Spirit Repository 0.2

Joel de Guzman

Hartmut Kaiser

Copyright © 2001-2011 Joel de Guzman, Hartmut Kaiser

Table of Contents

Preface

The Spirit Repository

The Spirit repository is a community effort collecting different reusable components (primitives, directives, grammars, etc.) for *Spirit.Qi* parsers and *Spirit.Karma* generators. All components in the repository have been peer reviewed and/or discussed on the Spirit General List. The aim is to broaden the scope of the Spirit library while being able to maintain its high standards in terms of code quality, flexibility, and maintainability. At the same time it encourages people to contribute even small components as it lessens the hurdle of becoming a Boost and Spirit author.

Maintainability of the code and author commitment over a longer period of time are crucial for *Spirit's* success (as it is for the whole Boost project). That allows the repository to play an additional role in terms of being a proving ground for interested authors. It is a lot easier to remove code from the repository than from the *Spirit* core library. So if an author can't maintain his/her contribution anymore for any reason, we are able to remove those parts from the repository more easily.

The repository is an excellent way of evolving things. The *Spirit* core has been developed for years, so we have a certain confidence of it being properly designed and exposing a proven API. On the other hand, new ideas often need some time to 'come to the point'. Changing API's is part of this business. At the same time changing API's always mean user disruption, which we want to keep to a minimum. Again, changing things in the repository is ought to be a lot easier than in the core library.

The quality of contributions is another key to success. That includes not only the code itself, but takes into consideration such things as documentation, tests, examples. The authors activity on the mailing list is related as well, it's an important point. Only well supported things will evolve over time into usable, high quality components. The mandatory discussions and the review of contributions on the Spirit General List ensure the targeted high quality standards.

Based on the user feedback and general usability of things it is possible over time to move repository components/grammars into the *Spirit* core library.

How to use this manual

Some icons are used to mark certain topics indicative of their relevance. These icons precede some text to indicate:

Table 1. Icons

Icon	Name	Meaning
	Note	Generally useful information (an aside that doesn't fit in the flow of the text)
	Tip	Suggestion on how to do something (especially something that not be obvious)
	Important	Important note on something to take particular notice of
	Caution	Take special care with this - it may not be what you expect and may cause bad results
	Danger	This is likely to cause serious trouble if ignored

This documentation is automatically generated by Boost QuickBook documentation tool. QuickBook can be found in the Boost Tools.

Support

Please direct all questions to Spirit's mailing list. You can subscribe to the Spirit General List. The mailing list has a searchable archive. A search link to this archive is provided in Spirit's home page. You may also read and post messages to the mailing list through Spirit General NNTP news portal (thanks to Gmane). The news group mirrors the mailing list. Here is a link to the archives: http://news.gmane.org/gmane.comp.parsers.spirit.general.

What's New

Spirit V2.5.1

This section is added since V2.5.1 (Boost V1.48.0).

New Features in Qi

- Added the *Spirit.Qi* directive `seek[]` enabling skipping over the input, getting to the interested portion.

Qi Components

Qi Parser Primitives

Qi advance Parser

Description

The *Spirit.Qi* `advance` is a primitive parser component allowing the parser to skip (advance) through a specified number of iterations without performing unnecessary work:

```
advance(distance)
```

The most obvious existing alternative to this, the `repeat` directive, will cause the parser to advance one iterator at a time while usually performing operations at each step. In some cases that work is unnecessary, as in the case where large binary objects are being parsed. Take, for example, the following binary data:

```
00 00 00 01 77 fc b4 51 0a b3 b7 ... 1e 60 70 b6 00 00 01 00
```

If the first 4 bytes are a little-endian 32-bit integer describing the length of the subsequent data, but the data itself is not relevant to parsing, then the repeat directive would cause all of the subsequent 16 MB of data to be consumed one byte at a time while generating page faults or other superfluous I/O. If the value is large, as it is in this case, the parser becomes very slow.

```
little_dword[_a = _1] >> repeat(_a)[byte_] >> little_dword...
```

