host  = +char_("a-z") >> '.' >> (lit("com") | "org" | "net");
auto email = name >> '@' >> host;
```

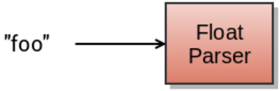
Concepts

- Parsers
- Rules
- Attribute Parsing

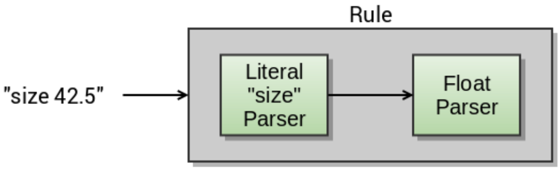
Parsers



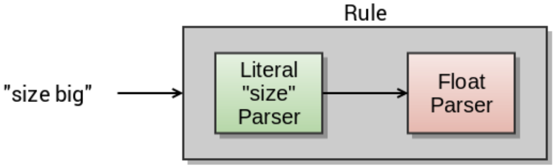
Parsers



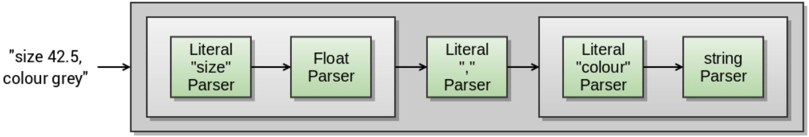
Rules



Rules



Rules



"size 42.5, colour grey"

Literal "size" Parser

Float Parser

Literal "," Parser

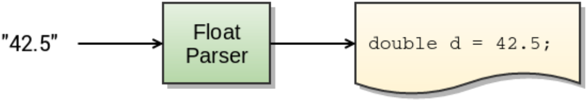
Literal "colour" Parser

string Parser

ciere.com

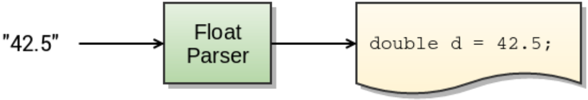
Attributes

Synthesized Attribute



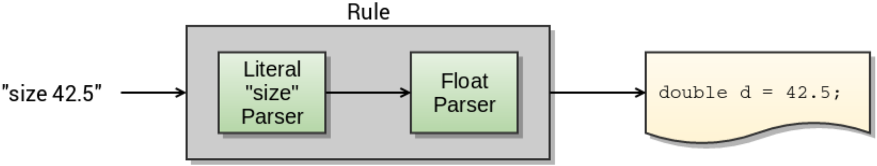
Attributes

Synthesized Attribute



```
double d = 42.5;
```

Attributes



Grammars??

shhhhhh ....

Outline

1. Introduction
   - Spirit X3
   - Concepts
2. Elements
   - Parsers
   - Rules
   - Grammars
   - Attributes

ciere.com

Parser

Data Stream → X3 → 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

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")
```

ciere.com

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 `x3::parse` API.

```
std::string input( "1234" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ );
```

A First, Simple Example

We can use the parser with the `x3::parse` API.

```
std::string input( "1234" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ );
```

A First, Simple Example

We can use the parser with the `x3::parse` API.

```
std::string input( "1234" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ );
```

A First, Simple Example

We can use the parser with the `x3::parse` API.

```
std::string input( "1234" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ );
```

A First, Simple Example

We can use the parser with the `x3::parse` API.

```
std::string input( "1234" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ );
```

A First, Simple Example

Parsing the double in just as simple.

```
std::string input( "1234.56" );

auto iter = input.begin();
auto end_iter = input.end();

x3::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_`, **`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_, 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
character	char_, 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	

ciere.com

Some of the Available Parsers

Type	Parser	Example
character	**char_**, 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	char_, **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	`char_, 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	char_, 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	`char_, 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`	

ciere.com

Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ >> ’ ’ >> double_ );
```

Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ >> ’ ’ >> double_ );
```

Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ >> ' ' >> double_ );
```

Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ >> ’ ’ >> double_ );
```

Sequence of Parsers

Combining parsers allows us to build more complex parsers.

```
std::string input( "876 1234.56" );

auto iter = input.begin();
auto end_iter = input.end();

x3::parse( iter, end_iter,
           int_ >> ' ' >> double_ );
```

Operators

Description	PEG	Spirit X3
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		a - b
Expectation		a > b
List		a % b

Operators

Description	PEG	Spirit X3
Sequence	a b	**a >> b**
Alternative	a | b	a | b
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		a - b
Expectation		a > b
List		a % b

Read as *a* is followed by *b*

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

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

Operators

Description	PEG	Spirit X3
Sequence	a b	a >> b
Alternative	a | b	**a | b**
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		a - b
Expectation		a > b
List		a % b

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

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

Operators

Description	PEG	Spirit X3
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero of more (Kleene)	a*	***a**
One or more (Plus)	a+	**+a**
Optional	a?	**-a**
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		a - b
Expectation		a > b
List		a % b

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

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

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

Operators

Description	PEG	Spirit X3
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	**&a**
Not-predicate	!a	!a
Difference		a - b
Expectation		a > b
List		a % b

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

```
int_ >> &char_(';')
"86;"
"-99" fails to parse
```

Operators

Description	PEG	Spirit X3
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	**!a**
Difference		a - b
Expectation		a > b
List		a % b

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 X3
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		**a - b**
Expectation		a > b
List		a % b

Match *a* but not *b*.

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

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

Always fails.

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

Operators

Description	PEG	Spirit X3
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		a - b
Expectation		**a > b**
List		a % b

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 X3
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero of more (Kleene)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		a - b
Expectation		a > b
List		**a % b**

Shortcut for:

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

```
int_ % ','
```

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

Combining Parsers - Parse key/value pairs

```
std::string input{ "foo    : bar , "
                   "gorp   : smart , "
                   "falcou : \"crazy frenchman\" , "
                   "name   : sam " };

auto iter = input.begin();
auto iter_end = input.end();

phrase_parse( iter, iter_end,
              // ------ start parser -------


              ( name >> ':' >> ( quote | name ) ) % ','




              // ------- end parser --------
              , space );
```

Outline

1. Introduction
   - Spirit X3
   - Concepts
2. Elements
   - Parsers
   - Rules
   - Grammars
   - Attributes

ciere.com

Combining Parsers - Rules

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

- Allows us to name parsers
- Specify the attribute type
- Allows for recursion (the rule may recursively call itself directly or indirectly)
- Provide error handling (on_error)
- Attach custom handlers when a match is found (on_sucess)

Combining Parsers - Rules

Using C++11 auto.

