A Framework for RAD Spirit

Programs = Algorithms + Data Structures

Joel de Guzman (joel@boostpro.com)
Hartmut Kaiser (hkaiser@cct.lsu.edu)

Outline

- What's Boost.Spirit? A short introduction
  - Qi and Karma: The Yin and Yang of Parsing Input and Generating Output
- Scheme - the Minimalistic Power
  - The Spirit RAD Framework
  - Parsing and Generating S-Expressions
  - Scheme Compiler and Interpreter
  - Parsing and Generating Qi
  - Interpreting Qi

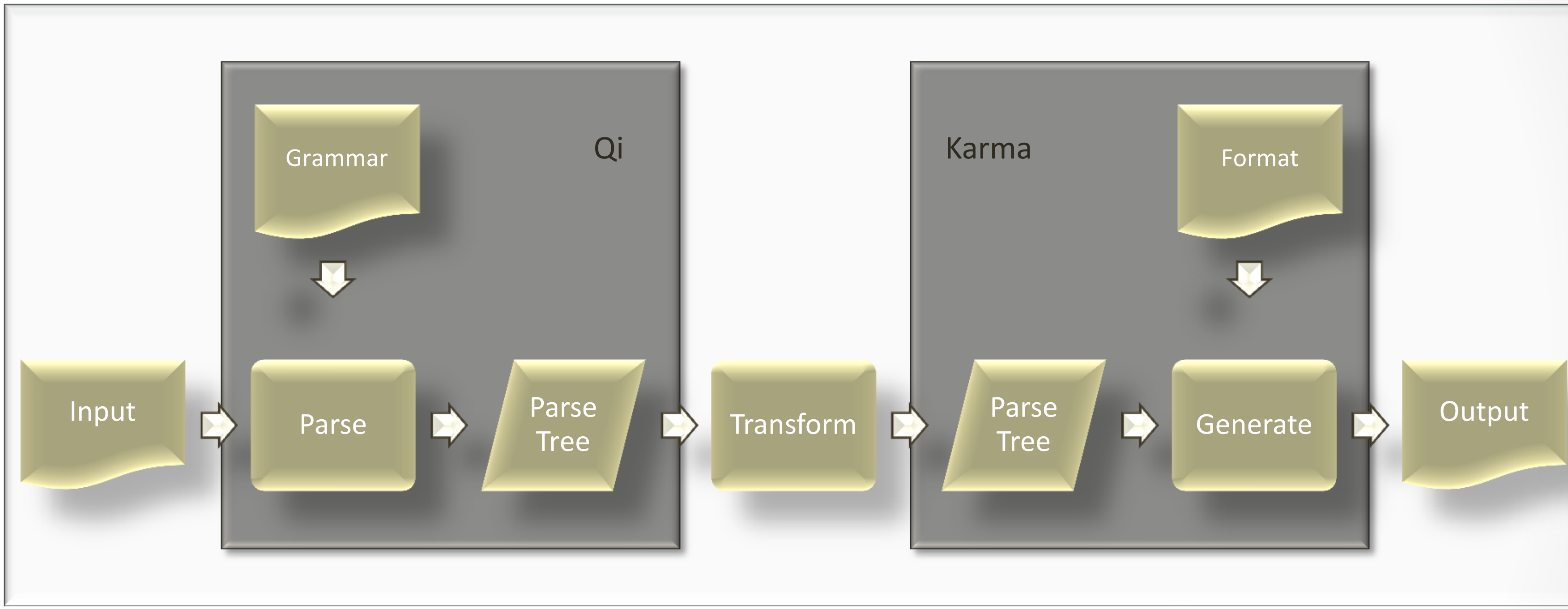
What's Boost.Spirit?

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

Where to get the stuff

- Current version: Spirit V2.3
  - Fully integrated with Boost SVN::trunk, released since V1.40
  - Code for this talk: Boost::SVN needed (or Spirit V2.4, to be released with Boost V1.44)
- Mailing lists:
  - Spirit mailing list: http://sourceforge.net/mail/?group_id=28447
- Web:
  - http://boost-spirit.com/home

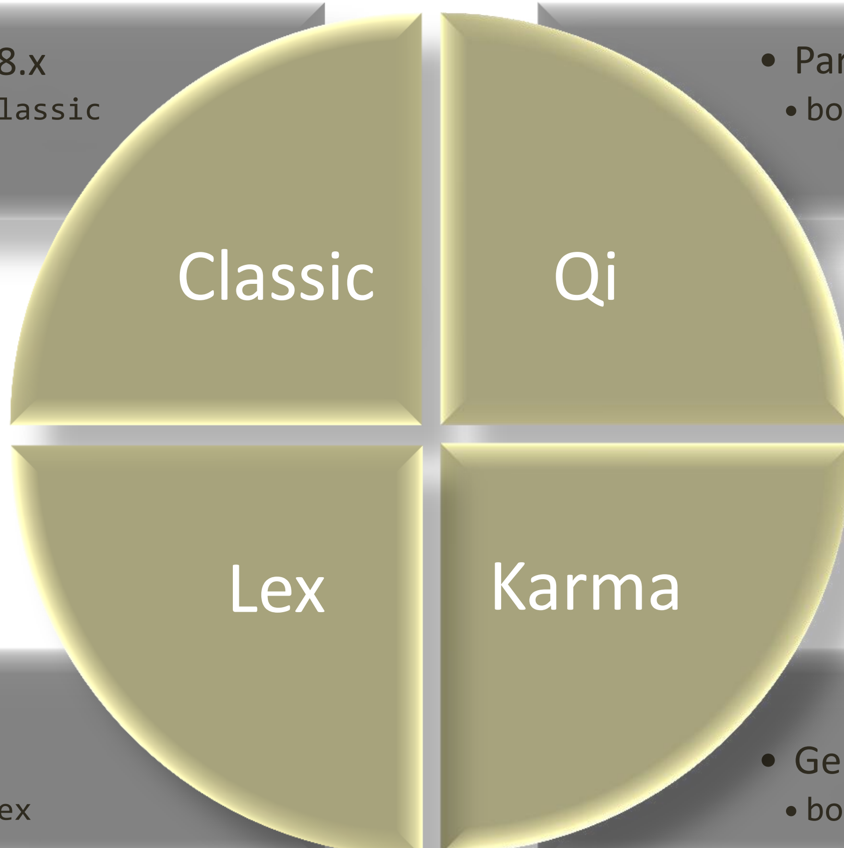
What's Boost.Spirit?



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

Library Structure

- Former Spirit V1.8.x
  - `boost::spirit::classic`
- Parser library
  - `boost::spirit::qi`



- Lexer Library
  - `boost::spirit::lex`
- Generator Library
  - `boost::spirit::karma`

Spirit's Components

- Spirit Classic (`spirit::classic`)
- Create lexical analyzers (`spirit::lex`)
  - Token definition (patterns, values, lexer states)
  - Semantic actions, i.e. attach code to matched tokens
- Parsing Input (`spirit::qi`)
  - Grammar specification
    - Token sequence definition
    - Semantic actions, i.e. attaching code to matched sequences
    - Parsing Expression Grammar (PEG)
    - Error handling
- Generating Output (`spirit::karma`)
  - Format specification
    - Token sequence definition
    - Semantic actions, i.e. attaching code to sequences
    - Inverse Parsing Expression Grammars (IPEG)
  - Formatting directives
    - Alignment, whitespace delimiting, line wrapping, indentation

Qi and Karma

THE YIN AND YANG OF PARSING INPUT AND GENERATING OUTPUT

Parsing Expression Grammars

- Formal grammar for describing a formal language in terms of a set of rules used to recognize strings of this language
- Does not require a tokenization stage
  - But it doesn't prevent it
- Similar to regular expressions being added to the Extended Backus-Naur Form (EBNF)
- Unlike (E)BNF, PEG's are not ambiguous
  - Exactly one valid parse tree for each PEG
- Any PEG can be directly represented as a recursive-descent parser
- Different Interpretation than EBNF
  - Greedy Loops
  - First come first serve alternates

Parsing Input

- Qi is a library allowing to flexibly parse input based on a given grammar (PEG)
  - ‚Parser generator‘, in the vein of yacc, bison, etc.
  - Currently generates recursive descent parsers, which perfectly map onto PEG grammars
    - A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures, each representing one of the grammar elements
    - Thus the structure of the resulting program closely mirrors that of the grammar it recognizes
    - Elements: Terminals (primitives, i.e. plain characters, integer, etc.), non-terminals, sequences, alternatives, modifiers (Kleene, plus, etc.)
- Qi defines a DSEL (domain specific embedded language) hosted directly in C++
  - Using operator overloading, expression templates and template meta-programming
- Inline grammar specifications can mix freely with other C++ code, allowing excellent integration of your data types

Infix Calculator Grammar

Using Parsing Expression Grammars:

```
fact  ←  integer / '('    expr    ')'
term  ←  fact     (('*'    fact) / ('/'    fact))*
expr  ←  term     (('+'    term) / ('-'    term))*
```

Infix Calculator Grammar

Using Qi:

```
using namespace boost::spirit;
typedef qi::rule<std::string::iterator> rule;
rule fact, term, expr;

fact  =  int_    | '(' >> expr >> ')' ;
term  =  fact >> *(('*' >> fact) | ('/' >> fact)) ;
expr  =  term >> *(('+' >> term) | ('-' >> term)) ;
```

Generating Output

- Karma is a library allowing to flexibly generate arbitrary character (byte) sequences
  - Based on the idea, that a grammar usable to parse an input sequence may as well be used to generate the very same sequence in the output
  - For parsing of some input most programmers use hand written code or parser generator tools
  - Need similar tools: 'unparser generators'

- Karma is such a tool
  - Inspired by the StringTemplate library (ANTLR)
  - Allows strict model-view separation (Separation of format and data)
  - Defines a DSEL (domain specific embedded language) allowing to specify the structure of the output to generate in a language derived from PEG

RPN Expression Format

Using Inverse Parsing Expression Grammars:

```
ast_node  → integer / bin_node / u_node
bin_node  → ast_node    ast_node    bin_code
u_node    → '('    ast_node    u_code    ')'
bin_code  →      '+'  / '-' / '*' / '/'
u_code    →      '+'  / '-'
```

RPN Expression Format

Using Karma:

```
using namespace boost::spirit;
typedef karma::rule<output_iterator> rule;
rule ast_node, bin_node, u_node, bin_code, u_code;

ast_node  =  int_    | bin_node | u_node;
bin_node  =  ast_node << ast_node << bin_code;
u_node    =  '(' << ast_node << u_code << ')';
bin_code  =  lit('+') | '-' | '*' | '/';
u_code    =  lit('+') | '-';
```

Spirit versus PEG Operators

Description	PEG	Spirit
Sequence	a b	Qi: a >> b Karma: a << b
Alternative	a / b	a | b
Zero or more (Kleene)	a*	*a
One or more	a+	+a
And-predicate	&a	&a
Not-predicate	!a	!a
Optional	a?	-a

More Spirit Operators

Description	Syntax
Sequential-or (non-shortcutting, Qi only)	a || b
List	a % b
Permutation (Qi only)	a ^ b
Expect (Qi only)	a > b
Semantic Action	a[f]
Character set negation (char_ only)	~a

More about Parsers and Generators

- Currently recursive-descent implementation
  - Other schemes are possible, but not yet implemented
- Spirit makes the compiler generate format driven parser and generator routines
  - The C++ expression is expressed as a Proto type (representing the expression tree) at compile time
    - Achieved by ‚tainting' the C++ expression by using Proto placeholders, which selects the proper overloaded Proto operators
  - The expression tree is converted into a corresponding parser/generator execution tree at runtime
- Parsers and generators are fully attributed
  - Each component either provides or expects a value of a specific type
  - Usual compatibility (convertibility) rules apply

Parser Types and their Attributes

	Qi Parser Types	Attribute Type
Literals	• 'a', "abc", lit(1.0)	• No attribute
Primitive components	• int_, char_, double_, bin, oct, hex • byte, word, dword, qword, … • stream • symbol<A>	• int, char, double • uint8_t, uint16_t, uint32_t, int64_t, … • boost::any • Explicitly specified (A)
Non-terminals	• rule<A()>, grammar<A()>	• Explicitly specified (A)
Operators	• *a (Kleene) • +a (one or more) • -a (optional) • a % b (list) • a >> b (sequence) • a | b (alternative) • &a, !a (predicates/eps) • a ^ b (permutation)	• std::vector<A> (std container) • std::vector<A> (std container) • boost::optional<A> • std::vector<A> (std container) • fusion::vector<A, B> (Fusion sequence) • boost::variant<A, B> • No attribute • fusion::vector< optional<A>, optional<B> >
Directives	• lexeme[a], omit[a], nocase[a] • raw[]	• A • boost::iterator_range<Iterator>
Semantic action	• a[f]	• A

Generator Types and their Attributes

	Karma Generator Types	Attribute Type
Literals	• 'a', "abc", double_(1.0)	• No attribute
Primitive components	• int_, char_, double_, bin, oct, hex • byte, word, dword, qword, … • stream	• int, char, double • uint8_t, uint16_t, uint32_t, uint64_t • boost::any
Non-terminals	• rule<A()>, grammar<A()>	• Explicitly specified (A)
Operators	• *a (Kleene) • +a (one or more) • -a (optional) • a % b (list) • a << b (sequence) • a | b (alternative) • &a, !a (predicates/eps)	• std::vector<A> (std container) • std::vector<A> (std container) • boost::optional<A> • std::vector<A> (std container) • fusion::vector<A, B> (Fusion sequence) • boost::variant<A, B> • A
Directives	• verbatim[a], delimit(…)[a] • lower[a], upper[a] • left_align[a], center[a], right_align[a]	• A • A • A
Semantic action	• a[f]	• A

Attribute Propagation

- Primitive components expose specific attribute type
  - `int_` → `int`, `double_` → `double`, `char_` → `char`
  - Normal C++ convertibility rules apply
    - Qi: any C++ type may receive the parsed value as long as the attribute type of the parser is convertible to the type provided
    - Karma: any C++ type may be consumed as long as it is convertible to the attribute type of the generator
- Compound components implement specific propagation rules
  - `a: A, b: B` → `(a >> b): tuple<A, B>`
  - Given a and b are components, and A is the attribute type of a, and B is the attribute type of b, then the attribute type of a >> b will be tuple<A, B> (any Fusion sequence of A and B).
- Some compound components implement additional compatibility rules
  - `a: A, b: A` → `(a >> b): vector<A>`
- In order for a type to be compatible with the attribute type of a compound expression it has to
  - Either be convertible to the attribute type,
  - Or it has to expose certain functionalities, i.e. it needs to conform to a concept compatible with the component.

Comparison Qi/Karma

	Qi	Karma
Main component	parser	generator
Main routines	parse(), match()	generate(), format()
Primitive components	• int_, char_, double_, … • bin, oct, hex • byte, word, dword, qword, … • stream	• int_, char_, double_, … • bin, oct, hex • byte, word, dword, qword, pad, … • stream
Non-terminals	• rule, grammar	• rule, grammar
Operators	• * (Kleene) • + (one or more) • - (optional) • % (list) • >> (sequence) • | (alternative) • &, ! (predicates/eps)	• * (Kleene) • + (one or more) • - (optional) • % (list) • << (sequence) • | (alternative) • &, ! (predicates/eps)
Directives	• lexeme[], skip[], omit[], raw[] • nocase[]	• verbatim[], delimit[] • left_align[], center[], right_align[] • upper[], lower[]

Comparison Qi/Karma

	Qi	Karma

Semantic Action	receives value
	`int_ [ ref(i) = _1 ]`
	`(char_ >> int_)`
	`   [ref(c) = _1, ref(i) = _2]`
		provides value
		`int_ [ _1 = ref(i) ]`
		`(char_ << int_)`
		`   [_1 = ref(c), _2 = ref(i)]`

Attributes	• Attribute of a parser component (‚return type‘) is the type of the value it generates, it must be convertible to the target type. • Attributes are propagated up. • Attributes are passed as non-const& • Parser components may not have target attribute value	• The attribute of a generator component is the type of the values it expects, i.e. the provided value must be convertible to this type. • Attributes are passed down. • Attributes are passed as const& • Generator components need always a ‚source‘ value: either literal or parameter

In traditional Chinese culture, Qi (氣) is an active principle forming part of any living thing. It is frequently translated as "energy flow", or "vitalism".