The `advance` parser component solves this problem by performing as little work as possible to advance the parser's iterator, and will optimize for the case of random-access iterators by advancing directly to the desired relative iterator position.

```
little_dword[_a = _1] >> advance(_a) >> little_dword...
```

Header

```
// forwards to <boost/spirit/repository/home/qi/primitive/advance.hpp>
#include <boost/spirit/repository/include/qi_advance.hpp>
```

Synopsis

```
advance(distance)
```

Parameters

Parameter	Description
'distance'	The distance that the iterator shall be advanced

Attribute

The `advance` component exposes no attribute (the exposed attribute type is `unused_type`):

```
advance --> unused
```

Example

The following example shows simple use cases of the `advance` component. We will illustrate its usage by generating parsers for some binary data (for the full example code see advance.cpp)

Prerequisites

In addition to the main header file needed to include the core components implemented in *Spirit.Qi* we add the header file needed for the new `advance` component.

```
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/spirit/repository/include/qi_advance.hpp>
```

In order to make the examples below more readable we introduce or use the following namespaces.

```
namespace qi = boost::spirit::qi;
using boost::spirit::repository::qi::advance;
```

Setting up the Grammar

This is a very simple grammar that recognizes several fields of a binary stream of data. There are two fields explicitly delimited by a field indicating the number of bytes that are spanned. They are separated by a literal string.

```
template <typename Iterator>
struct advance_grammar : qi::grammar<Iterator, qi::locals<int> >
{
    advance_grammar() : advance_grammar::base_type(start)
    {
        using qi::byte_;
        using qi::eoi;
        using namespace qi::labels;

        start
            =  byte_  [_a = _1]
            >> advance(_a)
            >> "boost"
            >> byte_  [_a = _1]
            >> (advance(_a) | "qi") // note alternative when advance fails
            >> eoi
            ;
    }

    qi::rule<Iterator, qi::locals<int> > start;
};
```

Note that the second binary field may either contain the number of specified bytes, or the word "qi". If the `advance` parser component fails to advance the specified number of bytes before reaching the end of input, it will fail and the parser will attempt to descend into alternatives.

Parsing a Correctly-delimited String of Data

The data below is correctly delimited and will thus result in a valid parse. Note that both random-access and bidirectional iterators are used here.

```
unsigned char const alt1[] =
{
    5,                          // number of bytes to advance
    1, 2, 3, 4, 5,              // data to advance through
    'b', 'o', 'o', 's', 't',    // word to parse
    2,                          // number of bytes to advance
    11, 12                      // more data to advance through
    // eoi
};

std::string const alt1_string(alt1, alt1 + sizeof alt1);
std::list<unsigned char> const alt1_list(alt1, alt1 + sizeof alt1);

result =
    qi::parse(alt1_string.begin(), alt1_string.end()
        , client::advance_grammar<std::string::const_iterator>())
    ? "succeeded" : "failed";
std::cout << "Parsing alt1 using random access iterator " << result << std::endl;

result =
    qi::parse(alt1_list.begin(), alt1_list.end()
        , client::advance_grammar<std::list<unsigned char>::const_iterator>())
    ? "succeeded" : "failed";
std::cout << "Parsing alt1 using bidirectional iterator " << result << std::endl;
```

Parsing the Alternative Representation

The data below is not correctly delimited, but will correctly parse because the alternative word "qi" is available.

```
unsigned char const alt2[] =
{
    2,                          // number of bytes to advance
    1, 2,                       // data to advance through
    'b', 'o', 'o', 's', 't',    // word to parse
    4,                          // number of bytes to advance
    'q', 'i'                    // alternative (advance won't work)
    // eoi
};

std::string const alt2_string(alt2, alt2 + sizeof alt2);
std::list<unsigned char> const alt2_list(alt2, alt2 + sizeof alt2);

result =
    qi::parse(alt2_string.begin(), alt2_string.end()
        , client::advance_grammar<std::string::const_iterator>())
    ? "succeeded" : "failed";
std::cout << "Parsing alt2 using random access iterator " << result << std::endl;

result =
    qi::parse(alt2_list.begin(), alt2_list.end()
        , client::advance_grammar<std::list<unsigned char>::const_iterator>())
    ? "succeeded" : "failed";
std::cout << "Parsing alt2 using bidirectional iterator " << result << std::endl;
```

Notes

The `advance` parser component will fail unconditionally on negative values. It will never attempt to advance the iterator in the reverse direction.