```
auto name  = alpha >> *alnum;

auto quote = '"' >> *( ~char_('"') ) >> '"';
```

Combining Parsers - Rules

Using C++11 auto.

```
auto name  = alpha >> *alnum;

auto quote = '"' >> *( ~char_('"') ) >> '"';
```

Caution

Only use **`auto`** for non-recursive rules.

ciere.com

Combining Parsers - Rules

Using X3 Rules.

```
auto name = x3::rule<class name>{}
          = alpha >> *alnum;


auto quote = x3::rule<class quote>{}
           = '"' >> *( ~char_('"') ) >> '"';
```

Combining Parsers - Rules

Using X3 Rules.

```
auto name = x3::rule<class name>{}
          = alpha >> *alnum;


auto quote = x3::rule<class quote>{}
           = '"' >> *( ~char_('"') ) >> '"';
```

Combining Parsers - Rules

The ID tag to be used by the rule.

```
auto name = x3::rule<class name>{}
          = alpha >> *alnum;

auto quote = x3::rule<class quote>{}
           = '"' >> *( ~char_('"') ) >> '"';
```

Combining Parsers - Parse key/value pairs refined

```
std::string input{ "foo         : bar , "
                   "gorp        : smart , "
                   "falcou      : \"crazy frenchman\" , "
                   "name        : sam " };

auto iter = input.begin();
auto iter_end = input.end();

auto name  = alpha >> *alnum;
auto quote =    '"'
             >> lexeme[ *(~char_('"')) ]
             >> '"'
           ;


phrase_parse(iter, iter_end,

             ( name >> ':' >> (quote | name) ) % ','

             , space);
```

Outline

1. Introduction
   - Spirit X3
   - Concepts
2. Elements
   - Parsers
   - Rules
   - Grammars
   - Attributes

ciere.com

No Grammar in X3

Grammars are not required in X3

Outline

1. Introduction
   - Spirit X3
   - Concepts
2. Elements
   - Parsers
   - Rules
   - Grammars
   - Attributes

ciere.com

Getting Parse Results

How do we get at the parsed results?

```
std::string input{ "foo    : bar , "
                   "gorp   : smart , "
                   "falcou : \"crazy frenchman\" , "
                   "name   : sam " };

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

// Do something clever here ??????????
```

Parsers Expose Attributes - Synthesized Attributes

	X3 Parser Type	Attribute Type
Literals	’a’, "abc", int_(42), …	No attribute
Primitives	int_, char_, double_, …	int, char, double, …
	bin, oct, hex	unsigned
	string("abc")	"abc"
Non-terminal	rule<Tag, A>	A
Operators	a >> b	tuple<A, B>
	a | b	boost::variant<A,B>
	⋆a	std::vector<A>
	+a	std::vector<A>
	−a	boost::optional<A>
	&a, !a	No attribute
	a % b	std::vector<A>

Parsers Expose Attributes - Synthesized Attributes

	X3 Parser Type	Attribute Type
Literals	`'a'`, **`"abc"`**, `int_(42)`, …	**No attribute**
Primitives	`int_`, `char_`, `double_`, …	`int`, `char`, `double`, …
	`bin`, `oct`, `hex`	`unsigned`
	`string("abc")`	`"abc"`
Non-terminal	`rule<Tag, A>`	`A`
Operators	`a >> b`	`tuple<A, B>`
	`a | b`	`boost::variant<A,B>`
	`*a`	`std::vector<A>`
	`+a`	`std::vector<A>`
	`-a`	`boost::optional<A>`
	`&a, !a`	No attribute
	`a % b`	`std::vector<A>`

Parsers Expose Attributes - Synthesized Attributes

	X3 Parser Type	Attribute Type
Literals	`'a', "abc", int_(42),` ...	No attribute
Primitives	`int_, char_,` **`double_`**, ...	`int, char,` **`double`**, ...
	`bin, oct, hex`	`unsigned`
	`string("abc")`	`"abc"`
Non-terminal	`rule<Tag, A>`	`A`
Operators	`a >> b`	`tuple<A, B>`
	`a | b`	`boost::variant<A,B>`
	`*a`	`std::vector<A>`
	`+a`	`std::vector<A>`
	`-a`	`boost::optional<A>`
	`&a, !a`	No attribute
	`a % b`	`std::vector<A>`

Parsers Expose Attributes - Synthesized Attributes

	X3 Parser Type	Attribute Type
Literals	`'a', "abc", int_(42)`, ...	No attribute
Primitives	`int_, char_, double_`, ...	`int, char, double`, ...
	`bin, oct, hex`	`unsigned`
	`string("abc")`	`"abc"`
Non-terminal	`rule<Tag, A>`	`A`
Operators	`a >> b`	`tuple<A, B>`
	`a | b`	`boost::variant<A,B>`
	`*a`	`std::vector<A>`
	`+a`	`std::vector<A>`
	`-a`	`boost::optional<A>`
	`&a, !a`	No attribute
	`a % b`	`std::vector<A>`

Parsers Expose Attributes - Synthesized Attributes

	X3 Parser Type	Attribute Type
Literals	’a’, "abc", int_(42), ...	No attribute
Primitives	int_, char_, double_, ...	int, char, double, ...
	bin, oct, hex	unsigned
	string("abc")	"abc"
Non-terminal	rule<Tag, A>	A
Operators	**a >> b**	**tuple<A, B>**
	a | b	boost::variant<A,B>
	⋆a	std::vector<A>
	+a	std::vector<A>
	−a	boost::optional<A>
	&a, !a	No attribute
	a % b	std::vector<A>

Parsers Expose Attributes - Synthesized Attributes

	X3 Parser Type	Attribute Type
Literals	'a', "abc", int_(42), …	No attribute
Primitives	int_, char_, double_, …	int, char, double, …
	bin, oct, hex	unsigned
	string("abc")	"abc"
Non-terminal	rule<Tag, A>	A
Operators	a >> b	tuple<A, B>
	a | b	**boost::variant<A,B>**
	*a	std::vector<A>
	+a	std::vector<A>
	-a	boost::optional<A>
	&a, !a	No attribute
	a % b	std::vector<A>

Parsers Expose Attributes - Synthesized Attributes

	X3 Parser Type	Attribute Type
Literals	`'a', "abc", int_(42),` ...	No attribute
Primitives	`int_, char_, double_,` ...	`int, char, double,` ...
	`bin, oct, hex`	`unsigned`
	`string("abc")`	`"abc"`
Non-terminal	`rule<Tag, A>`	`A`
Operators	`a >> b`	`tuple<A, B>`
	`a | b`	`boost::variant<A,B>`
	`*a`	**`std::vector<A>`**
	`+a`	**`std::vector<A>`**
	`-a`	`boost::optional<A>`
	`&a, !a`	No attribute
	`a % b`	**`std::vector<A>`**

A First Attribute Example

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