SPIRIT.QI

A LIBRARY FOR PARSING INPUT

Parsing Input

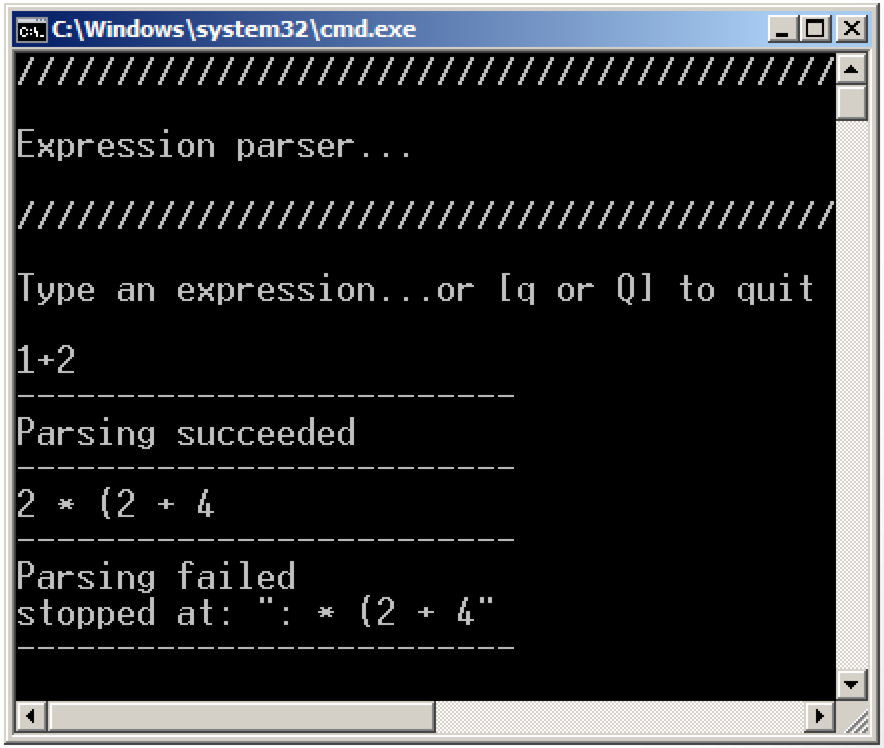
- *Qi* is designed to be a practical parsing tool
- Generates a fully-working parser from a formal EBNF specification inlined in C++
- Regular-expression libraries (such as boost regex) or scanners (such as Boost tokenizer) do not scale well when we need to write more elaborate parsers.
- Attempting to write even a moderately-complex parser using these tools leads to code that is hard to understand and maintain.
- One of *Qi's* prime objectives is to make the parsing easy to use
  - Header only library
- Very fast execution, tight generated code

A Calculator: The Parser

```
template <typename Iterator>
struct calculator
    : grammar<Iterator>
{
    calculator() : calculator::base(expr)
    { /*…definition here*/ }

    rule<Iterator>
        expr, term, factor;
};
```

A Calculator: The Parser

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

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

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



A Calculator: The Interpreter

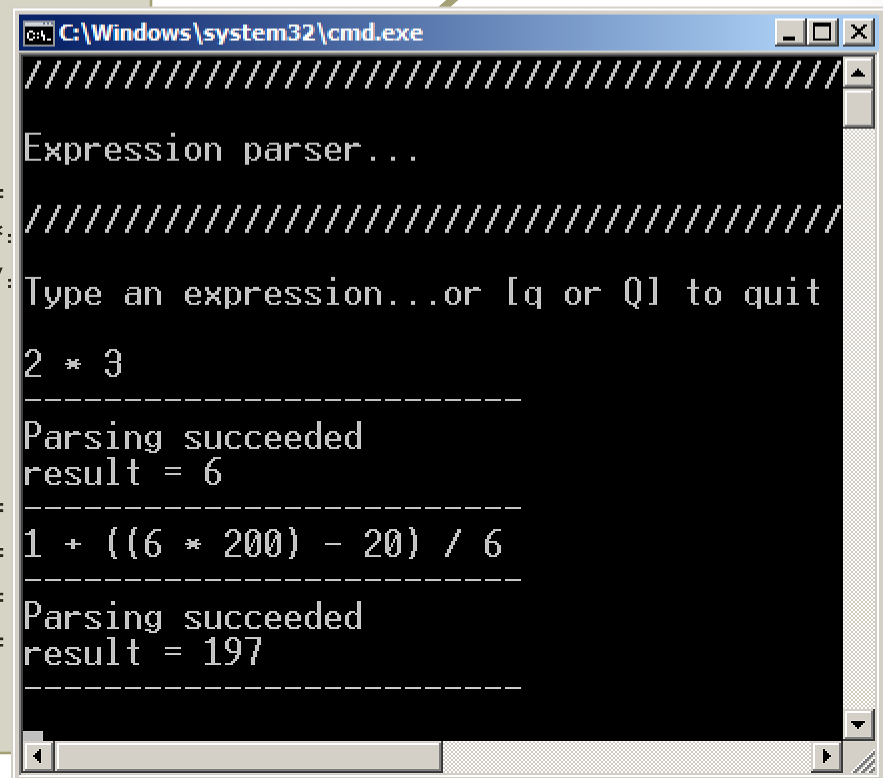
```
template <typename Iterator>
struct calculator
    : grammar<Iterator, int()>
{
    calculator() : calculator::base(expr)
    { /*…definition here*/ }

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

Grammar and Rule Signature

A Calculator: The Interpreter

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

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

factor =
        uint_                           [ _val =
    |   '(' >> expr                     [ _val =
    |   '-' >> factor                   [ _val =
    |   '+' >> factor                   [ _val =
    ;
```

Semantic Actions



Semantic Actions

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

- Syntax:

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

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

A Calculator: The Compiler

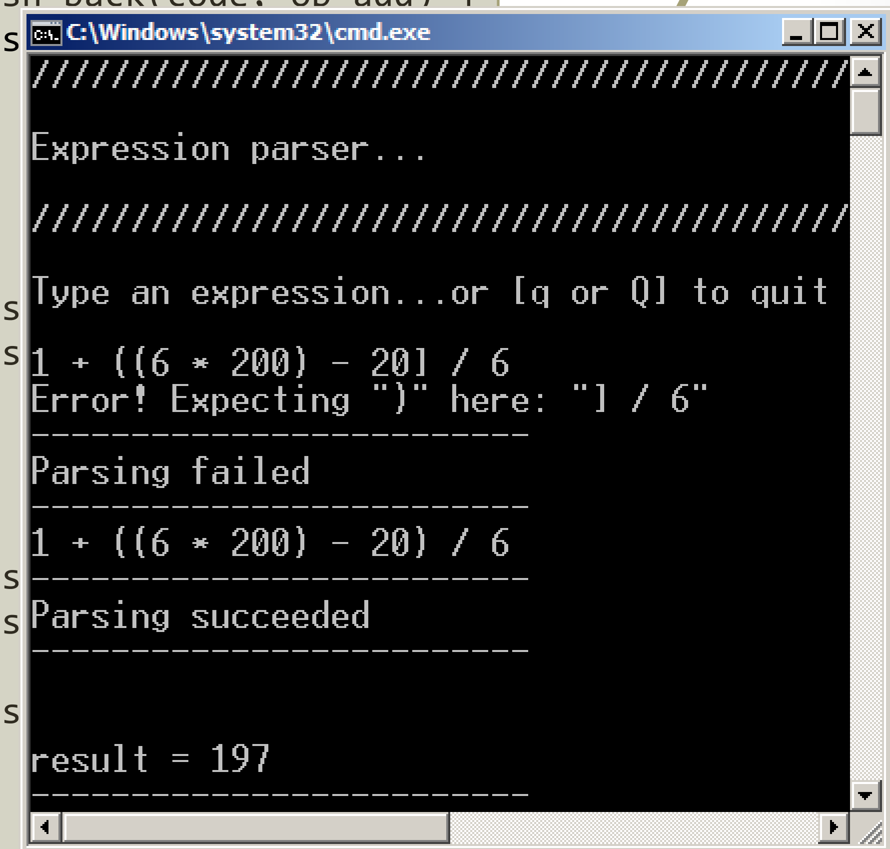
```
expression =
    term
    >> *(   '+'   > term        [ push_back(code, op_add) ]
        |   '-'   > term        [ push_back(code, op_sub) ]
        )
    ;

term =
    factor
    >> *(   '*'   > factor      [ push_back(code, op_mul) ]
        |   '/'   > factor      [ push_back(code, op_div) ]
        )
    ;

factor =
        uint_                   [ push_back(code, op_int),
                                  push_back(code, _1) ]
    |   '('       > expr > ')'
    |   '-'       > factor      [ push_back(code, op_neg) ]
    |   '+'       > factor
    ;
```

Expectation Points

A Calculator: The Compiler

```
expression =
    term
    >> *(   '+'   > term         [ push_back(code, op_add) ]
        |   '-'   > term         [ pus
        )
    ;

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

factor =
        uint_                    [ pus
                                   pus
    |   '('       > expr > ')'
    |   '-'       > factor       [ pus
    |   '+'       > factor
    ;
```

The Compiler



A Calculator: Creating an AST

- Here is the AST (simplified):

```
// A node of the AST holds either an integer, a binary
// operation description, or an unary operation description
struct ast_node
{
    boost::variant<int, binary_op, unary_op> expr;
};

// For instance, an unary_op holds the description of the
// operation and a node of the AST
struct unary_op
{
    char op;   // '+' or '-'
    ast_node subject;
};
```

```
struct binary_op
{
    char op; // '+', '-', '*', '/'
    ast_node left;
    ast_node right;
};
```

A Calculator: Creating an AST

```
template <typename Iterator>
struct calculator
  : grammar<Iterator, ast_node()>
{
    calculator() : calculator::base(expr)
    { /*…definition here*/ }

    rule<Iterator, ast_node()>
        expr, term, factor;
};
```

Grammar and Rule Signature

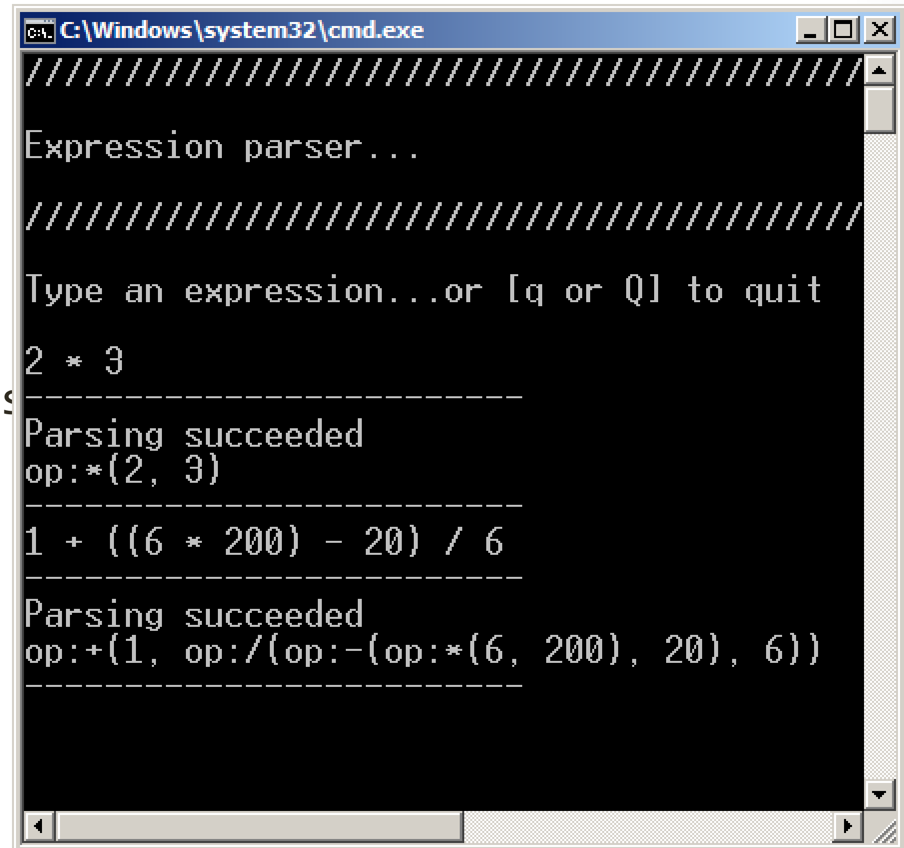
A Calculator: Creating an AST

Semantic Actions

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

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

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

A Calculator: Creating an AST

```
calculator calc;
ast_node ast;
std::string str("2*3");

// do it!
if (parse (str.begin(), s
    print_ast(ast);
```



Karma (Sanskrit: कर्म: act, action, performance) is the concept of "action" or "deed" in Indian religions understood as that which causes the entire cycle of cause and effect.

SPIRIT.KARMA

A LIBRARY FOR GENERATING OUTPUT

Generating Output

- Karma is the Yang to Qi's Yin
  - Everything you know about Qi's parsers is still true but has to be applied upside down (or inside out)

- Qi is all about *input* data matching and conversion, Karma is about converting and formatting data for *output*.
- Qi gets input from *input iterators*, Karma outputs the generated data using an *output iterator*
- Qi uses `operator>>()`, Karma uses `operator<<()`
- Qi's semantic actions are called *after* a match and *receive* a value, Karma's semantic actions are called *before* generating and *provide* one
- Qi's parser attributes are passed *up* (are *returned*), Karma's attributes are passed *down* (are *consumed*)

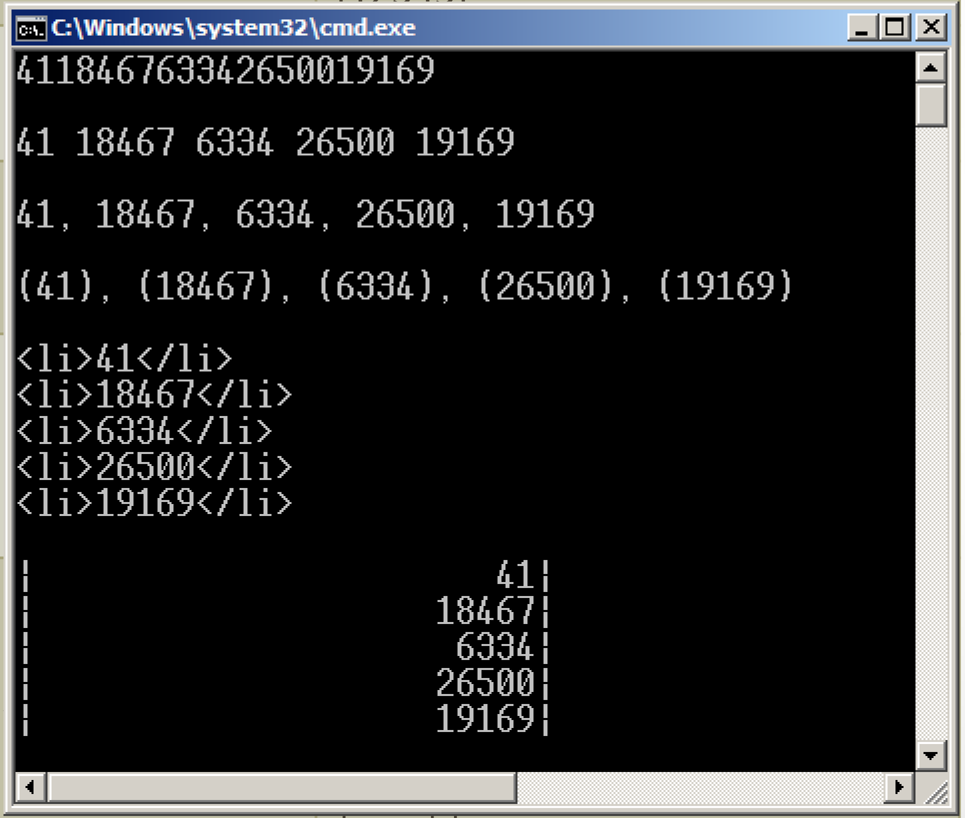
Generating Output

- Karmas DSEL (domain specific embedded language) was modeled after the PEG as used for parsing, i.e. set of rules describing what output is generated in what sequence:

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

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

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

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

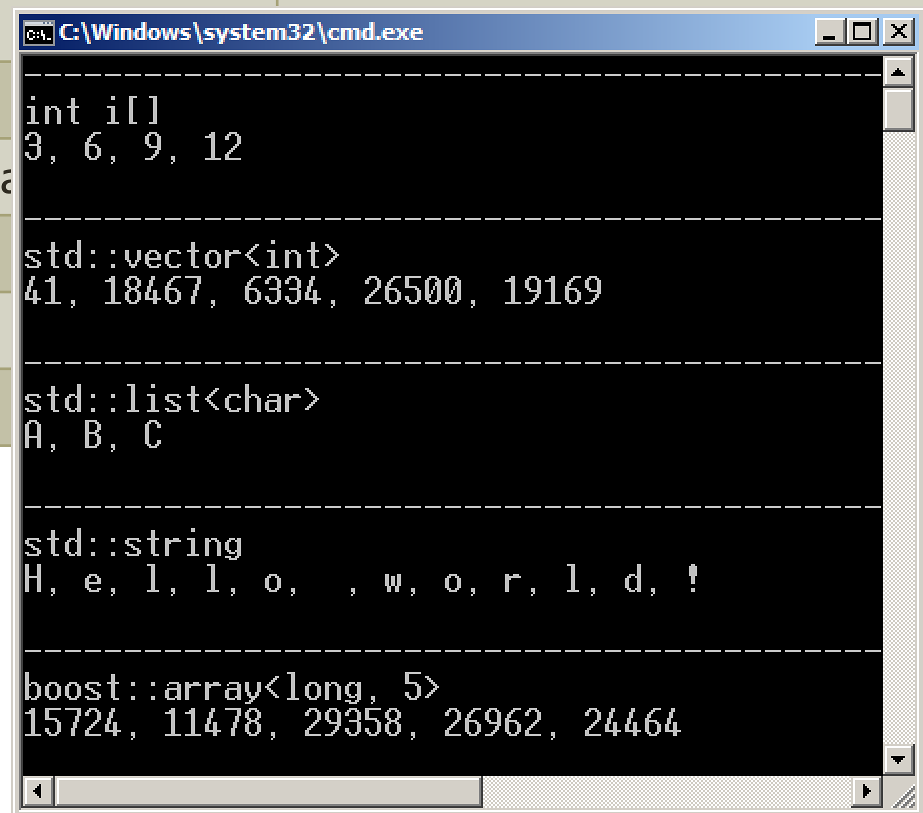
Different Output Grammars

Different output formats for:	std::vector<int>
'[' << *int_ << ']'	Without any separation: [12345]
'[' << (int_ % ", ") << ']'	
('(' << int_ << ')') % ",	
*(" " << int_ << " ")	
*("|" << right_align[int_] <	| 5|



Different Data Structures

Different data structures for:	stream % ", "
int i[4];	C style arrays
std::vector<int>	
std::list<char>	
std::vector<boost::gregoria	
std::string, std::wstring	
boost::iterator_range<…>	
boost::array<long, 20>	



Semantic Actions

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

- Syntax similar to parser semantic actions

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

- Easiest way to write semantic actions is phoenix
  - `_1, _2, …` refer to elements of generator
  - `_a, _b, …` refer to locals (for `rule<>`'s)
  - `_r1, _r2, …` refer to arguments (for `rule<>`'s))
  - _val refer to the left hand side's attribute
  - `pass` assign false to make generator fail

Expression Generator

- Here is the AST again (still simplified):

```
// A node of the AST holds either an integer, a binary
// operation description, or an unary operation description
struct ast_node
{
    boost::variant<int, binary_op, unary_op> expr;
};

// For instance, an unary_op holds the description of the
// operation and a node of the AST
struct unary_op
{
    char op;   // '+' or '-'
    ast_node subject;
};
```

```
struct binary_op
{
    char op; // '+', '-', '*', '/'
    ast_node left;
    ast_node right;
};
```

Expression Generator

```
template <typename OutputIterator>
struct gen_expr
    : grammar<OutputIterator, ast_node()>
{
    gen_expr() : gen_expr::base(ast)
    { /*…definition here*/ }

    rule<OutputIterator, ast_node()> ast;
    rule<OutputIterator, unary_op()> unode;
    rule<OutputIterator, binary_op()> binode;
};
```

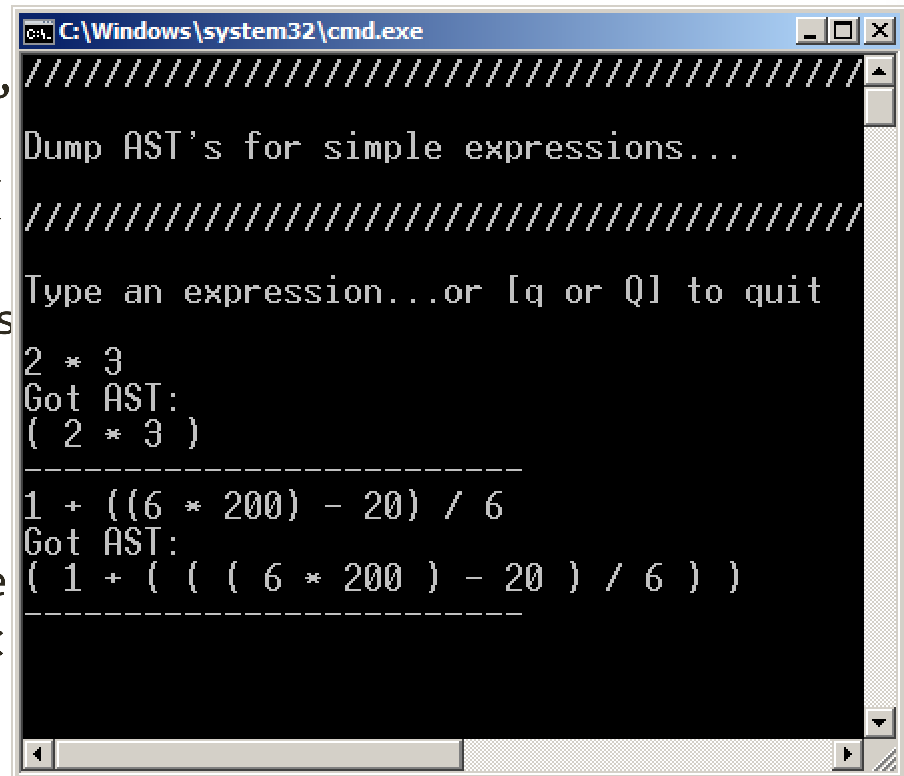
Infix Expression Generator

- Adapting the AST types as Fusion sequences

```
BOOST_FUSION_ADAPT_STRUCT(
    binary_op,
    (ast_node, left)(char,

BOOST_FUSION_ADAPT_STRUCT(
    unary_op,
    (char, op)(ast_node, s
```

- Format description:

```
ast        = int_ | binode
binode     = '(' << ast <<
unode      = '(' << char_
```



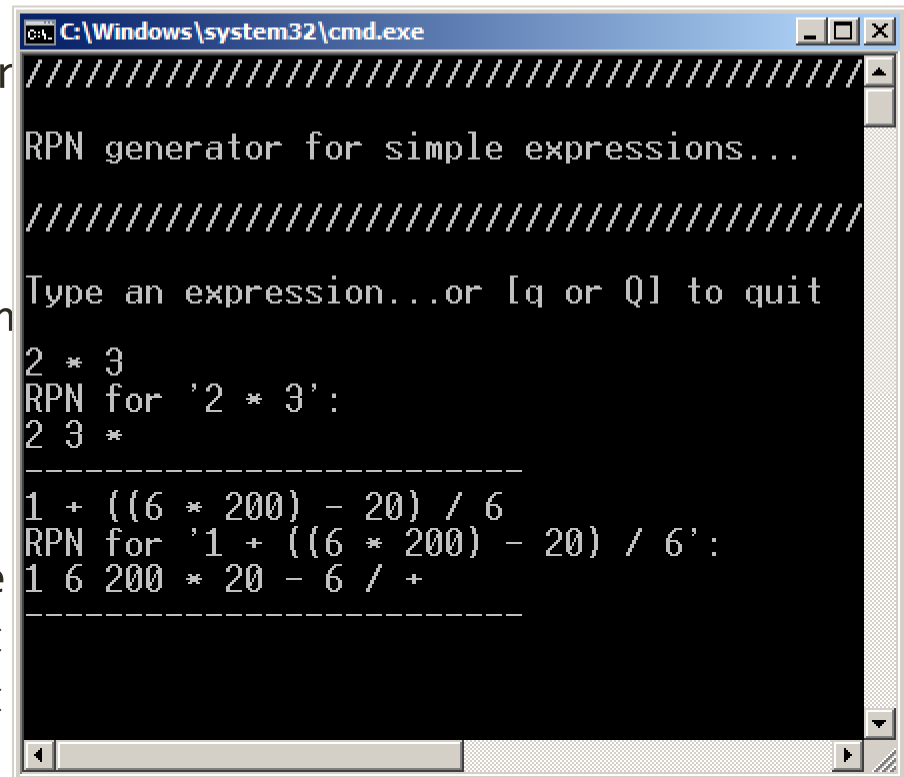
Postfix Expression Generator

- Adapting the AST types as Fusion sequences

```
BOOST_FUSION_ADAPT_STRUCT(
    binary_op,
    (ast_node, left)(ast_r

BOOST_FUSION_ADAPT_STRUCT(
    unary_op,
    (ast_node, subject)(ch
```

- Format description:

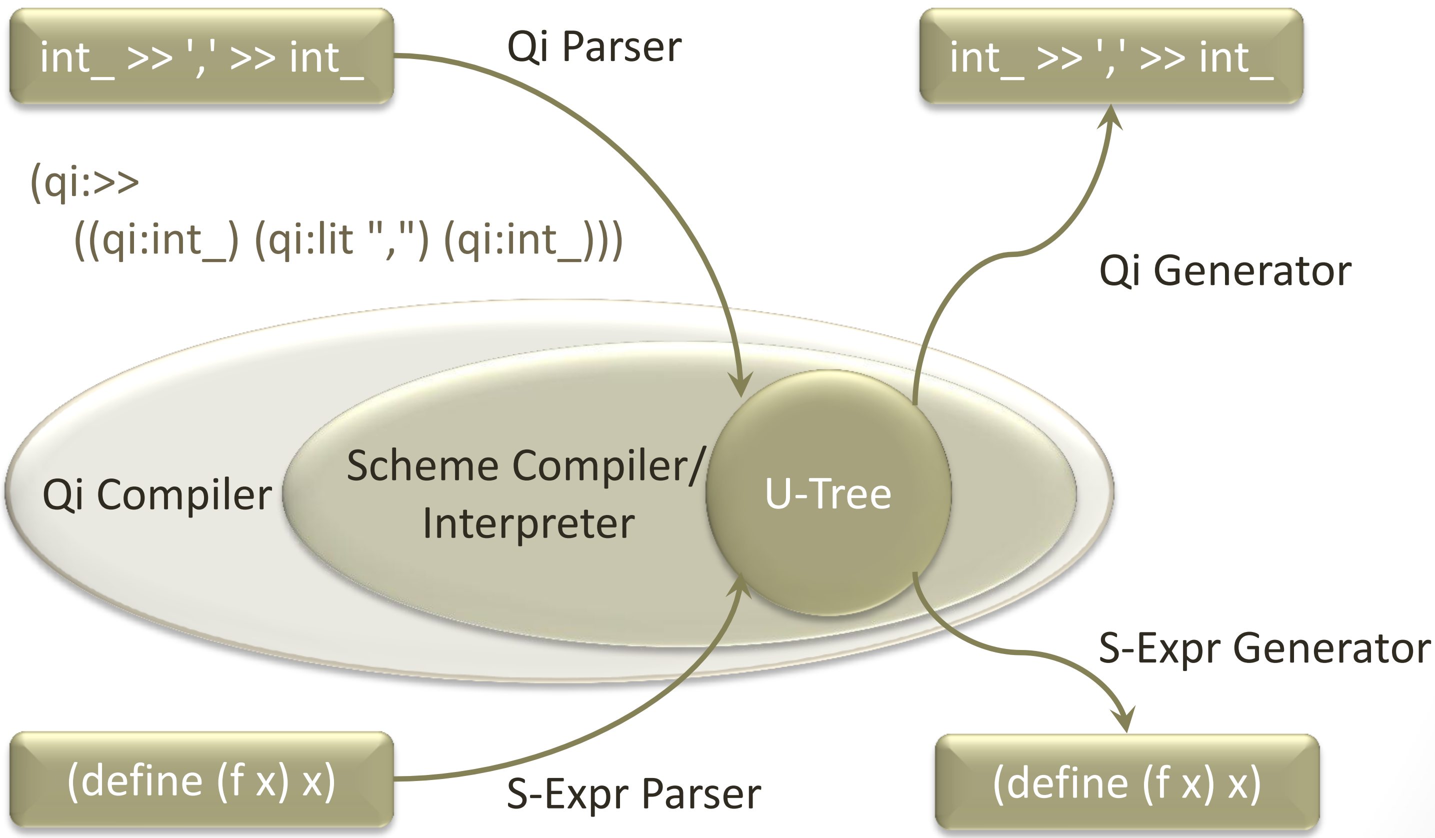
```
ast         = int_ | binode
binode      = '(' << ast <<
unode       = '(' << ast <<
```



The Spirit RAD Framework

SCHEME
THE MINIMALISTIC POWER

RAD Framework Overview



Scheme – Short Introduction

- A Small But Powerful Language
  - General-purpose
  - Scripting language
  - Extension language embedded within applications
- Derivative of LISP
- Abstract *lists* used universally for both data and functions
- Everything is an expression
- Lexically-scoped, block structured
- Dynamically typed
- Mostly functional language (but like C, it is still an imperative language with side-effects and all)
- First-class procedures (functions)
- Arguments are eagerly evaluated, but since functions are first class citizens, you can return functions for deferred evaluation