Qi flush_multi_pass parser

Description

The *Spirit.Qi* `flush_multi_pass` parser is a primitive (pseudo) parser component allowing to clear the internal buffer of a `multi_pass` iterator. Clearing the buffer of a `multi_pass` might be beneficial for grammars where it is clear that no backtracking can occur. The general syntax for using the `flush_multi_pass` is:

```
flush_multi_pass
```

which will call the `clear_queue()` member function if the current iterators are of the type `multi_pass`. This will cause any buffered data to be erased. This also will invalidate all other copies of multi_pass and they should not be used. If they are, an `boost::illegal_backtracking` exception will be thrown. For all other iterator types this is a no-op. The `flush_multi_pass` generates a parser component which always succeeds and which does not consume any input (very much like `eps`).

Header

```
// forwards to <boost/spirit/repository/home/qi/primitive/flush_multi_pass.hpp>
#include <boost/spirit/repository/include/qi_flush_multi_pass.hpp>
```

Synopsis

```
flush_multi_pass
```

Parameters

The `flush_multi_pass` does not require any parameters.

Attribute

The `flush_multi_pass` component exposes no attribute (the exposed attribute type is `unused_type`):

```
flush_multi_pass --> unused
```

Example

The following example shows a simple use case of the `flush_multi_pass` parser.

We will illustrate its usage by generating different comment styles and a function prototype (for the full example code see here: flush_multi_pass.cpp)

Prerequisites

In addition to the main header file needed to include the core components implemented in *Spirit.Qi* we add the header file needed for the new `flush_multi_pass` parser.

```
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/repository/include/qi_flush_multi_pass.hpp>
```

To make all the code below more readable we introduce the following namespaces.

```
namespace spirit = boost::spirit;
using boost::spirit::repository::flush_multi_pass;
```

Clearing the internal buffer

The example grammar recognizes the (simplified) preprocessor commands #define and #undef both of which are constraint to a single line. This makes it possible to delete all internal iterator buffers on each detected line break. This is safe as no backtracking will occur after any line end. The following code snippet shows the usage of flush_multi_pass for this purpose.

```cpp
template <typename Iterator, typename Skipper>
struct preprocessor : spirit::qi::grammar<Iterator, Skipper>
{
    // This is a simplified preprocessor grammar recognizing
    //
    //   #define MACRONAME something
    //   #undef MACRONAME
    //
    // Its sole purpose is to show an example how to use the
    // flush_multi_pass parser. At the end of each line no backtracking can
    // occur anymore so that it's safe to clear all internal buffers in the
    // multi_pass.
    preprocessor() : preprocessor::base_type(file)
    {
        using spirit::ascii::char_;
        using spirit::qi::eol;
        using spirit::qi::lit;

        file =
            *line
            ;

        line = ( command |  *(char_ - eol) )
            >>  eol
            >>  flush_multi_pass
            ;

        command =
                "#define" >> *lit(' ') >> *(char_ - ' ') >> *lit(' ') >> *(char_ - eol)
            |   "#undef"  >> *lit(' ') >> *(char_ - eol)
            ;
    }

    spirit::qi::rule<Iterator, Skipper> file, line, command;
};
```

> **Note**
>
> Using the flush_multi_pass parser component with iterators other than multi_pass is safe as it has no effect on the parsing.