```
std::string input "1234" ;
auto iter = input.begin();
auto 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" ;
auto iter = input.begin();
auto 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" ;
auto iter = input.begin();
auto 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" };
auto iter = input.begin();
auto 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" };
auto iter = input.begin();
auto 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" };
auto iter = input.begin();
auto 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" };
auto iter = input.begin();
auto 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" };
auto iter = input.begin();
auto 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" };
auto iter = input.begin();
auto 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" );
auto iter = input.begin();
auto 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" );
auto iter = input.begin();
auto 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" );
auto iter = input.begin();
auto 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\" " );

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
    = alpha >> *alnum;
auto quote = rule<class quote, std::string>()
    = ’"’
    >> lexeme[ *(~char_(’"’)) ]
    >> ’"’;

auto item = rule<class item, std::pair<std::string, std::string>>()
    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

The rule’s (synthesized) attribute must be compatible with its (RHS) definition.

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

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
    = alpha >> *alnum;
auto quote = rule<class quote, std::string>()
    = ’"’
    >> lexeme[ *(~char_(’"’)) ]
    >> ’"’;

auto item = rule<class item, std::pair<std::string, std::string>>()
    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> → (a >> b): std::vector<char>

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

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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>** → (a >> b): std::vector<char>

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

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
    = alpha >> *alnum;
auto quote = rule<class quote, std::string>()
    = ’"’
    >> lexeme[ *(~char_(’"’)) ]
    >> ’"’;

auto item = rule<class item, std::pair<std::string, std::string>>()
    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> → (a >> b): **std::vector<char>**

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

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
    = alpha >> *alnum;
auto quote = rule<class quote, std::string>()
    = ’"’
    >> lexeme[ *(~char_(’"’)) ]
    >> ’"’;

auto item = rule<class item, std::pair<std::string, std::string>>()
    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\" " );

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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\" " );

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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\" " );

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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 → (a | b): variant<string, string> → string

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

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
    = alpha >> ⋆alnum;
auto quote = rule<class quote, std::string>()
    = ’"’
    >> lexeme[ ⋆(~char_(’"’)) ]
    >> ’"’;

auto item = rule<class item, std::pair<std::string, std::string>>()
    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** → (a | b): variant<string, string> → string

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

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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 → (a | b): variant<string, string> → **string**

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

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
    = alpha >> *alnum;
auto quote = rule<class quote, std::string>()
    = ’"’
    >> lexeme[ *(~char_(’"’)) ]
    >> ’"’;

auto item = rule<class item, std::pair<std::string, std::string>>()
    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** → (a >> b >> c): tuple<string, string>

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

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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 → (a >> b >> c): tuple<string, string>

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

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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 → (a >> b >> c): **tuple<string, string>**

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

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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\" " );

auto iter = input.begin();
auto iter_end = input.end();

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

auto item = rule<class item, std::pair<std::string, std::string>>()
    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\" " );

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
    = alpha >> ⋆alnum;
auto quote = rule<class quote, std::string>()
    = ’"’
    >> lexeme[ ⋆(~char_(’"’)) ]
    >> ’"’;

auto item = rule<class item, std::pair<std::string, std::string>>()
    name >> ’:’ >> ( quote | name );

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

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

Rule Declarations

The rule's attribute type (optional).

```
auto name = x3::rule<class name, name_attr>{}
          = alpha >> *alnum;

auto quote = x3::rule<class quote, quote_attr>{}
           = '"' >> *( ~char_('"') ) >> '"';
```

Rule Declarations

The rule’s attribute type (optional).

```
auto name = x3::rule<class name, name_attr>{}
          = alpha >> ⋆alnum;

auto quote = x3::rule<class quote, quote_attr>{}
           = ’"’ >> ⋆( ~char_(’"’) ) >> ’"’;
```