Scheme – Short Introduction

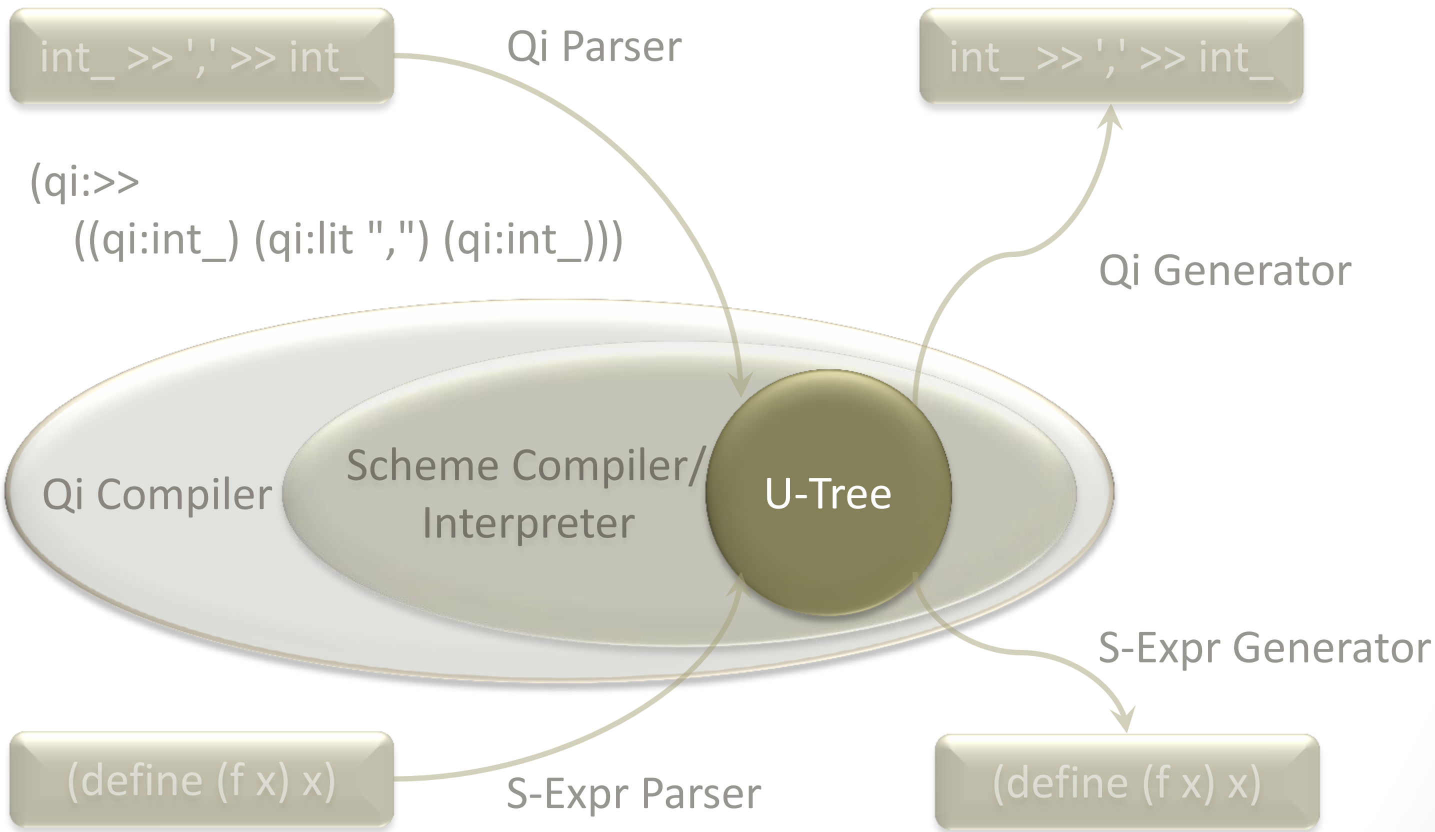
Everything is a prefix expression

```
(foo x y)                       ; foo(x, y)
(foo (bar x) (baz y))           ; foo(bar(x),baz(y))
(+ x y)                         ; x + y
(+ (* x y) (/ a b))             ; (x * y) + (a / b)
(if (< a b) a b)                ; if (a < b)
                                ;     return a;
                                ; else
                                ;     return b;
(define (square n) (* n n))     ; int square(int n)
                                ; {
                                ;     return n * n;
                                ; }
```

S-Expressions

- Symbolic expressions, or s-expressions, or sexps
- The language of LISP/Scheme programs (parenthesized prefix expressions)
- Very simple grammar
- Recursive, list based, data structures
- Can represent hierarchical information
  - A suitable (and terser!) replacement for XML
  - Even terser than JSON
- Other uses:
  - Document Style Semantics and Specification Language (DSSSL)
  - Internet Message Access Protocol (IMAP)
  - John McCarthy's Common Business Communication Language (CBCL)

The U-Tree


int_ >> ',' >> int_
Qi Parser
int_ >> ',' >> int_
(qi:>>
((qi:int_) (qi:lit ",") (qi:int_)))
Qi Generator
Qi Compiler
Scheme Compiler/
Interpreter
U-Tree
S-Expr Generator
(define (f x) x)
S-Expr Parser
(define (f x) x)

The U-Tree

Essentially:

```
variant<
  nil,
  string, symbol,
  list, range, st…e, reference,
  any, function>
```

()

The U-Tree

Essentially:

```
variant<
  nil, bool,
  string, symbol, binar
  list, range, string_ra                e,
  any, function>
```

true

The U-Tree

Essentially:

```
variant<
  nil, bool, int,
  string, symbol, binary,
  list, range, string_range,
  any, function>
```

123

The U-Tree

Essentially:

```
variant<
  nil, bool, int, double,
  string, symbol, binary,
  list, range, string_range, reference,
  any, function>
```

123.456

The U-Tree

Essentially:

```
variant<
  nil, bool, int, dou
  string,
  list, range, string
  any, function>
```

“Hello, World”

The U-Tree

Essentially:

```
variant<
  nil, bool, int, double,
  string, symbol,
  list, range, string_range,
  any, function>
```

foo

The U-Tree

Essentially:

```
variant<
  nil, bool, int, double,
  string, symbol, binary,
  list, range, string_range, r
  any, function>
```

#ff99dd#

The U-Tree

Essentially:

```
variant<
  nil, bool, int, double,
  string, symbol,
  list,
  any, function>
```

(foo 1 “jazz”)

The U-Tree

Essentia

Slices, String ranges and
References to Utrees.
→ For internal representations

```
variant<
  nil, bool, t, dou le,
  string, sy ol, bi ry,
  list, range, string_range, reference,
  any, function>
```

The U-Tree

Essentially:

```
variant<
  nil, bool, int, double,
  string, symbol, binary,
  list, range, ref
  any,
```

```
T* → Stored internally as void*
plus type_info.

any.get<T*>(); // runtime checked
```

The U-Tree

Essentially:

```
variant<
  nil, bool, int, double,
  string, symbol, binary
  list, range, re
  any, function
```

```
utree operator()(
    scope const& env) const;
```

scope → basically a container of arguments

U-Tree Examples

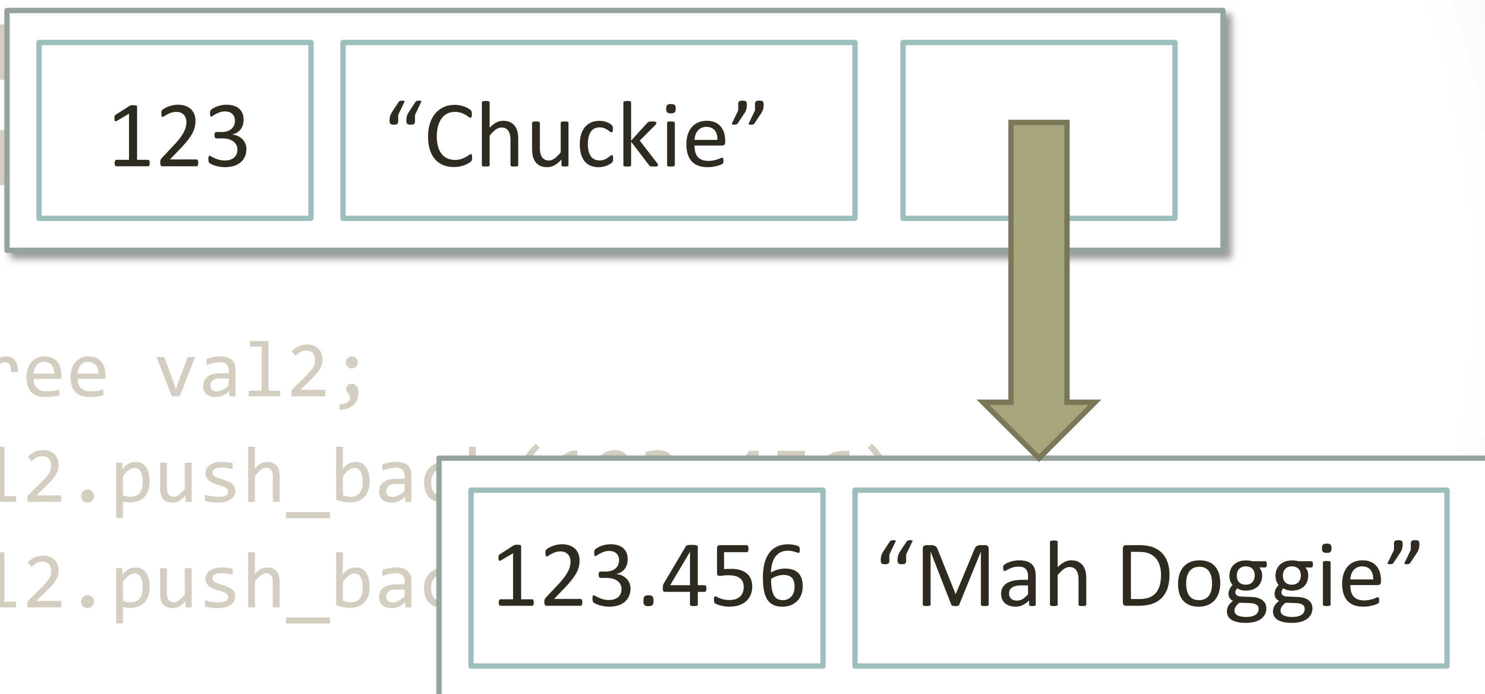
```
utree val;
utree val(true);
utree val(123);
utree val('x');
utree val(123.456);
utree val("Hello, World");
```

U-Tree Examples

```
utree val;
val.push_back(123);
val.push_back("Chuckie");

utree val2;
val2.push_back(123.456);
val2.push_back("Mah Doggie");

val.push_back(val2);
```

U-Tree Examples

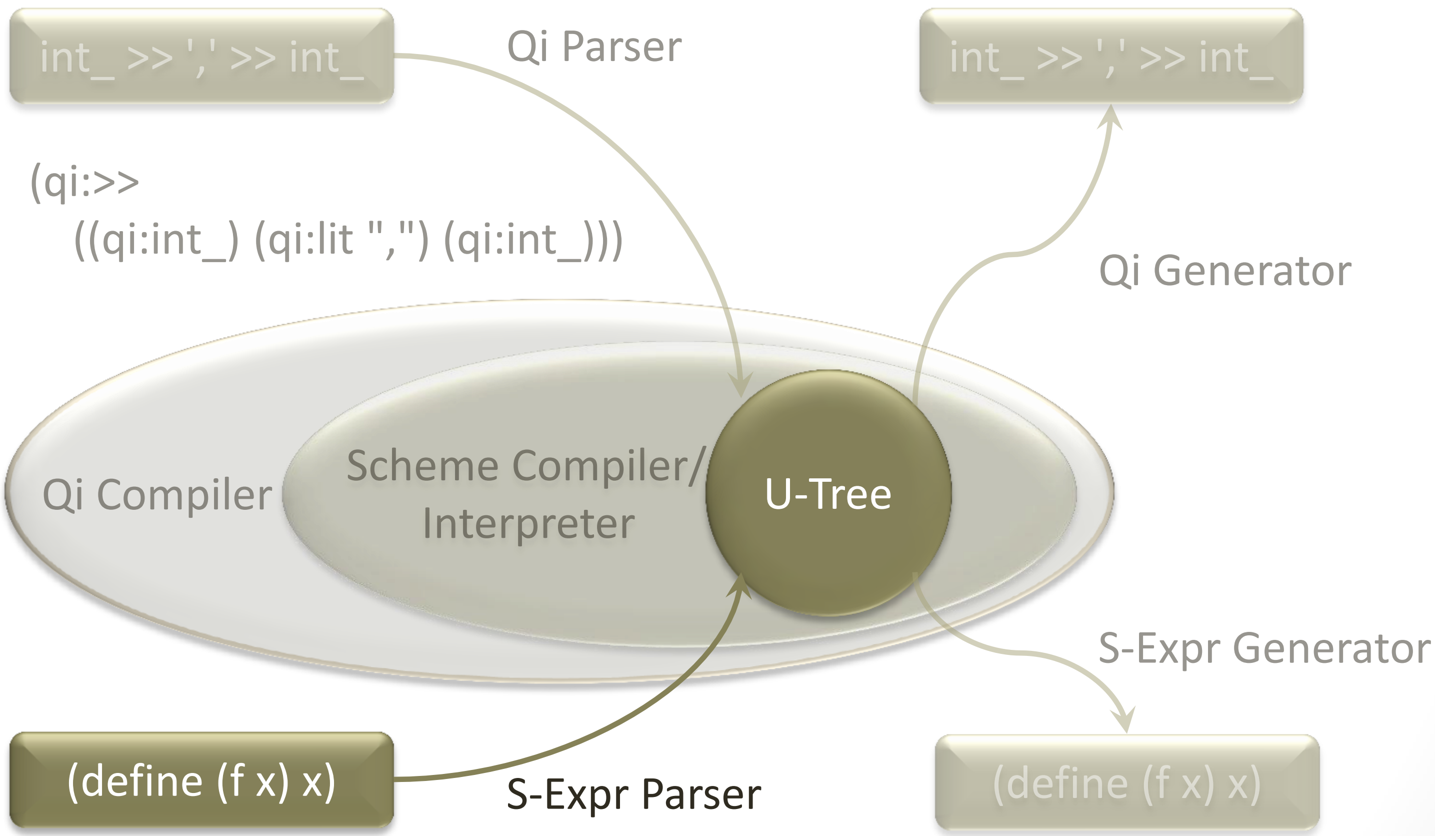
```
utree val;
val
val

utree val2;
val2.push_bac
val2.push_bac

val.push_back(val2);
```



U-Tree Examples

```
utree("apple") == utree("apple")
utree(1) < utree(2)
utree(456) + utree(789.123)

utree val(123);
utree ref(boost::ref(val));

utree alias(
    utree::range(b, e),
        scheme::shallow);
```

The S-Expr Parser



The S-Expr (Scheme) Parser

- Parsing S-expr (Scheme) input using *Qi* while creating an U-tree:

```
template <typename Iterator>
struct sexpr : grammar<Iterator, utree()> { … };
```

- Input:

```
(   123.45 true false 0xFF 077
    "this is a \u20AC string"       ; UTF-8 strings
    "Τη γλώσσα μου έδωσαν ελληνική"
    #0123456789ABCDEF0123456789ab#  ; A binary stream
    ( 92 ("another string" apple Sîne) ) )
```

  - A list of expressions which may be of type
    - symbol, double, int, boolean, string, binary data, or a list of those
  - Fully Unicode capable
    - Internally stored as UTF-8 byte sequences

The S-Expr (Scheme) Parser

```
// an element is: either an atom or a list
element = atom | list;                                   // utree()

// a list is: 0..N elements enclosed in '()'
list = '(' > *element > ')';                             // utree()

// an atom is: double, integer, string, binary data, symbol, or bool
atom = strict_double | integer | string_lit | byte_str | symbol | bool_;
                                                         // utree()
// an integer is: hexadecimal, octal, decimal
integer = lexeme[no_case["0x"] > hex] | lexeme['0' >> oct] | int_;
                                                           // int()
// binary data is: 1..N pairs of hex digits enclosed in '#'
byte_str = lexeme['#' > +hex2 > '#'];            // binary_string()

// a symbol is: a character sequence excluding some
std::string exclude = std::string(" ();\"\x01-\x1f\x7f") + '\0';
symbol = lexeme[+(~char_(exclude))];               // utf8_symbol()
```

The S-Expr (string_lit) Parser

- Parsing Unicode string
  - Matching escape sequences: '\u1234' and '\U12345678' inside strings and character literals
  - Matching ‘normal’ escape sequences: '\b', '\t', '\n', etc.
  - Converting input Unicode (UTF-16/UTF-32) code points to internally stored UTF-8 byte sequences
- Tricky as one input code point may have to be internally represented as a sequence of UTF-8 bytes
- The string_lit parser has `std::string` as its attribute, storing the UTF-8 bytes