Qi Parser Directives

Qi Confix Parser Directive

Description

The *Spirit.Qi* confix directive is a unary parser component allowing to embed a parser (the subject) inside an opening (the prefix) and a closing (the suffix):

```cpp
confix(prefix, suffix)[subject]
```

This results in a parser that is equivalent to the sequence

```
omit[prefix] >> subject >> omit[suffix]
```

A simple example is a parser for non-nested comments which can now be written as:

```
confix("/*", "*/")[*(char_ - "*/")]  // C style comment
confix("//", eol)[*(char_ - eol)]     // C++ style comment
```

Using the `confix` directive instead of the explicit sequence has the advantage of being able to encapsulate the prefix and the suffix into a separate construct. The following code snippet illustrates the idea:

```
namespace spirit = boost::spirit;
namespace repo = boost::spirit::repository;

// Define a metafunction allowing to compute the type
// of the confix() construct
template <typename Prefix, typename Suffix = Prefix>
struct confix_spec
{
    typedef typename spirit::result_of::terminal<
        repo::tag::confix(Prefix, Suffix)
    >::type type;
};

confix_spec<std::string>::type const c_comment = repo::confix("/*", "*/");
confix_spec<std::string>::type const cpp_comment = repo::confix("//", "\n");
```

Now, the comment parsers can be written as

```
c_comment[*(char_ - "*/")]     // C style comment
cpp_comment[*(char_ - eol)]     // C++ style comment
```

> **Note**
>
> While the `confix_p(prefix, subject, suffix)` parser in *Spirit.Classic* was equivalent to the sequence `prefix >> *(subject - suffix) >> suffix`, the *Spirit.Qi* confix` directive will not perform this refactoring any more. This simplifies the code and makes things more explicit.

Header

```
// forwards to <boost/spirit/repository/home/qi/directive/confix.hpp>
#include <boost/spirit/repository/include/qi_confix.hpp>
```

Synopsis

```
confix(prefix, suffix)[subject]
```

Parameters

Parameter	Description
`prefix`	The parser for the opening (the prefix).
`suffix`	The parser for the ending (the suffix).
`subject`	The parser for the input sequence between the `prefix` and `suffix` parts.

All three parameters can be arbitrarily complex parsers themselves.

Attribute

The `confix` directive exposes the attribute type of its subject as its own attribute type. If the `subject` does not expose any attribute (the type is `unused_type`), then the `confix` does not expose any attribute either.

```
a: A, p: P, s: S: --> confix(p, s)[a]: A
```

> **Note**
>
> This notation is used all over the Spirit documentation and reads as: Given, `a`, `p`, and `s` are parsers, and `A`, `P`, and `S` are the types of their attributes, then the type of the attribute exposed by `confix(p, s)[a]` will be `A`.

Example

The following example shows simple use cases of the `confix` directive. We will illustrate its usage by generating parsers for different comment styles and for some simple tagged data (for the full example code see confix.cpp)

Prerequisites

In addition to the main header file needed to include the core components implemented in *Spirit.Qi* we add the header file needed for the new `confix` directive.

```
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/repository/include/qi_confix.hpp>
```

In order to make the examples below more readable we import a number of elements into the current namespace:

```
using boost::spirit::eol;
using boost::spirit::lexeme;
using boost::spirit::ascii::alnum;
using boost::spirit::ascii::char_;
using boost::spirit::ascii::space;
using boost::spirit::qi::parse;
using boost::spirit::qi::phrase_parse;
using boost::spirit::repository::confix;
```

Parsing Different Comment Styles

We will show how to parse different comment styles. First we will parse a C++ comment:

```
template <typename Iterator>
bool parse_cpp_comment(Iterator first, Iterator last, std::string& attr)
{
    bool r = parse(first, last,
        confix("//", eol)[*(char_ - eol)],    // grammar
        attr);                                // attribute

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

This function will obviously parse input such as "// This is a comment \n ". Similarily parsing a 'C'-style comment proves to be straightforward:

```
template <typename Iterator>
bool parse_c_comment(Iterator first, Iterator last, std::string& attr)
{
    bool r = parse(first, last,
        confix("/*", "*/")[*(char_ - "*/")],    // grammar
        attr);                                  // attribute

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

which again will be able to parse e.g. "/* This is a comment */ ".