Part II

Example

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

ciere.com

Build on Success

- Start small
  - Alternatives are a natural place to build
  - Leaves up
- Compose and test
- Test early and often
- Parsing first, Attributes second
- Allow the natural AST to fall out
- Refine grammar/AST

ciere.com

Build on Success

- Start small
  - Alternatives are a natural place to build
  - Leaves up
- Compose and test
- Test early and often
- Parsing first, Attributes second
- Allow the natural AST to fall out
- Refine grammar/AST

ciere.com

Build on Success

- Start small
  - Alternatives are a natural place to build
  - Leaves up
- Compose and test
- Test early and often
- Parsing first, Attributes second
- Allow the natural AST to fall out
- Refine grammar/AST

ciere.com

Build on Success

- Start small
  - Alternatives are a natural place to build
  - Leaves up
- Compose and test
- Test early and often
- Parsing first, Attributes second
- Allow the natural AST to fall out
- Refine grammar/AST

Build on Success

- Start small
  - Alternatives are a natural place to build
  - Leaves up
- Compose and test
- Test early and often
- Parsing first, Attributes second
- Allow the natural AST to fall out
- Refine grammar/AST

ciere.com

Build on Success

- Start small
  - Alternatives are a natural place to build
  - Leaves up
- Compose and test
- Test early and often
- Parsing first, Attributes second
- Allow the natural AST to fall out
- Refine grammar/AST

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

JSON Example

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

AST

```
using  string_t         =  std::string;
using  double_t         =  double;
using  float_t          =  double;
using  int_t            =  int64_t;
using  bool_t           =  bool;
struct null_t              {};

class  value;

using  object_t         =  std::map<std::string, value>;
using  object_member_t  =  object_t::value_type;
using  array_t          =  container::stable_vector<value>;
```

AST

```
class value
   : public x3::variant<
                null_t
              , bool_t
              , string_t
              , int_t
              , double_t
              , object_t
              , array_t      >
{

public:
   using value_type = value;

   using base_type::base_type;
   using base_type::operator=;

   value(null_t val = null_t{})     : base_type(val) {}
   value(char const * val)          : base_type((string_t(val))) {}

};
```

AST

```
class value
   : public x3::variant<
                null_t
              , bool_t
              , string_t
              , int_t
              , double_t
              , object_t
              , array_t      >
{

public:
   using value_type = value;

   using base_type::base_type;
   using base_type::operator=;

   value(null_t val = null_t{})     : base_type(val) {}
   value(char const * val)          : base_type((string_t(val))) {}

};
```

AST

```
class value
   : public x3::variant<
                null_t
              , bool_t
              , string_t
              , int_t
              , double_t
              , object_t
              , array_t      >
{



   template< typename T >
   value( T val
        , typename std::enable_if<
                std::is_floating_point<T>::value
          >::type* = 0)
      : base_type( double_t{val} ) {}



};
```

AST

```
class value
   : public x3::variant<
                null_t
              , bool_t
              , string_t
              , int_t
              , double_t
              , object_t
              , array_t      >
{


   template< typename T >
   value( T val
        , typename std::enable_if<
                std::is_floating_point<T>::value
          >::type* = 0)
      : base_type( double_t{val} ) {}


};
```

AST

```
class value
   : public x3::variant<
                null_t
              , bool_t
              , string_t
              , int_t
              , double_t
              , object_t
              , array_t       >
{


   template< typename T >
   value( T val
        , typename std::enable_if<
             std::integral_constant<
                 bool
               ,   std::is_integral<T>::value
                 | std::is_enum<T>::value
              >
          >::type* = 0)
      : base_type( int_t{val} ) {}


};
```

AST

```
class value
   : public x3::variant<
                null_t
              , bool_t
              , string_t
              , int_t
              , double_t
              , object_t
              , array_t      >
{


   template< typename T >
   value( T val
        , typename std::enable_if<
             std::integral_constant<
                 bool
               ,   std::is_integral<T>::value
                 | std::is_enum<T>::value
              >
          >::type* = 0)
      : base_type( int_t{val} ) {}


};
```

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

Grammar Declaration

Using BOOST_SPIRIT_DECLARE

```
namespace ciere { namespace json { namespace parser
{
   namespace x3 = boost::spirit::x3;

   struct json_class;
   using  json_type = x3::rule<json_class, json::value>;

   BOOST_SPIRIT_DECLARE(json_type);

}}}
```

Grammar Declaration

Using BOOST_SPIRIT_DECLARE

```
namespace ciere { namespace json { namespace parser
{
   namespace x3 = boost::spirit::x3;

   struct json_class;
   using  json_type = x3::rule<json_class, json::value>;

   BOOST_SPIRIT_DECLARE(json_type);

}}}
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Grammar Definition

```
namespace ciere { namespace json { namespace parser
{



   struct value_class;
   struct object_class;
   struct member_pair_class;
   struct array_class;



}}}
```

Grammar Definition

```
namespace ciere { namespace json { namespace parser
{


using value_type = x3::rule<value_class, json::value>;

using object_type = x3::rule<object_class, json::object_t>;

using member_pair_type = x3::rule< member_pair_class
                                 , json::object_member_t>;

using array_type = x3::rule<array_class, json::array_t>;


value_type const value = "value";
object_type const object = "object";
member_pair_type const member_pair = "member_pair";
array_type const array = "array";


}}}
```

Rule Definitions

```
namespace ciere { namespace json { namespace parser
{




auto const value_def =
       null_value
     | bool_value
     | detail::unicode_string
     | lexeme[!('+' | (-lit('-') >> '0' >> digit)) >> int_ >> !char_(".eE")]
     | lexeme[!('+' | (-lit('-') >> '0' >> digit)) >> double_]
     | object
     | array
     ;




}}}
```

Rule Definitions

```
namespace ciere { namespace json { namespace parser
{




   auto const null_value =
         lit("null")
      >> attr(json::null_t{})
      ;