The S-Expr (string_lit) Parser

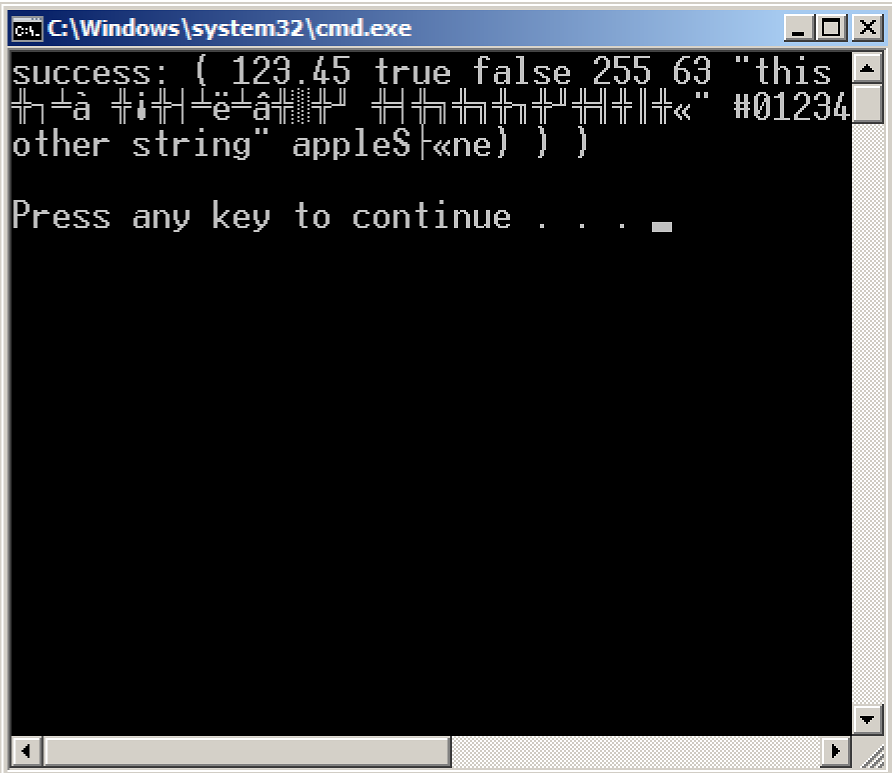
```
// a character literal is: a single escaped character or not a '\''
//                         enclosed in '\''
char_lit   = '\'' > (char_esc(_val) | (~char_('\'')) [_val += _1] ) > '\'';
                                                           // std::string()

// a string literal is: 0..N escaped characters or not '"'
//                      enclosed in '"'
string_lit =  '"' > *(char_esc(_val) | (~char_('"')) [_val += _1] ) > '"';
                                                           // std::string()

// an escaped character is: '\u1234', '\U12345678', or normal escaped char
char_esc   = '\\' > ( ('u' > hex4)        [push_utf8(_r1, _1)]
                    | ('U' > hex8)        [push_utf8(_r1, _1)]
                    | char_("btnfr\\\"'") [push_esc(_r1, _1)]
                    );                                // void(std::string&)
```

The S-Expr (String) Parser

```
// define a (lazy) function converting a single Unicode (UTF-32) codepoint
// to UTF-8
struct push_utf8_impl
{
    template <typename S, typename C>
    struct result { typedef void type; };

    void operator()(std::string& utf8, boost::uint32_t code_point) const
    {
        typedef std::back_insert_iterator<std::string> insert_iter;
        insert_iter out_iter(utf8);
        boost::utf8_output_iterator<insert_iter> utf8_iter(out_iter);

        *utf8_iter++ = code_point;
    }
};
boost::phoenix::function<push_utf8_impl> push_utf8;
```

The S-Expr (String) Parser

```
// define a (lazy) function converting a single Unicode (UTF-32) codepoint
// to UTF-8
struct push_esc_impl
{
    template <typename S, typename
    struct result { typedef void t

    void operator()(std::string& u
    {
        switch (code_point)
        {
            case 'b':  utf8 += '\b
            case 't':  utf8 += '\t
            // ...
            case '"':  utf8 += '"'
            case '\\': utf8 += '\\
        }
    }
};
boost::phoenix::function<push_esc_
```



The S-Expr Parser Error Handling

```
// define function object to be used as error handler
template <typename Iterator>
struct error_handler
{
    std::string source_file;
    error_handler(std::string const& source_file = "")
      : source_file(source_file) {}

    void operator()(Iterator first, Iterator last,
        Iterator err_pos, boost::spirit::info const& what) const
    {
        // print information about error
    }
};

// create instance of error handler
error_handler<Iterator> handler(source_file);
```

The S-Expr Parser Error Handling

- Error handlers take 4 parameters:
  - Begin of input sequence
  - End of input sequence
  - Error position in input sequence
  - Instance of `spirit::info` allowing to extract error context

```
// Install error handler for expect operators
on_error<fail>(element, handler(_1, _2, _3, _4));
```

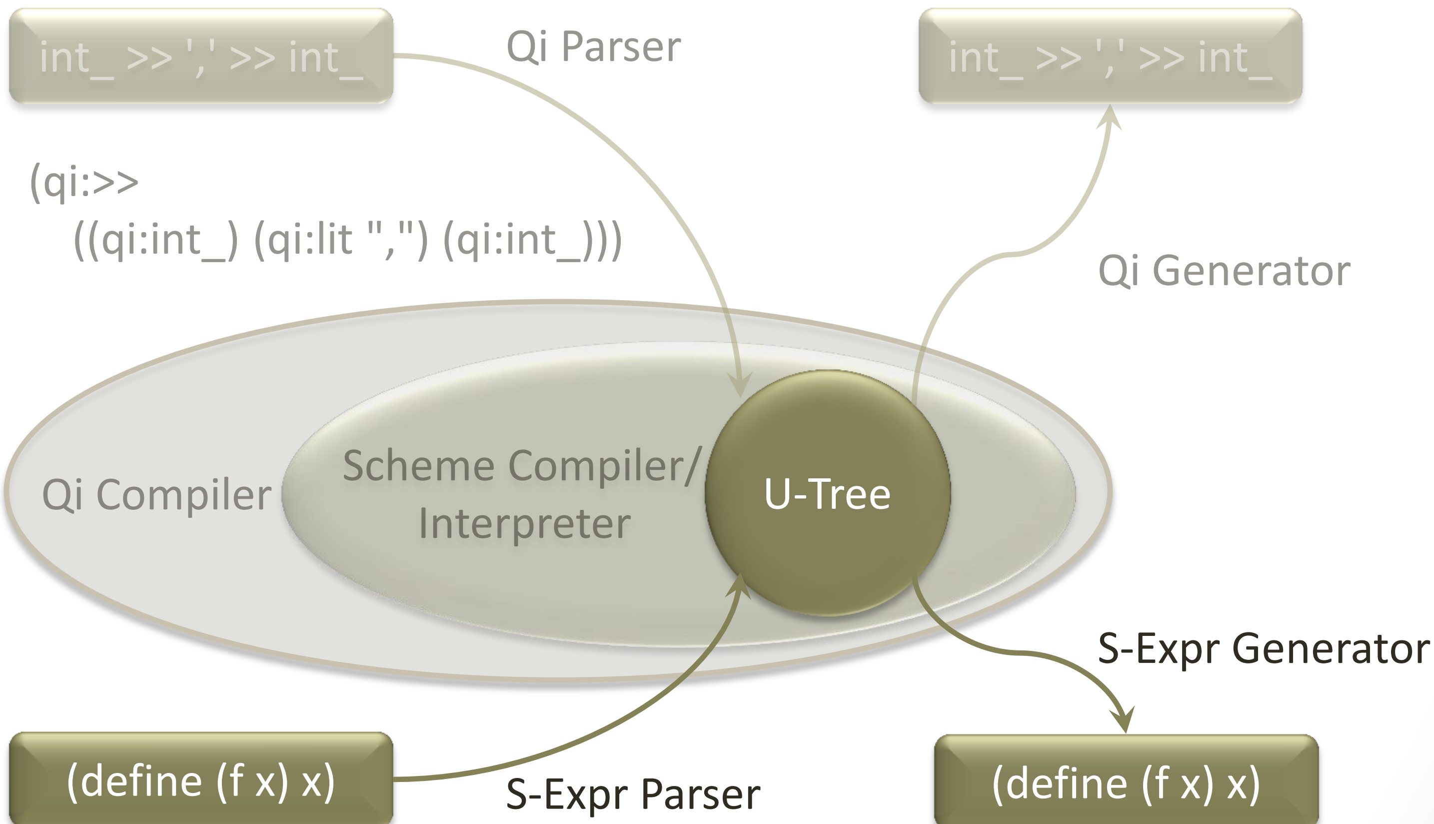
- Template parameter
  - `fail`: fail parsing
  - `retry`: retry after handler executed
  - `accept`: pretend parsing succeeded
  - `rethrow`: rethrow exception for next handler to catch

Lessons Learnt

- Definition of a `rule` *not* having semantic actions in its right hand side (or using `operator%=()` for initialization)
  - The rule's attribute is passed to the right hand side by reference
  - The right hand side's elements store their result directly in this attribute instance without any explicit code
  - Know the attribute propagation and attribute compatibility rules

- Definition of a `rule` having semantic actions in its right hand side
  - The rule creates a new instance of its attribute passing it to the right hand side elements
  - The right hand side's elements are responsible for storing their results in this attribute (using the place holder `_val`)

The Scheme Generator



The S-Expr (Scheme) Generator

- Generating S-expr (Scheme) output using *Karma* from a given u-tree

```
template <typename OutputIterator>
struct sexpr : grammar<OutputIterator, utree()> {…};
```

- Recreates the textual representation of an U-tree
  - Output in UTF-8
  - If output in UTF-16 or UTF-32 is required, additional output iterator wrapping is needed
- Based on type of current U-tree node (double, int, symbol, etc.) branch to corresponding format
  - Karma alternative (`operator|`) takes in variant (or variant like) attribute and does runtime dispatching based on actual stored type

The S-Expr (Scheme) Generator

```
// a node is: a double, int, string, symbol, binary data, a list of
//            nodes, or an empty node
node = double_ | int_ | bool_ | string_ | symbol | byte_str | list | nil;
                                                            // utree()

// a list of nodes is enclosed in '()'
list = '(' << *node << ')';                                 // utree()

// a (UTF-8) string is enclosed in '"'
string_ = '"' << string << '"';                       // utf8_string()

// a symbol is just a sequence of characters
symbol = string;                                      // utf8_symbol()

// binary data is a sequence of hex digit pairs enclosed in '#'
byte_str = '#' << *right_align(2, '0')[hex2] << '#';
                                                    // binary_string()
// nil prints nothing
nil = eps;                                                    // nil()
```

The S-Expr (Scheme) Generator

- Problem: U-tree is not boost::variant (obviously) and does not expose a similar interface
  - Out of the box it is not usable as an attribute for Karma's alternatives
  - Spirit has customization points (see the documentation)
    - Functions used by Spirit to access the attribute of a component
      - Need to be overloaded for custom types in user code
    - Templates which need to be specialized for custom types in user code
      - Need to (partially) specialize certain templates for custom types in user code
  - Some customization points are global for Spirit, some specific for Qi or Karma (use `domain::qi`, `domain::karma` to specialize)
  - Customization points are usually placed in

    `namespace boost::spirit::traits`
    - You are allowed to add your specializations and overloads there
  - We provide all necessary specializations and overloads for `scheme::utree`

The S-Expr (Scheme) Generator

```
// tell Spirit to handle scheme::utree as if - in the context of
// karma - it was a 'real' variant (namespace boost::spirit::traits)
template <>
struct not_is_variant<scheme::utree, karma::domain>
   : mpl::false_ {};

// map any node of type utree_type::double_type to alternative
// exposing double attribute (namespace boost::spirit::traits)
template <>
struct compute_compatible_component_variant<scheme::utree, double>
   : mpl::true_
{
    typedef double compatible_type;
    static bool is_compatible(int d)
    {
        return d == scheme::utree_type::double_type;
    }
};
```

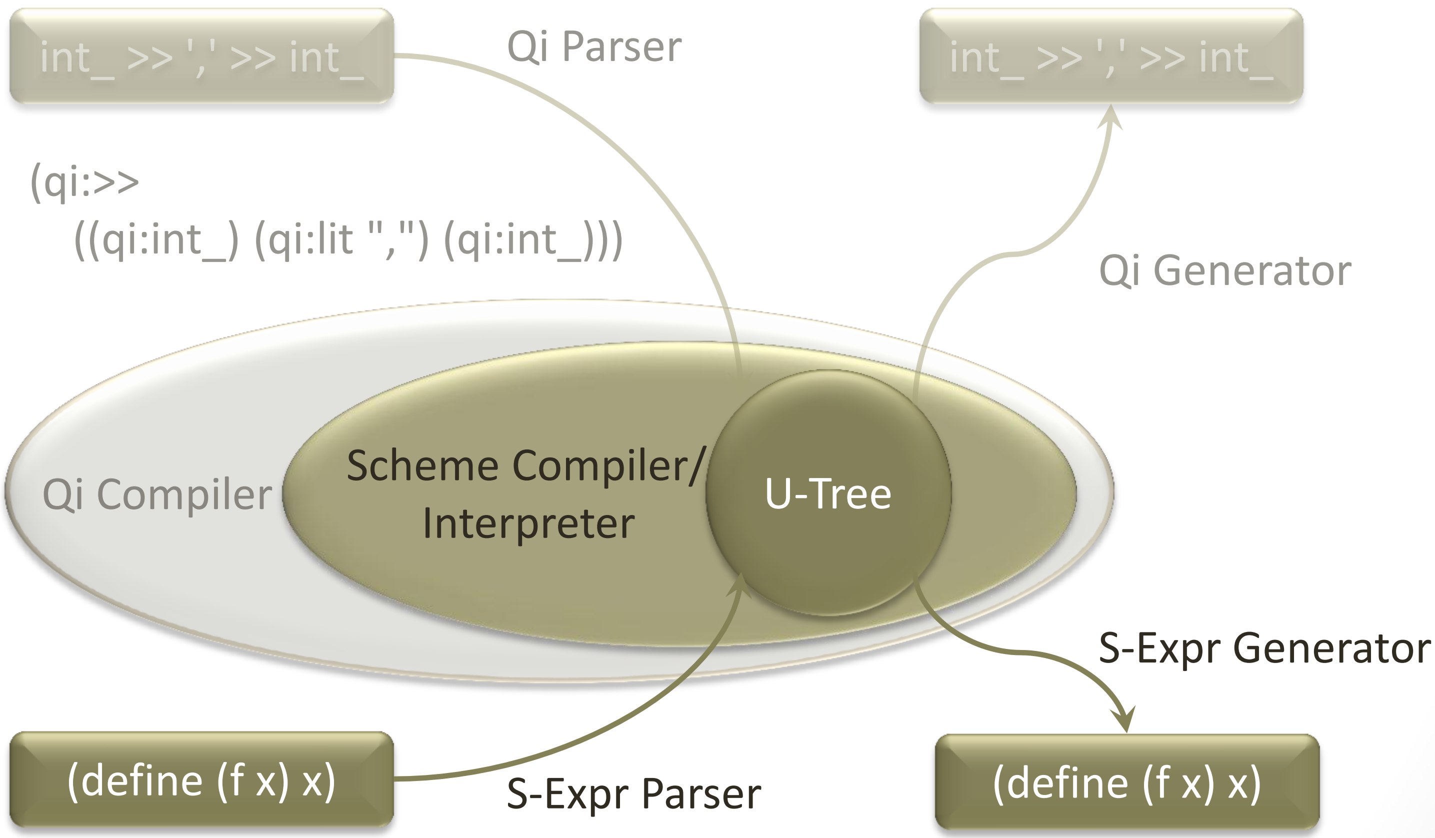
The S-Expr (Scheme) Generator

```
// return type of value which is stored in current node
// (namespace boost::spirit::traits)
template <>
struct variant_which<scheme::utree>
{
    int call(scheme::utree const& node)
    {
        return node.which();
    }
};

// return value stored in node as type T (namespace boost)
template <typename T>
T boost::get(scheme::utree const&)
{
    return node.get<T>();
}
```