Parsing Tagged Data

Generating a parser that extracts the body from the HTML snippet "The Body" is not very hard, either:

```
template <typename Iterator>
bool parse_tagged(Iterator first, Iterator last, std::string& attr)
{
    bool r = phrase_parse(first, last,
        confix("<b>", "</b>")[lexeme[*(char_ - '<')]],  // grammar
        space,                                          // skip
        attr);                                          // attribute

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

Qi Distinct Parser Directive

Description

The *Spirit.Qi* distinct parser is a directive component allowing to avoid partial matches while parsing using a skipper. A simple example is the common task of matching a C keyword. Consider:

```
"description" >> -lit(":") >> *(char_ - eol)
```

intended to match a line in a configuration file. Let's assume further, that this rule is used with a space skipper and that we have the following strings in the input:

```
"description: ident\n"
"description ident\n"
"descriptionident\n"
```

It might seem unexpected, but the parser above matches all three inputs just fine, even if the third input should not match at all! In order to avoid the unwanted match we are forced to make our rule more complicated:

```
lexeme["description" >> !char_("a-zA-Z_0-9")] >> -lit(":") >> *(char_ - eol)
```

(the rule reads as: match `"description"` as long as it's not *directly* followed by a valid identifier).

The `distinct[]` directive is meant to simplify the rule above:

```
distinct(char_("a-zA-Z_0-9"))["description"] >> -lit(":") >> *(char_ - eol)
```

Using the `distinct[]` component instead of the explicit sequence has the advantage of being able to encapsulate the tail (i.e the `char_("a-zA-Z_0-9")`) as a separate parser construct. The following code snippet illustrates the idea (for the full code of this example please see distinct.cpp):

```cpp
namespace spirit = boost::spirit;
namespace ascii = boost::spirit::ascii;
namespace repo = boost::spirit::repository;

// Define metafunctions allowing to compute the type of the distinct()
// and ascii::char_() constructs
namespace traits
{
    // Metafunction allowing to get the type of any repository::distinct(...)
    // construct
    template <typename Tail>
    struct distinct_spec
      : spirit::result_of::terminal<repo::tag::distinct(Tail)>
    {};

    // Metafunction allowing to get the type of any ascii::char_(...) construct
    template <typename String>
    struct char_spec
      : spirit::result_of::terminal<spirit::tag::ascii::char_(String)>
    {};
};

// Define a helper function allowing to create a distinct() construct from
// an arbitrary tail parser
template <typename Tail>
inline typename traits::distinct_spec<Tail>::type
distinct_spec(Tail const& tail)
{
    return repo::distinct(tail);
}

// Define a helper function allowing to create a ascii::char_() construct
// from an arbitrary string representation
template <typename String>
inline typename traits::char_spec<String>::type
char_spec(String const& str)
{
    return ascii::char_(str);
}
```

```
// the following constructs the type of a distinct_spec holding a
// charset("0-9a-zA-Z_") as its tail parser
typedef traits::char_spec<std::string>::type charset_tag_type;
typedef traits::distinct_spec<charset_tag_type>::type keyword_tag_type;

// Define a new Qi 'keyword' directive usable as a shortcut for a
// repository::distinct(char_(std::string("0-9a-zA-Z_")))
std::string const keyword_spec("0-9a-zA-Z_");
keyword_tag_type const keyword = distinct_spec(char_spec(keyword_spec));
```

These definitions define a new Qi parser recognizing keywords! This allows to rewrite our declaration parser expression as:

```
keyword["description"] >> -lit(":") >> *(char_ - eol)
```

which is much more readable and concise if compared to the original parser expression. In addition the new `keyword[]` directive has the advantage to be usable for wrapping any parser expression, not only strings as in the example above.

Header

```
// forwards to <boost/spirit/repository/home/qi/directive/distinct.hpp>
#include <boost/spirit/repository/include/qi_distinct.hpp>
```

Synopsis

```
distinct(tail)[subject]
```

Parameters

Parameter	Description
tail	The parser construct specifying what whould not follow the subject in order to match the overall expression.
subject	The parser construct to use to match the current input. The distinct directive makes sure that no unexpected partial matches occur.

All two parameters can be arbitrary complex parsers themselves.

Attribute

The `distinct` component exposes the attribute type of its subject as its own attribute type. If the `subject` does not expose any attribute (the type is `unused_type`), then the `distinct` does not expose any attribute either.