}}}
```

Rule Definitions

```
namespace ciere { namespace json { namespace parser
{



   x3::int_parser<int64_t> const int_ = {};
   ascii::bool_type const bool_value = {};




}}}
```

Rule Definitions

```
namespace ciere { namespace json { namespace parser
{



   auto const object_def =
         lit(’{’)
      >> -(member_pair % ’,’)
      >> lit(’}’)
      ;


   auto const member_pair_def =
         detail::unicode_string
      >> ’:’
      >> value
      ;



}}}
```

Rule Definitions

```
namespace ciere { namespace json { namespace parser
{



   auto const object_def =
         lit('{')
      >> -(member_pair % ',')
      >> lit('}')
      ;


   auto const member_pair_def =
         detail::unicode_string
      >> ':'
      >> value
      ;



}}}
```

Rule Definitions

```
namespace ciere { namespace json { namespace parser
{



   auto const array_def =
         lit('[')
      >> -(value % ',')
      >> lit(']')
      ;



}}}
```

Define

```
namespace ciere { namespace json { namespace parser
{



   BOOST_SPIRIT_DEFINE(
       value
     , object
     , member_pair
     , array
   );



}}}
```

Unicode

```
struct unicode_string_class;
using unicide_string_type =
   x3::rule<unicode_string_class, std::string>;
unicode_string_type const unicode_string = "unicode_string";

auto const unicode_string_def = double_quoted;

BOOST_SPIRIT_DEFINE(unicode_string);
```

Unicode

```
auto const char_esc =
   '\\' > escape
   ;

auto const append = [](auto& ctx) { _val(ctx) += _attr(ctx); }

auto const double_quoted =
     lexeme[ '"'
   > *(  char_esc
       | (char_("\x20\x21\x23-\x5b\x5d-\x7e")  )    [append]
      )
   > '"' ]
   ;
```

Unicode

```
auto const char_esc =
   '\\' > escape
   ;
```

```
auto const append = [](auto& ctx) { _val(ctx) += _attr(ctx); }
```

```
auto const double_quoted =
     lexeme[ '"'
   > *(  char_esc
       | (char_("\x20\x21\x23-\x5b\x5d-\x7e")  )    [append]
      )
   > '"' ]
   ;
```

Unicode

```
auto const char_esc =
   '\\' > escape
   ;

auto const append = [](auto& ctx) { _val(ctx) += _attr(ctx); }

auto const double_quoted =
     lexeme[ '"'
   > *(  char_esc
       | (char_("\x20\x21\x23-\x5b\x5d-\x7e")  )    [append]
      )
   > '"' ]
   ;
```

Unicode

```
uint_parser<uchar, 16, 4, 4> const hex4 = {};

auto const escape =
       (’u’ > hex4)           [push_utf8]
   |   char_("\"\\/bfnrt")   [push_esc]
   ;
```

Unicode

```
auto push_esc = [](auto& ctx)
{
   auto& utf8 = _val(ctx);
   switch (_attr(ctx))
   {
      case '"': utf8 += '"';          break;
      case '\\': utf8 += '\\';        break;
      case '/': utf8 += '/';          break;
      case 'b': utf8 += '\b';         break;
      case 'f': utf8 += '\f';         break;
      case 'n': utf8 += '\n';         break;
      case 'r': utf8 += '\r';         break;
      case 't': utf8 += '\t';         break;
   }
};
```

Unicode

```
auto push_utf8 = [](auto& ctx)
{
   typedef std::back_insert_iterator<std::string> insert_iter;
   insert_iter out_iter(_val(ctx));
   boost::utf8_output_iterator<insert_iter> utf8_iter(out_iter);
   *utf8_iter++ = _attr(ctx);
};
```

Invoke the Parse

```
auto const grammar = ciere::json::parser::value;
try
{
   parse_success = spirit::x3::phrase_parse( iter, iter_end
                                           , grammar
                                           , x3::ascii::space_type{}
                                           , v );
}
catch(spirit::x3::expectation_failure<Iterator> const &){}
```

Thank you

Boost.Spirit X3

Find it in current Boost releases

You can find us on freenode : **`##spirit`**

The CiereLabs JSON library is in github:

`https://github.com/cierelabs/json_spirit`

See C++Now 2015 X3 Workshop Slides/Video for more X3 tutorials.

Outline

3. Grammars from Scratch
   - Grammars from Scratch
4. JSON Parser
   - What is JSON
   - AST
   - Grammars
5. Fun with X3
   - X3 Fun
   - ASTs
   - Grammars
   - Error Handling
6. Attributes
   - AST Traversal

x3_fun

A calculator example supporting functions.

x3_fun

Input:

```
(123 + 456) * 789
```

Output:

```
456831
```

x3_fun

Input:

```
sin(45 * (pi / 180))
```

Output:

```
0.707
```

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

ciere.com

AST

```
struct nil {};
struct signed_;
struct expression;
struct function_call;

struct operand :
    x3::variant<
        nil
      , double
      , x3::forward_ast<signed_>
      , x3::forward_ast<expression>
      , x3::forward_ast<function_call>
    >
{
    using base_type::base_type;
    using base_type::operator=;
};
```

AST

```
struct signed_
{
    char sign;
    operand operand_;
};

struct operation : x3::position_tagged
{
    char operator_;
    operand operand_;
};

struct expression : x3::position_tagged
{
    operand first;
    std::list<operation> rest;
};

struct function_call : x3::position_tagged
{
    std::string name;
    std::list<expression> arguments;
};
```

Fusion Adaptation (ast_adapted.hpp)

```
BOOST_FUSION_ADAPT_STRUCT(
    fun::ast::signed_,
    (char, sign)
    (fun::ast::operand, operand_)
)