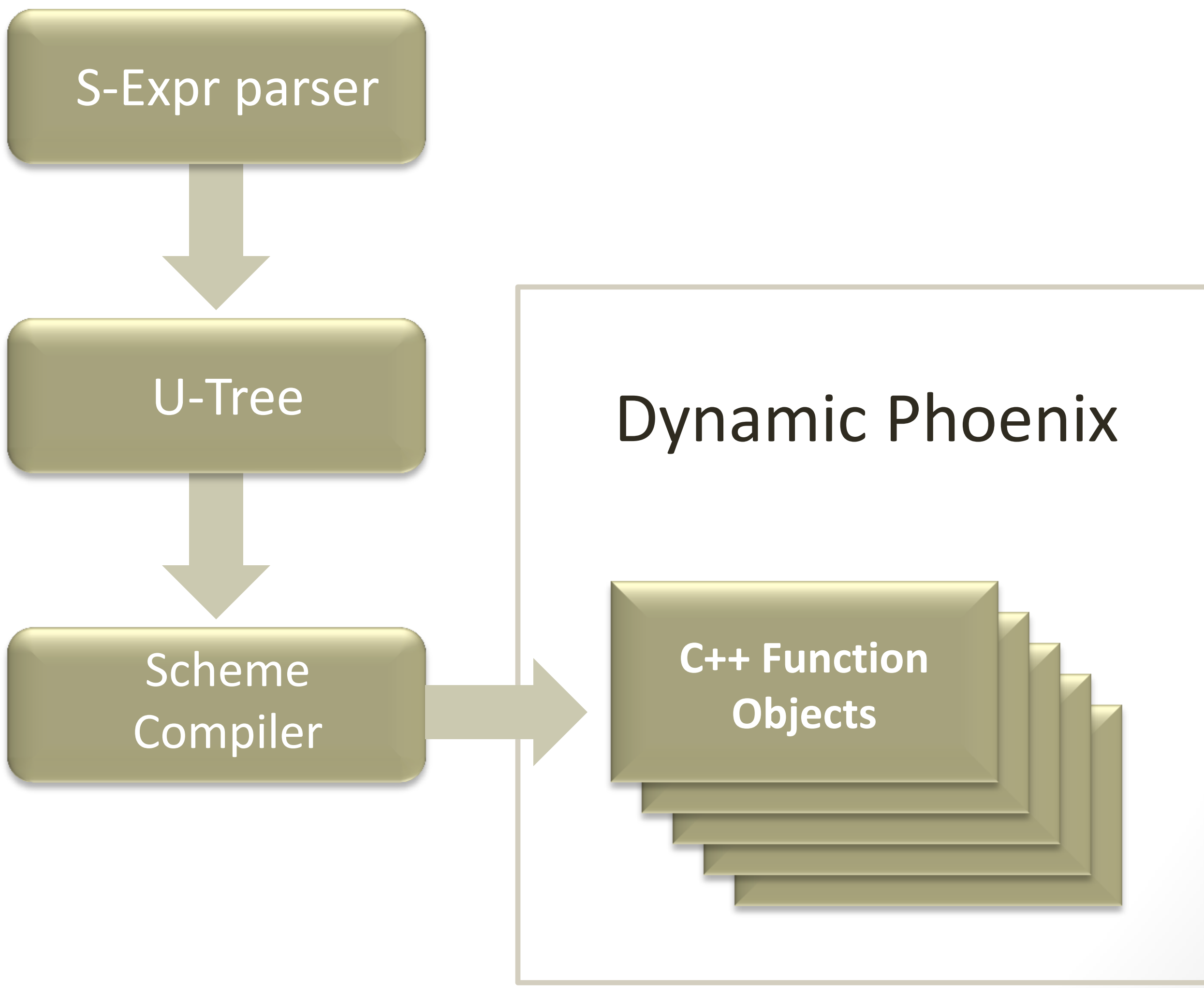
Lessons Learnt

- Build output formatting grammars based on your data types and not based on required output format
  - Attribute propagation is sufficiently powerful most of the time
  - Grammars are a natural extension of your data types, think about them as being 'yet another algorithm' to handle your data
- Prefer usage of customization points over semantic actions
  - Especially in Karma semantic actions (attribute handling by value) tend to be less efficient than direct attribute propagation (attribute handling by reference)
- Spirit's primitive components expose unique and well defined attribute types and allow for generic attribute handling
  - For instance, while the `int_` component exposes `int` as its attribute, it is still compatible with any integral data type
- Spirit's compound operators have unique and well defined built-in attribute handling capabilities
  - For instance, while sequences expose fusion::vector as their attribute, they are additionally compatible with containers (under certain circumstances)

Scheme Compiler/Interpreter



Scheme Compiler/Interpreter



Dynamic Phoenix

- Everything is a function
  - The C++ function object is the main building block
  - We compose functions to build more complex functions... to build more complex functions... and so on
  - Values are functions
  - Arguments (_1, _2 … _N) are functions
  - References (ref(x)) are functions
  - Control structures are functions
- Everything you know about *core Phoenix* applies. **Except!** We have one and only one function signature:

```
utree operator()(scope const& env) const;
```

The Scope

```
class scope : public boost::iterator_range<utree*>
{
public:

    scope(utree* first = 0,
        utree* last = 0,
        scope const* parent = 0);

    scope const* outer() const;
    int level() const;
};
```

The Actor

```
template <typename Derived>
struct actor
{
    typedef utree result_type;
    typedef actor<Derived> base_type;

    // operators here (later…)

    Derived const& derived() const
    {
        return *static_cast<Derived const*>(this);
    }
};
```

The Actor Operators

```
utree operator()(scope const& env) const
{
    return derived().eval(env);
}

utree operator()() const
{
    return derived().eval(scope());
}

template <typename A0>
utree operator()(A0 const& _0) const
{
    boost::array<utree, 1> elements;
    elements[0] = _0;
    return derived().eval(get_range(elements));
}
```

The Actor Operators

```
template <typename A0, typename A1>
utree operator()(A0 const& _0, A1 const& _1) const
{
    boost::array<utree, 2> elements;
    elements[0] = _0;
    elements[1] = _1;
    return derived().eval(get_range(elements));
}

template <std::size_t n>
static scope
get_range(boost::array<utree, n>& array)
{
    return scope(array.begin(), array.end());
}
```

The Polymorphic Function

```
struct function : actor<function>
{
    utree f;
    function() : f() {}
    function(utree const& f) : f(f) {}

    template <typename F> function(F const& f)
      : f(stored_function<F>(f)) {}

    utree eval(scope const& env) const
    {
        return f.eval(env);
    }
};
```

Values

```
struct value_function : actor<value_function>
{
    utree val;
    value_function(utree const& val) : val(val) {}

    utree eval(scope const& /*env*/) const
    {
        return utree(boost::ref(val));
    }
};

function val(utree const& x) const
{
    return function(value_function(x));
}
```

Arguments

```
struct argument_function : actor<argument_function>
{
    std::size_t n;
    argument_function(std::size_t n) : n(n) {}

    utree eval(scope const& env) const
    {
        utree const& arg = env[n];
        return arg.eval(env);
    }
};

function const _1 = argument_function(0);
function const _2 = argument_function(1);
function const _3 = argument_function(2);
```

The If Function

```
struct if_function : actor<if_function>
{
    function cond;
    function then;
    function else_;

    if_function(
        function const& cond,
        function const& then,
        function const& else_)
      : cond(cond), then(then), else_(else_)
    {}

    typedef utree result_type;
    utree eval(scope const& env) const
    {
        return cond(env).get<bool>() ? then(env) : else_(env);
    }
};
```

The Composite

```
template <typename Derived>
struct composite
{
    typedef function result_type;
    typedef composite<Derived> base_type;

    // operators here. (later …)

    Derived const& derived() const
    {
        return *static_cast<Derived const*>(this);
    }
};
```

Composite Operators

```
function operator()(actor_list const& elements) const
{
    return derived().compose(elements);
}

template <typename A0>
function operator()(A0 const& _0) const
{
    actor_list elements;
    elements.push_back(as_function(_0));
    return derived().compose(elements);
}
```

The If Composite

```
struct if_composite : composite<if_composite>
{
    function compose(actor_list const& elements) const
    {
        actor_list::const_iterator i = elements.begin();
        function if_ = *i++;
        function then = *i++;
        function else_ = *i;
        return function(if_function(if_, then, else_));
    }
};

if_composite const if_ = if_composite();
```

Actors On Stage!

```
plus(11, 22, 33)            ()          == utree(66)
plus(11, 22, _1)            (33)        == utree(66)
plus(11, _1, _2)            (22, 33)    == utree(66)
plus(11, plus(_1, _2))      (22, 33)    == utree(66)

lambda factorial;
factorial =
    if_(lte(_1, 0), 1,
        times(_1, factorial(minus(_1, 1))));

factorial(_1)               (10)        == utree(3628800)
```

Our Objective

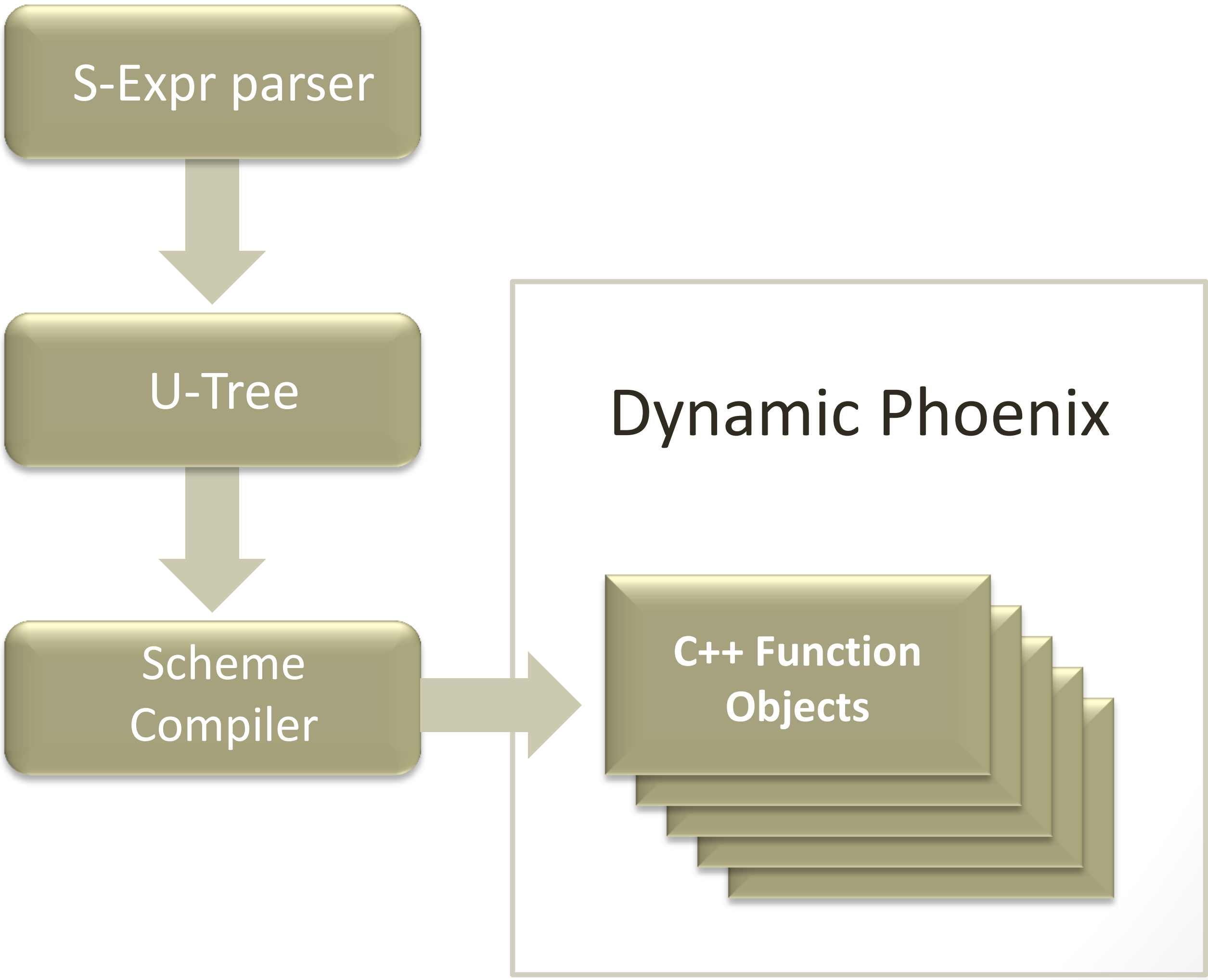
Transform this:

```
(define (factorial n)
    (if (<= n 0) 1 (* n (factorial (- n 1)))))
```

To This:

```
factorial =
    if_(lte(_1, 0), 1, times(_1,
          factorial(minus(_1, 1))));
```

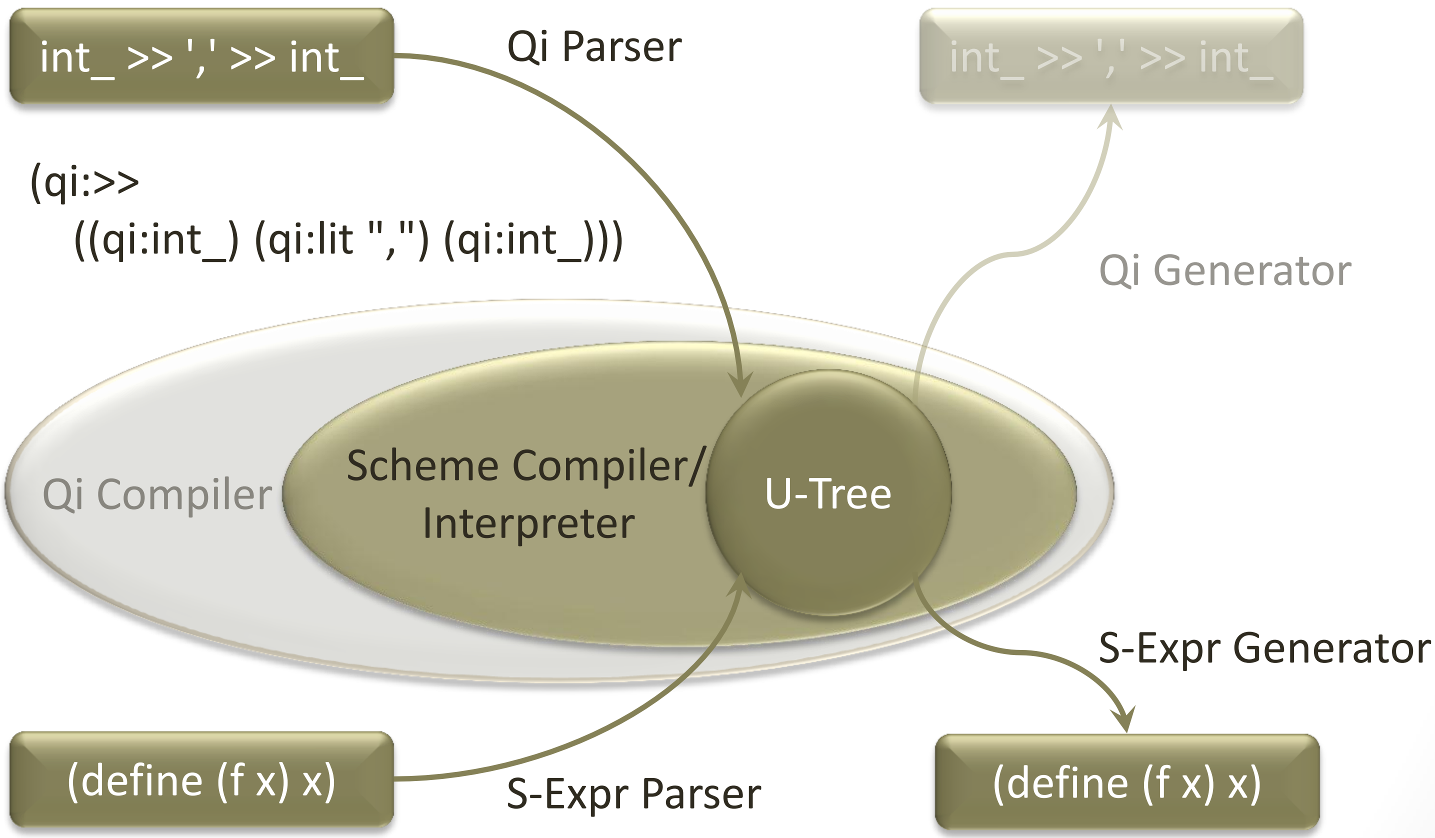
Scheme Compiler/Interpreter



Scheme Interpreter Example

```
using scheme::interpreter;
using scheme::function;
using scheme::utree;

utree src =
    "(define (factorial n) "
        "(if (<= n 0) 1 (* n (factorial (- n 1)))))";

interpreter program(src);
function factorial = program["factorial"];
std::cout << factorial(10) << std::endl;
```