BOOST_FUSION_ADAPT_STRUCT(
    fun::ast::operation,
    (char, operator_)
    (fun::ast::operand, operand_)
)

BOOST_FUSION_ADAPT_STRUCT(
    fun::ast::expression,
    (fun::ast::operand, first)
    (std::list<fun::ast::operation>, rest)
)

BOOST_FUSION_ADAPT_STRUCT(
    fun::ast::function_call,
    (std::string, name)
    (std::list<fun::ast::expression>, arguments)
)
```

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

Simple Grammars

Using BOOST_SPIRIT_DEFINE

```
using x3::raw;
using x3::lexeme;
using x3::alpha;
using x3::alnum;

struct identifier_class;
typedef
    x3::rule<identifier_class, std::string>
identifier_type;
identifier_type const identifier = "identifier";

auto const identifier_def
    = raw[lexeme[(alpha | ’_’) >> *(alnum | ’_’)]];

BOOST_SPIRIT_DEFINE(identifier);
```

Simple Grammars

Using BOOST_SPIRIT_DEFINE

```
struct identifier_class;

typedef
    x3::rule<identifier_class, std::string>
identifier_type;
identifier_type const identifier = "identifier";

auto const identifier_def
    = raw[lexeme[(alpha | ’_’) >> ⋆(alnum | ’_’)]];

BOOST_SPIRIT_DEFINE(identifier);
```

Simple Grammars

Using BOOST_SPIRIT_DEFINE

```
struct identifier_class;

typedef
    x3::rule<identifier_class, std::string>
identifier_type;
identifier_type const identifier = "identifier";

auto const identifier_def
    = raw[lexeme[(alpha | ’_’) >> *(alnum | ’_’)]];

BOOST_SPIRIT_DEFINE(identifier);
```

ciere.com

Simple Grammars

Using BOOST_SPIRIT_DEFINE

```
struct identifier_class;

typedef
    x3::rule<identifier_class, std::string>
identifier_type;
identifier_type const identifier = "identifier";

auto const identifier_def
    = raw[lexeme[(alpha | ’_’) >> *(alnum | ’_’)]];

BOOST_SPIRIT_DEFINE(identifier);
```

Simple Grammars

Using BOOST_SPIRIT_DEFINE

```
struct identifier_class;

typedef
    x3::rule<identifier_class, std::string>
identifier_type;
identifier_type const identifier = "identifier";

auto const identifier_def
    = raw[lexeme[(alpha | ’_’) >> *(alnum | ’_’)]];

BOOST_SPIRIT_DEFINE(identifier);
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

ciere.com

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Rule Naming Convention

Example (The Rule ID)

```
identifier_class
```

Example (The Rule Type)

```
identifier_type
```

Example (The Rule Definition)

```
identifier_def
```

Example (The Rule)

```
identifier
```

Declaring a Grammar

Using BOOST_SPIRIT_DECLARE

```
namespace parser
{
    struct expression_class;
    typedef
        x3::rule<expression_class, ast::expression>
    expression_type;
    BOOST_SPIRIT_DECLARE(expression_type);
}

parser::expression_type const& expression();
```

Defining a Grammar

```
struct additive_expr_class;
struct multiplicative_expr_class;
struct unary_expr_class;
struct primary_expr_class;
struct argument_list_class;
struct function_call_class;
```

Defining a Grammar

```
typedef x3::rule<additive_expr_class, ast::expression>
additive_expr_type;

typedef
    x3::rule<multiplicative_expr_class, ast::expression>
multiplicative_expr_type;

typedef
    x3::rule<unary_expr_class, ast::operand>
unary_expr_type;

typedef
    x3::rule<primary_expr_class, ast::operand>
primary_expr_type;

typedef
    x3::rule<argument_list_class, std::list<ast::expression>>
argument_list_type;

typedef
    x3::rule<function_call_class, ast::function_call>
function_call_type;
```

Defining a Grammar

```
expression_type const
    expression = "expression";

additive_expr_type const
    additive_expr = "additive_expr";

multiplicative_expr_type const
    multiplicative_expr = "multiplicative_expr";

unary_expr_type const
    unary_expr = "unary_expr";

primary_expr_type const
    primary_expr = "primary_expr";

argument_list_type const
    argument_list = "argument_list";

function_call_type const
    function_call = "function_call";
```

Defining a Grammar

```
auto const additive_expr_def =
    multiplicative_expr
    >> *(   (char_('+') > multiplicative_expr)
        |   (char_('-') > multiplicative_expr)
        )
    ;

auto const multiplicative_expr_def =
    unary_expr
    >> *(   (char_('*') > unary_expr)
        |   (char_('/') > unary_expr)
        )
    ;

auto const unary_expr_def =
        primary_expr
    |   (char_('-') > primary_expr)
    |   (char_('+') > primary_expr)
    ;
```

Defining a Grammar

```
auto argument_list_def = expression % ',';

auto function_call_def =
        identifier
    >> -('(' > argument_list > ')')
    ;

auto const primary_expr_def =
        double_
    |   function_call
    |   '(' > expression > ')'
    ;

auto const expression_def = additive_expr;
```

Defining a Grammar

```
BOOST_SPIRIT_DEFINE(
    expression
  , additive_expr
  , multiplicative_expr
  , unary_expr
  , primary_expr
  , argument_list
  , function_call
);
```

Defining a Grammar

Decorators: Annotations and Error Handlers

```
struct unary_expr_class : annotation_base {};
struct primary_expr_class : annotation_base {};
struct function_call_class : annotation_base {};

struct expression_class :
    annotation_base, error_handler_base {};
```

Defining a Grammar

```
namespace fun
{
    parser::expression_type const& expression()
    {
        return parser::expression;
    }
}
```

Instantiating a Grammar

```
// Our Iterator Type
typedef std::string::const_iterator iterator_type;

// The Phrase Parse Context
typedef
    x3::phrase_parse_context<x3::ascii::space_type>::type
phrase_context_type;

// Our Error Handler
typedef error_handler<iterator_type> error_handler_type;

// Combined Error Handler and Phrase Parse Context
typedef x3::with_context<
    error_handler_tag
  , std::reference_wrapper<error_handler_type> const
  , phrase_context_type>::type
context_type;
```

Instantiating a Grammar

```
namespace fun { namespace parser
{
    BOOST_SPIRIT_INSTANTIATE(
        expression_type, iterator_type, context_type);
}}
```

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

Error Handling

Expectation Operator

```
auto const additive_expr_def =
    multiplicative_expr
    >> *(    (char_('+') > multiplicative_expr)
        |    (char_('-') > multiplicative_expr)
        )
    ;
```

Error Handling

Expectation Operator

```
auto const additive_expr_def =
    multiplicative_expr
    >> *(    (char_('+') > multiplicative_expr)
        |    (char_('-') > multiplicative_expr)
        )
    ;
```

Error Handling

Expect Directive

```
auto const additive_expr_def =
    multiplicative_expr
    >> *(   (char_('+') >> expect[multiplicative_expr])
        |   (char_('-') >> expect[multiplicative_expr])
        )
    ;
```

Error Handling

Expectation Failure

```
template <typename Iterator>
struct expectation_failure : std::runtime_error
{
public:

    expectation_failure(Iterator where, std::string const& which);
    ~expectation_failure() throw();

    std::string which() const;
    Iterator const& where() const;

    /*...*/
};
```

Defining a Grammar

Decorators: Annotations and Error Handlers

```
struct unary_expr_class : annotation_base {};
struct primary_expr_class : annotation_base {};
struct function_call_class : annotation_base {};

struct expression_class :
    annotation_base, error_handler_base {};
```

Error Handling

Error Handler

```
// X3 Error Handler Utility
template <typename Iterator>
using error_handler = x3::error_handler<Iterator>;

// tag used to get our error handler from the context
struct error_handler_tag;

struct error_handler_base
{
    error_handler_base();

    template <typename Iterator, typename Exception, typename Context>
    x3::error_handler_result on_error(
        Iterator& first, Iterator const& last
      , Exception const& x, Context const& context);

    std::map<std::string, std::string> id_map;
};
```

Error Handling

error_handler_base::on_error

```
template <typename Iterator, typename Exception, typename Context>
inline x3::error_handler_result
error_handler_base::on_error(
    Iterator& first, Iterator const& last
  , Exception const& x, Context const& context)
{
    std::string which = x.which();
    auto iter = id_map.find(which);
    if (iter != id_map.end())
        which = iter->second;

    std::string message = "Error! Expecting: " + which + " here:";
    auto& error_handler = x3::get<error_handler_tag>(context).get();
    error_handler(x.where(), message);
    return x3::error_handler_result::fail;
}
```

Error Handling

error_handler_base constructor

```
inline error_handler_base::error_handler_base()
{
    id_map["expression"] = "Expression";
    id_map["additive_expr"] = "Expression";
    id_map["multiplicative_expr"] = "Expression";
    id_map["unary_expr"] = "Expression";
    id_map["primary_expr"] = "Expression";
    id_map["argument_list"] = "Argument List";
}
```

Annotations

Annotating the AST with the iterator position

```
struct annotation_base
{
    template <typename Iterator, typename Context>
    void on_success(Iterator const& first, Iterator const& last
      , ast::operand& ast, Context const& context);

    template <typename T, typename Iterator, typename Context>
    inline void on_success(Iterator const& first, Iterator const& last
      , T& ast, Context const& context);
};
```

Annotations

annotation_base::on_success

```cpp
template <typename T, typename Iterator, typename Context>
inline void
annotation_base::on_success(Iterator const& first, Iterator const& last
  , T& ast, Context const& context)
{
    auto& error_handler = x3::get<error_handler_tag>(context).get();
    error_handler.tag(ast, first, last);
}
```

Annotations

annotation_base::on_success

```
template <typename Iterator, typename Context>
inline void
annotation_base::on_success(Iterator const& first, Iterator co
  , ast::operand& ast, Context const& context)
{
    auto& error_handler
        = x3::get<error_handler_tag>(context).get();

    auto annotate = [&](auto& node)
    {
        error_handler.tag(node, first, last);
    };

    ast.apply_visitor(
        x3::make_lambda_visitor<void>(annotate));
}
```

Error Handling

Bad Syntax

```
foo(123, $%)
```

Error Message

```
In file bad_arguments.fun, line 1:
Error! Expecting: ')' here:
foo(123, $%)
_______^_
```

Outline

3 Grammars from Scratch
- Grammars from Scratch

4 JSON Parser
- What is JSON
- AST
- Grammars

5 Fun with X3
- X3 Fun
- ASTs
- Grammars
- Error Handling

6 Attributes
- AST Traversal

Attribute Parsing vs Semantic Actions

Avoid semantic actions! Generate ASTs instead.

- Imperative semantic actions are ugly warts in an elegant declarative grammar.
- Semantic actions look even uglier and verbose in X3 with native C++ lambda.
- Use semantic actions only to facilitate the generation of an attribute.
- If you really can't avoid semantic actions, at least make them side-effect free. Back tracking can cause havoc when actions are called multiple times.

Attribute Parsing vs Semantic Actions

Avoid semantic actions! Generate ASTs instead.

- Imperative semantic actions are ugly warts in an elegant declarative grammar.
- Semantic actions look even uglier and verbose in X3 with native C++ lambda.
- Use semantic actions only to facilitate the generation of an attribute.
- If you really can’t avoid semantic actions, at least make them side-effect free. Back tracking can cause havoc when actions are called multiple times.