Qi Parser



int_ >> ',' >> int_
Qi Parser
int_ >> ',' >> int_
(qi:>>
((qi:int_) (qi:lit ",") (qi:int_)))
Qi Generator
Qi Compiler
Scheme Compiler/
Interpreter
U-Tree
S-Expr Generator
(define (f x) x)
S-Expr Parser
(define (f x) x)

Qi Parser

- Goal: convert Qi expressions into S-Expressions
  - Allow uniform interpretation: RAD tool
  - Allow to use Scheme code to transform the parser expression
    - Anything is possible, for instance: left recursion elimination, attribute analysis, etc.
  - No information loss, it should be possible to recreate the Qi expression encoded in S-Expr
- Parser should create an U-Tree instance encoding the Qi expressions
  - S-Expr symbols are the Qi names prefixed with "qi:"
    - `int_` → `'qi:int_'`
    - `>>` → `'qi:>>'`
  - Each parser component will be stored as a separate list-node:
    - `int_` → `(qi:int_)`
    - `char_('a')` → `(qi:char_ "a")`
    - `int_ >> char_` → `(qi:>> (qi:int_) (qi:char_))`
  - `(car p)` → refers to parser component
  - `(cdr p)` → refers to list of arguments

Qi Parser

```
// sequence: A >> B --> (qi:>>  A   B )
sequence =
        unary_term
    >> *( ">>" >> unary_term                              )
    ;
                                                           // utree()

// unary operators: *A --> (qi:*  A )
unary_term =
        "*" >> unary_term
    |   "+" >> unary_term
    |   "-" >> unary_term
    |   "&" >> unary_term
    |   "!" >> unary_term
    |   term
    ;
                                                           // utree()

// A, directives, (A) --> (A)
term = primitive | directive | '(' >> sequence >> ')';
                                                           // utree()
```

Qi Parser

```
// sequence: A >> B --> (qi:>> (A) (B))
sequence =
        unary_term               [ _val = _1 ]
    >> *( ">>" >> unary_term  [ make_sequence(_val, _1) ] )
    ;
                                                            // utree()

// unary operators: *A --> (qi:* (A))
unary_term =
        "*" >> unary_term     [ make_kleene(_val, _1) ]
    |   "+" >> unary_term     [ make_plus(_val, _1) ]
    |   "-" >> unary_term     [ make_optional(_val, _1) ]
    |   "&" >> unary_term     [ make_and_pred(_val, _1) ]
    |   "!" >> unary_term     [ make_not_pred(_val, _1) ]
    |   term                  [ _val = _1 ]
    ;
                                                            // utree()

// A, directives, (A) --> (A)
term = primitive | directive | '(' >> sequence >> ')';
                                                            // utree()
```

Qi Parser

```
// any parser directive: lexeme[A] --> (qi:lexeme (A))
directive = (directive0 >> '[' >> alternative >> ']')
                                [ make_directive(_val, _2, _1) ];
                                                       // utree()

// any primitive parser: char_('a') --> (qi:char_ "a")
primitive %=
        primitive2 >> '(' >> literal >> ',' >> literal >> ')'
    |   primitive1 >> '(' >> literal >> ')'
    |   primitive0                          // taking no parameter
    |   literal [ make_literal(_val) ]
    ;
                                                       // utree()
// a literal (either 'x' or "abc")
literal =
        string_lit            [ phoenix::push_back(_val, _1) ]
    |   string_lit.char_lit   [ phoenix::push_back(_val, _1) ]
    ;
                                                       // utree()
```

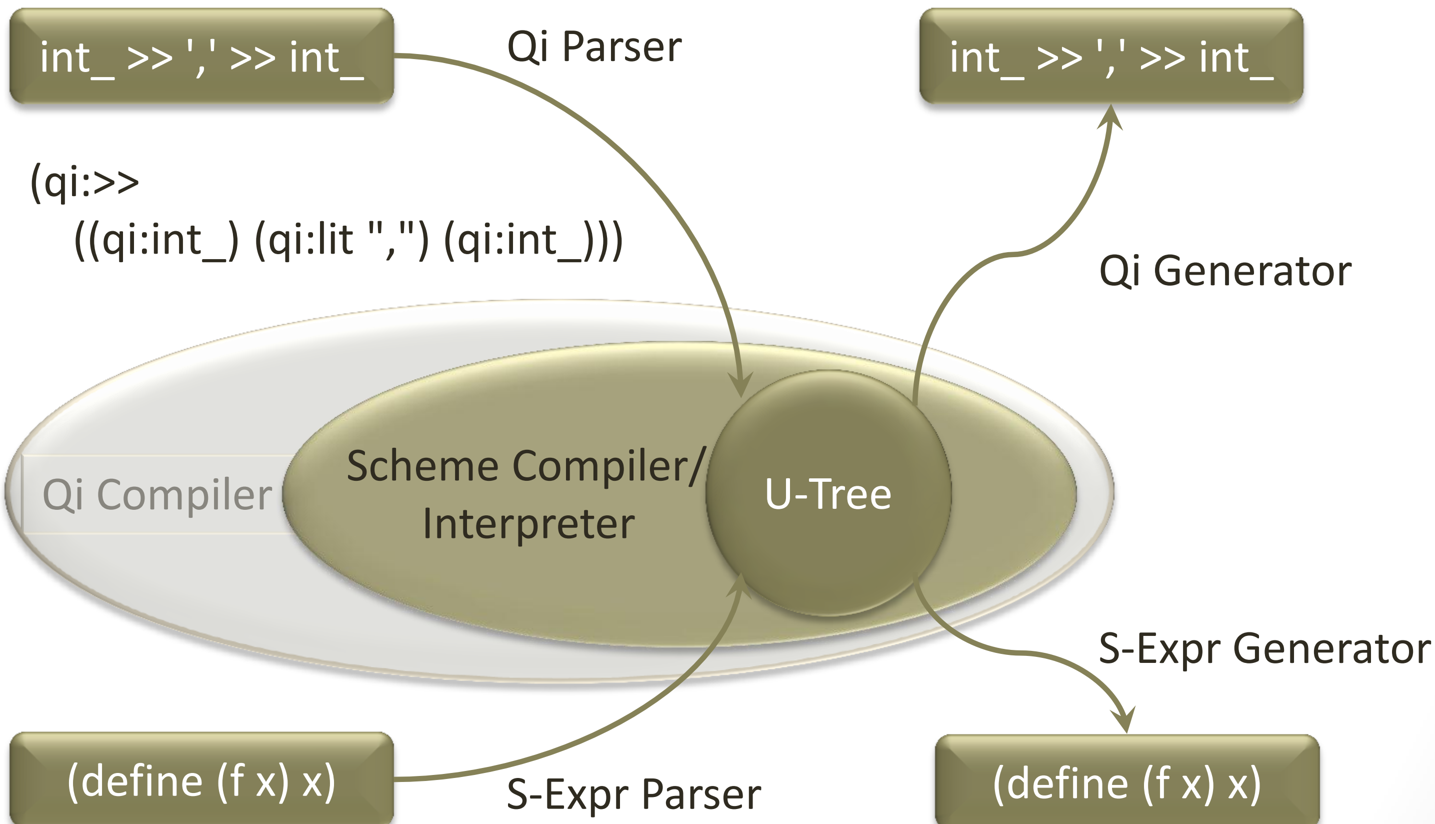
Qi Parser

```
// symbols parser recognizes keywords
qi::symbols<char, utree> primitive1;

// a list of names for all supported parser primitives
// taking 1 parameter
static char const* const primitives1[] =
{
    "char_", "lit", "string", 0
};

// initialize symbols parser with all corresponding keywords
std::string name("qi:");
for (char const* const* p = primitives1; *p; ++p)
{
    utree u;
    u.push_back(utf8_symbol(name + *p));
    primitive1.add(*p, u);
}
```

Lessons Learnt

- Write a parser based on given input structure (format) and not driven by required internal data structures
  - Formalize structure of input strings, identify terminals and non-terminals
    - Non-terminals are expressed as rule's, terminals as predefined components
    - Very much like structuring procedures, matter of experience, taste, personal preferences
  - If internal representation is *not* given
    - Create internal data structures matching the default attributes as exposed by the terminals and non-terminals of the parser
  - If internals representation is already given
    - Use `BOOST_FUSION_ADAPT_[STRUCT|CLASS]` to convert structures into Fusion sequences
      - Use `BOOST_FUSION_ADAPT_[STRUCT|CLASS]_NAMED` to define several different bindings
    - Use `fusion::nview` to reorder (or skip) elements of a Fusion sequence
    - Use customization points to make your data structures expose the interfaces expected by Spirit
    - Create global factory functions allowing to convert attributes exposed by parser components to your data types
    - Use semantic actions as a last resort

Qi Generator



Qi Generator (Naïve Version)

```
// sequence: (qi:>> (A) (B) …) → (A) >> (B) >> …
sequence =
        &string("qi:>>") << '(' << term % ">>" << ')'
    |   term
    ;                                                          // utree()
// term: (qi:* (A)) → (*A)
// either a unary, a primitive, a directive, or a (nested) sequence
term =  unary << '(' << sequence << ')'
    |   primitive2 << '(' << literal << ',' << literal << ')'
    |   primitive1 << '(' << literal << ')'
    |   primitive0
    |   directive0 << '[' << sequence << ']'
    |   sequence
    ;                                                          // utree()
// symbols generator is like an 'inverse' symbol table
symbols<scheme::utf8_symbol> primitive1;
std::string name("qi:");
for (char const* const* p = primitives0; *p; ++p)
    primitive1.add(utf8_symbol(name + *p));
```

Strict and Relaxed Modes

- Default mode is relaxed (or activated by `relaxed[]` directive)
  - Attributes may contain more data than expected by format
    - `int_ << char_:` may get passed a longer Fusion sequence `fusion::vector<int, char, double>`
    - `int_ << int_ << int_:` may consume container holding more than 3 integers
    - `repeat(3)[int]:` may consume container holding more than 3 elements
  - Repetitive generators silently skip failed invocations of their embedded generators
    - `*(int_[_pass = _1 % 2]):` will output only odd integers of consumed container
  - Alternatives silently accept attributes not convertible to any of the attribute types exposed by the alternative
    - Attribute: `variant<double, char const*> v (10.0);`
    - Format: `char_ | lit(11)`, will generate: `11`

Strict and Relaxed Modes

- Strict mode is activated by `strict[]` directive
  - Number of attributes must match number of generated elements
  - All of elements in containers must be consumed by generators (sequences and repetitive generators)
  - Alternatives fail immediately if attribute is not convertible to one of the consumed attributes of the format alternatives
    - Attribute: `variant<double, char const*> v (10.0);`
    - Format: `char_ | lit(11)`, will fail

- Compile time only directives, no runtime impact
  - Allow to fine tune behavior of compound operations

Qi Generator (Better Version)

```
// sequence: (qi:>> (A) (B) …) → (A) >> (B) >> …
sequence =
        &symbol(ref("qi:>>")) << '(' << strict[term % ">>"] << ')'
    |   term
    ;
                                                               // utree()
// term: (qi:* (A)) → (*A)
// either a unary, a primitive, a directive, or a (nested) sequence
term =  strict[
        unary << '(' << sequence << ')'
    |   primitive2 << '(' << literal << ',' << literal << ')'
    |   primitive1 << '(' << literal << ')'
    |   primitive0
    |   directive0 << '[' << sequence << ']'
    |   sequence
    ];
                                                               // utree()
```

Creating Your Own Directive

- Consider U-tree contains this data:
  ```
  // r = int_ >> double_
  [(define (r) (qi:>> (qi:int_) (qi:double_)))]
  ```
- If we wrote output format as:
  ```
  rule_ = &symbol(ref("define")) << rule_name << '=' << alternative;
  ```
  - Then `rule_name` and `alternative` would receive `[(r)]` and `[(qi:>> (qi:int_) (qi:double_))]` resp.
  - While they need to receive: `[r]` and `[qi:>> (qi:int_) (qi:double_)]`
- Easiest way to 'dereference' is to use repetitive container: `repeat(1)[…]`:
  ```
  rule_ = &symbol(ref("define")) <<
            repeat(1)[rule_name] << '=' << repeat(1)[alternative];
  ```
- Wouldn't it be nice if we could write:
  ```
  rule_ = &symbol(ref("define")) <<
            deref[rule_name] << '=' << deref[alternative];
  ```

Creating Your Own Directive

```
// meta-function exposing the type of new deref placeholder based on
// the type of repeat(N)
namespace traits
{
    template <typename Count>
    struct deref_spec_type
      : boost::spirit::result_of::terminal<
          boost::spirit::tag::repeat(Count)>  // uses predefined helper
    {};
}

// helper function to define new placeholder
inline typename traits::deref_spec<int>::type
deref_spec()
{
    return boost::spirit::karma::repeat(1);
}

typedef traits::deref_spec<int>::type deref_tag_type;
deref_tag_type const deref = deref_spec();   // defines new placeholder
```

Qi Generator (Final Version)

```
// sequence: (qi:>> (A) (B) …) → (A) >> (B) >> …
sequence =
        &symbol(ref("qi:>>")) << '(' << strict[term % ">>"] << ')'
    |   term
    ;
                                                            // utree()
// term: (qi:* (A)) → (*A)
// either a unary, a primitive, a directive, or a (nested) sequence
term =  strict[
        unary << '(' << deref[sequence] << ')'
    |   primitive2 << '(' << literal << ',' << literal << ')'
    |   primitive1 << '(' << literal << ')'
    |   deref[primitive0]
    |   directive0 << '[' << deref[sequence] << ']'
    |   deref[sequence]
    ];
                                                            // utree()
```

Lessons Learnt

- Karma has now debug mode as well:
  - Either,

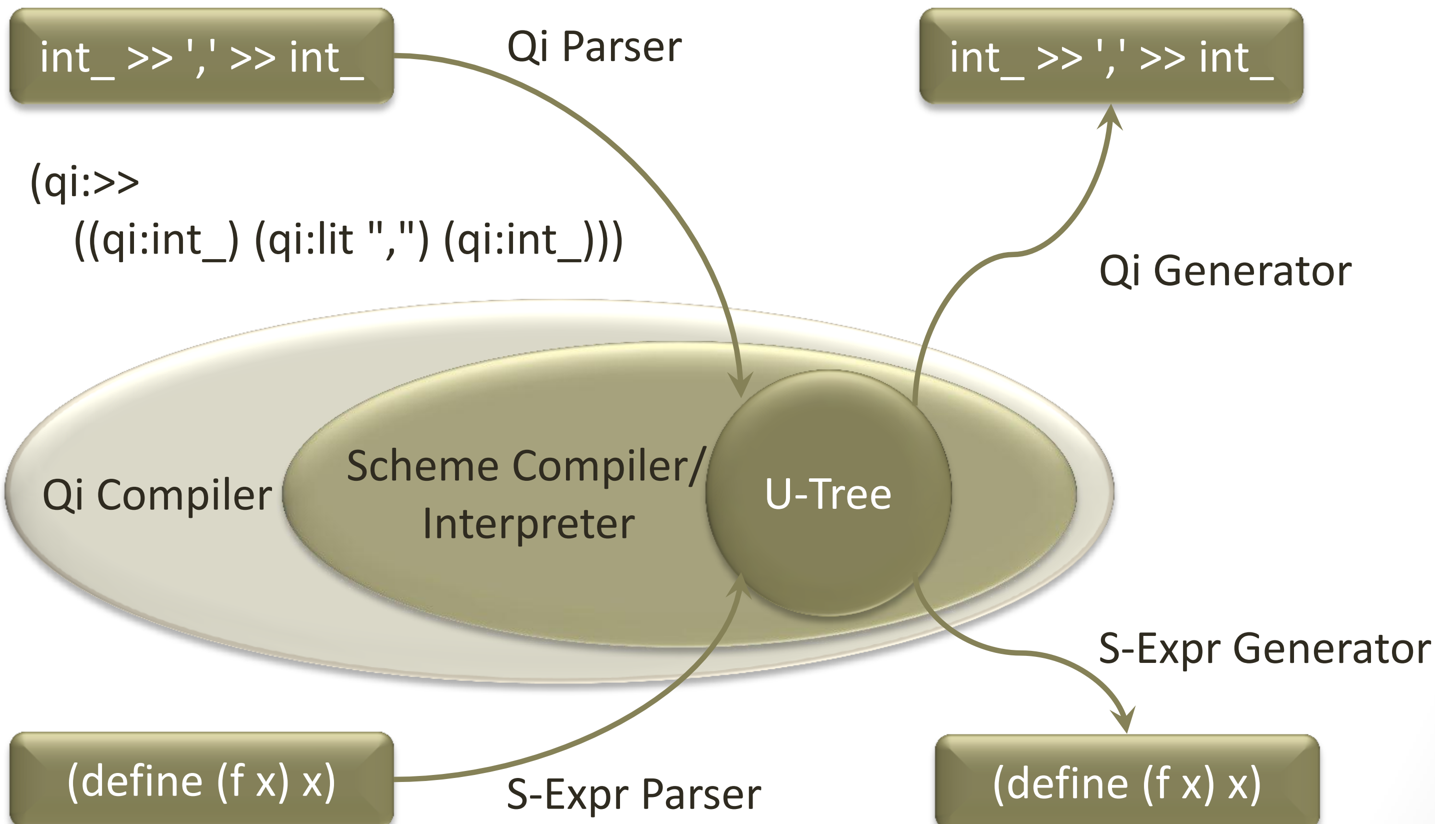
    `#define BOOST_SPIR`

    `BOOST_SPIRIT_DEBUG`
  - Or, register the rules to deb

    `r.name("name"); de`
- Karma generators may fail
  - If consumed attribute is no
    - i.e. `int_(10)` will fail for
  - Alternatives fail if all sub-e
  - Repetitive generators fail if container attribute do not
    - i.e. plus will fail for empty containers
  - Epsilon fails if supplied expression is `false`
    - i.e. `eps(_1 % 2)` fails if `_1` is an odd number

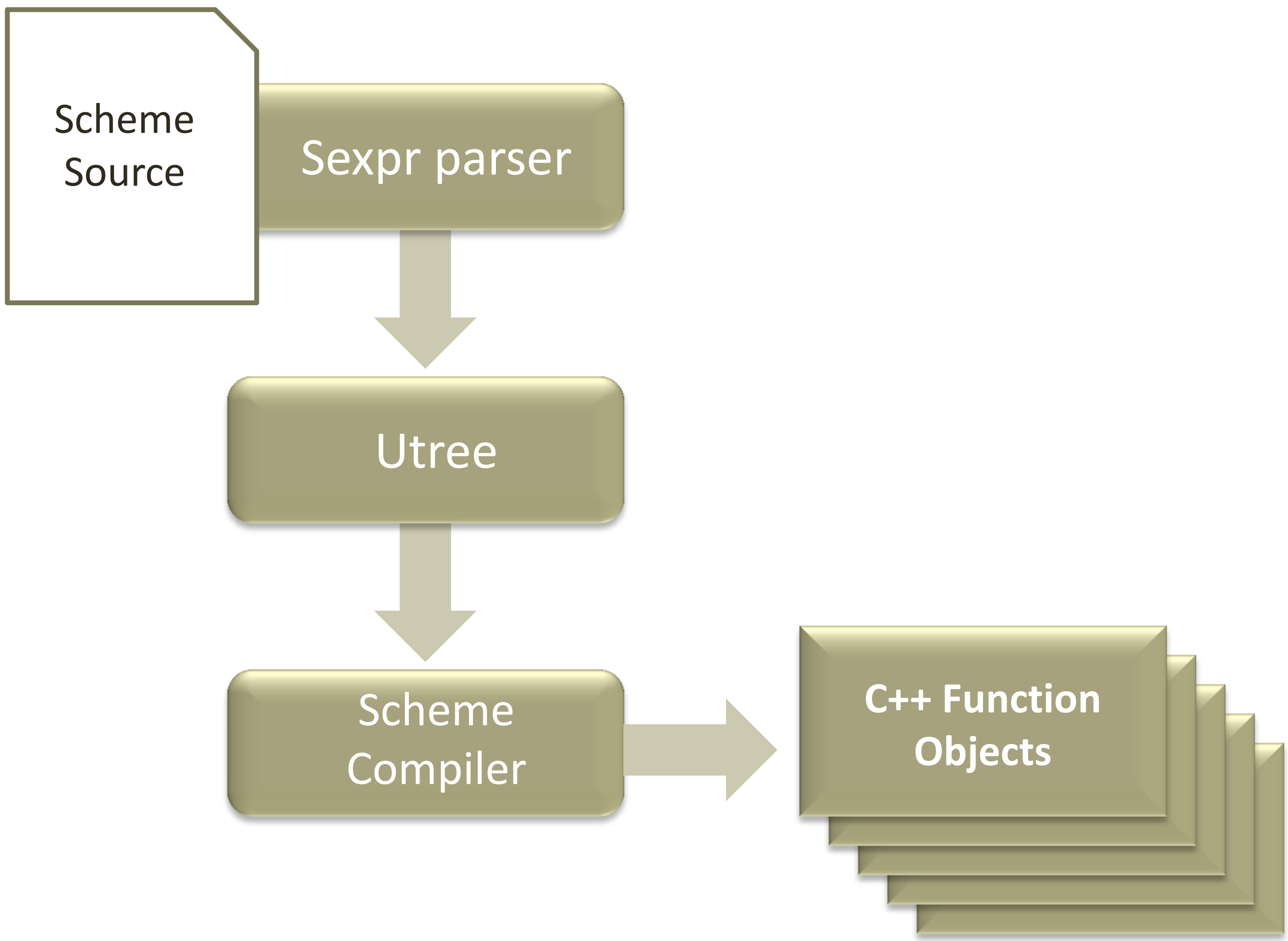
```
variant<double, char const*> v(1.0);
name = karma::string | karma::double_;

<name>
  <try>
    <attributes>[1.0]</attributes>
  </try>
  <success>
    <result>1.0</result>
  </success>
</name>
```

Qi Compiler

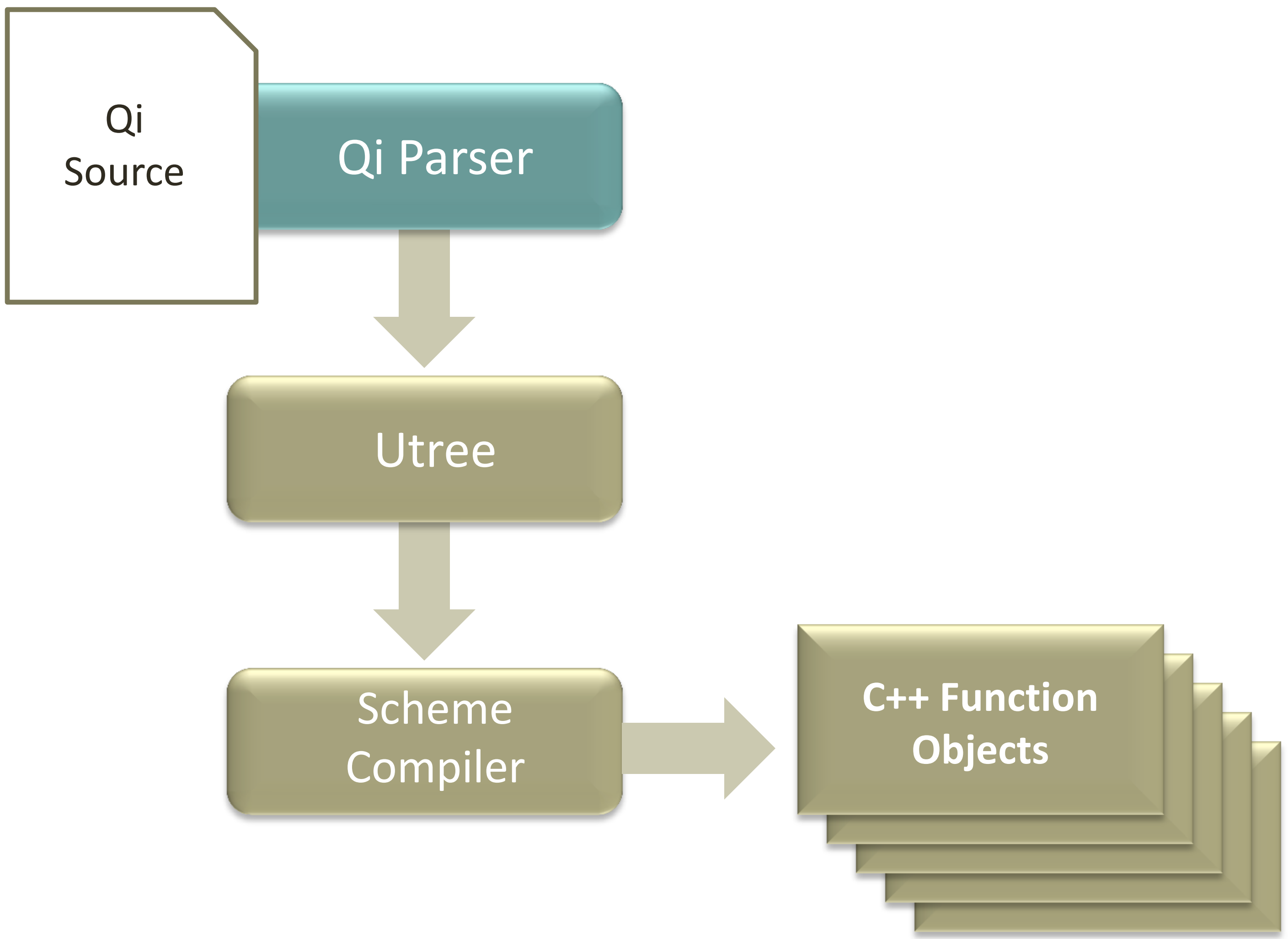


int_ >> ',' >> int_
Qi Parser
int_ >> ',' >> int_
(qi:>>
((qi:int_) (qi:lit ",") (qi:int_)))
Qi Generator
Qi Compiler
Scheme Compiler/
Interpreter
U-Tree
S-Expr Generator
(define (f x) x)
S-Expr Parser
(define (f x) x)

Qi Compiler

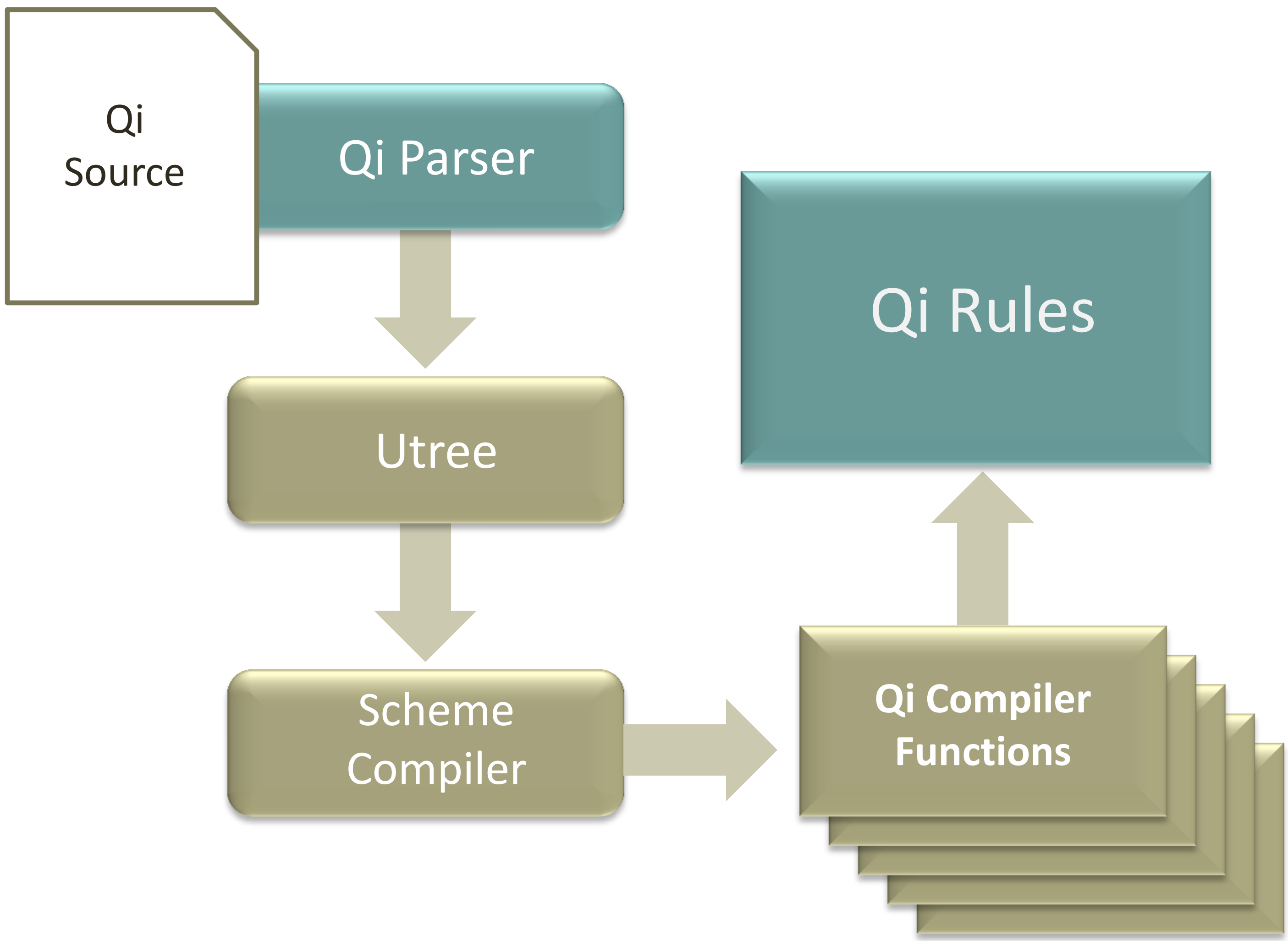

Scheme
Source
Sexpr parser
Utree
Scheme
Compiler
C++ Function
Objects

Qi Compiler



Qi
Source
Qi Parser
Utree
Scheme
Compiler
C++ Function
Objects

Qi Compiler


Qi
Source
Qi Parser
Utree
Scheme
Compiler
Qi Rules
Qi Compiler
Functions

Qi Source (Calculator)

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

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

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

Scheme Source (Calculator)

```
(define expression) ; forward declaration

(define factor
    (qi:|
        (qi:int_)
        (qi:>> (qi:lit "(") (expression) (qi:lit ")"))
        (qi:>> (qi:lit "-") (factor))
        (qi:>> (qi:lit "+") (factor))))

(define term
    (qi:>> (factor)
        (qi:*
            (qi:|
                (qi:>> (qi:lit "*") (factor))
                (qi:>> (qi:lit "/") (factor))))))

(define expression
    (qi:>> (term)
        (qi:*
            (qi:|
                (qi:>> (qi:lit "+") (term))
                (qi:>> (qi:lit "-") (term))))))
```

C++ Driver Code (Calculator)

```
using scheme::interpreter;
using scheme::environment;
using scheme::qi::build_environment;
using scheme::qi::rule_fragments;
using scheme::qi::rule_type;

environment env;
rule_fragments<rule_type> fragments;
build_environment(fragments, env);

interpreter parser(in, filename, &env);
rule_type calc = fragments[parser["expression"]()].alias();
```

Conclusions

- Programs = Data Structures + Algorithms + Glue
  - STL: Iterators
  - Here: Template specialization (full and partial)
- C++ is a multi-paradigm language
  - Pure compile-time
  - Pure run-time
  - Code sitting on the fence
- Scheme is cool
  - Seamlessly integrates with C++, while extending the functional repertoire of the C++ programmer
  - The more ‘run-time’ it gets, the more ‘dynamic‘ the code has to be (type erasure, type-less expressions)