Attribute Parsing vs Semantic Actions

Avoid semantic actions! Generate ASTs instead.

- Imperative semantic actions are ugly warts in an elegant declarative grammar.
- Semantic actions look even uglier and verbose in X3 with native C++ lambda.
- Use semantic actions only to facilitate the generation of an attribute.
- If you really can't avoid semantic actions, at least make them side-effect free. Back tracking can cause havoc when actions are called multiple times.

Attribute Parsing vs Semantic Actions

Avoid semantic actions! Generate ASTs instead.

- Imperative semantic actions are ugly warts in an elegant declarative grammar.
- Semantic actions look even uglier and verbose in X3 with native C++ lambda.
- Use semantic actions only to facilitate the generation of an attribute.
- If you really can't avoid semantic actions, at least make them side-effect free. Back tracking can cause havoc when actions are called multiple times.

Attribute Parsing vs Semantic Actions

Avoid semantic actions! Generate ASTs instead.

- Imperative semantic actions are ugly warts in an elegant declarative grammar.
- Semantic actions look even uglier and verbose in X3 with native C++ lambda.
- Use semantic actions only to facilitate the generation of an attribute.
- If you really can’t avoid semantic actions, at least make them side-effect free. Back tracking can cause havoc when actions are called multiple times.

Attribute Parsing vs Semantic Actions

Avoid semantic actions! Generate ASTs instead.

- Imperative semantic actions are ugly warts in an elegant declarative grammar.
- Semantic actions look even uglier and verbose in X3 with native C++ lambda.
- Use semantic actions only to facilitate the generation of an attribute.
- If you really can't avoid semantic actions, at least make them side-effect free. Back tracking can cause havoc when actions are called multiple times.

Printing the AST

```
struct printer
{
    typedef void result_type;

    printer(std::ostream& out)
        : out(out)
    {}

    void operator()(ast::nil) const { BOOST_ASSERT(0); }
    void operator()(double ast) const;
    void operator()(ast::operation const& ast) const;
    void operator()(ast::signed_ const& ast) const;
    void operator()(ast::expression const& ast) const;
    void operator()(ast::function_call const& ast) const;

    std::ostream& out;
};
```

Printing the AST

```
void printer::operator()(double ast) const
{
    out << ast;
}

void printer::operator()(ast::operation const& ast) const
{
    switch (ast.operator_)
    {
        case '+': out << " + "; break;
        case '-': out << " - "; break;
        case '*': out << " * "; break;
        case '/': out << " / "; break;

        default:
            BOOST_ASSERT(0);
            return;
    }
    boost::apply_visitor(*this, ast.operand_);
}
```

Printing the AST

```
void printer::operator()(ast::expression const& ast) const
{
    if (ast.rest.size())
        out << ’(’;
    boost::apply_visitor(*this, ast.first);
    for (auto const& oper : ast.rest)
        (*this)(oper);
    if (ast.rest.size())
        out << ’)’;
}
```

Printing the AST

```
void printer::operator()(ast::function_call const& ast) const
{
    out << ast.name;
    if (ast.arguments.size())
        out << '(';
    bool first = true;
    for (auto const& arg : ast.arguments)
    {
        if (first)
            first = false;
        else
            out << ", ";
        (*this)(arg);
    }
    if (ast.arguments.size())
        out << ')';
}
```

The Interpreter

```
class interpreter
{
public:

    typedef std::function<
        void(x3::position_tagged, std::string const&)>
    error_handler_type;

    template <typename ErrorHandler>
    interpreter(ErrorHandler const& error_handler);

    template <typename F>
    void add_function(std::string name, F f);

    float eval(ast::expression const& ast);

private:

    std::map<
        std::string
      , std::pair<std::function<double(double* args)>, std::size_t>
    >
    fmap;

    error_handler_type error_handler;
};
```

The Interpreter

```
// Add some functions:
interp.add_function("pi", []{ return M_PI; });
interp.add_function("sin", [](double x){ return std::sin(x); });
interp.add_function("cos", [](double x){ return std::cos(x); });
```

The Interpreter

```
sin(45 * (pi / 180))
```

The Interpreter

```
template <typename F>
inline void interpreter::add_function(std::string name, F f)
{
    static_assert(detail::arity<F>::value <= detail::max_arity,
        "Function F has too many arguments (maximum == 5).");

    std::function<double(double* args)> f_adapter = detail::adapter_function<F>(f);
    fmap[name] = std::make_pair(f_adapter, detail::arity<F>::value);
}
```

The Interpreter

```
double interpreter_impl::operator()(double lhs, ast::operation const& ast) const
{
    double rhs = boost::apply_visitor(*this, ast.operand_);
    switch (ast.operator_)
    {
        case '+': return lhs + rhs;
        case '-': return lhs - rhs;
        case '*': return lhs * rhs;
        case '/': return lhs / rhs;

        default:
           BOOST_ASSERT(0);
           return -1;
    }
}
```

The Interpreter

```
double interpreter_impl::operator()(ast::function_call const& ast) const
{
    auto iter = fmap.find(ast.name);
    if (iter == fmap.end()) {
        error_handler(ast, "Undefined function " + ast.name + '.');
        return -1;
    }

    if (iter->second.second != ast.arguments.size()) {
        std::stringstream out;
        out << "Wrong number of arguments to function " << ast.name << " ("
            << iter->second.second << " expected)." << std::endl;

        error_handler(ast, out.str());
        return -1;
    }

    // Get the args
    double args[detail::max_arity];
    double* p = args;
    for (auto const& arg : ast.arguments)
        *p++ = (*this)(arg);

    // call user function
    return iter->second.first(args);
}
```