```
a: A, b: B --> distinct(b)[a]: A
```

Example

The following example shows simple use cases of the `distinct` parser. distinct.cpp)

Prerequisites

In addition to the main header file needed to include the core components implemented in *Spirit.Qi* we add the header file needed for the new `distinct` generator.

```
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/repository/include/qi_distinct.hpp>
```

To make all the code below more readable we introduce the following namespaces.

```
using namespace boost::spirit;
using namespace boost::spirit::ascii;
using boost::spirit::repository::distinct;
```

Using The Distinct Directive to Match keywords

We show several examples of how the distinct[] directive can be used to force correct behavior while matching keywords. The first two code snippets show the correct matching of the description keyword (in this hypothetical example we allow keywords to be directly followed by an optional "--"):

```
{
    std::string str("description ident");
    std::string::iterator first(str.begin());
    bool r = qi::phrase_parse(first, str.end()
      , distinct(alnum | '_')["description"] >> -lit("--") >> +(alnum | '_')
      , space);
    BOOST_ASSERT(r && first == str.end());
}
```

```
{
    std::string str("description--ident");
    std::string::iterator first(str.begin());
    bool r = qi::phrase_parse(first, str.end()
      , distinct(alnum | '_')["description"] >> -lit("--") >> +(alnum | '_')
      , space);
    BOOST_ASSERT(r && first == str.end());
}
```

The last example shows that the distinct[] parser component correctly refuses to match "description-ident":

```
{
    std::string str("description-ident");
    std::string::iterator first(str.begin());
    bool r = qi::phrase_parse(first, str.end()
      , distinct(alnum | '_')["description"] >> -lit("--") >> +(alnum | '_')
      , space);
    BOOST_ASSERT(!r && first == str.begin());
}
```

Qi Keyword Parser Directive

Description

The kwd[] and ikwd[] provide a powerful and flexible mechanism for parsing keyword based input. It works in conjuction with the / operator to create an effective keyword parsing loop. The keyword parsing loop doesn't require the keywords to appear in a defined order in the input but also provides the possibility to check how many times a keyword appears in the input.

The kwd directive will parse the keywords respecting case sensitivity whereas the ikwd direcive is case insensitive. You can mix the kwd and ikwd directives inside a set of keywords, but be aware that this has a small overhead. It should be prefered not to mix the kwd and ikwd directives.

The kwd directive is very similar to the repeat directive in that it enables to enforce keyword occurence constraints but also provides very interesting speed improvement over the pure EBNF syntax or the Nabialek-Trick.

Header

```
// forwards to <boost/spirit/repository/home/qi/directive/kwd.hpp>
#include <boost/spirit/repository/include/qi_kwd.hpp>
```

Synopsis

Expression	Semantics
`kwd(keyword)[subject]`	Parse (`"keyword"` > `subject`) zero or more times.
`kwd(keyword,n)[subject]`	Parse (`"keyword"` > `subject`) exactly n times.
`kwd(keyword,min, max)[subject]`	Parse (`"keyword"` > `subject`) at least min times and at most max times.
`kwd(keyword,min, inf)[subject]`	Parse (`"keyword"` > `subject`) at least min or more.

For non case sensitive keywords use the ikwd directive.

Parameters

Parameter	Description
`keyword`	The parser for the opening (the prefix).
`subject`	The parser for the input sequence following the keyword part.
`n`	Int representing the exact number of times the keyword must be repeated.
`min`	Int representing the minimum number of times the keyword must be repeated.
`max`	Int representing the maximum number of times the keyword must be repeated.

All three parameters can be arbitrarily complex parsers themselves.

Attributes

Expression	Attribute
`kwd(k1)[a]`	`a: A --> kwd(k1)[a]: optional<A> or vector<A>` `a: Unused --> kwd(k1)[a]: Unused`

`kwd(k1,n)[a]`	`a: A --> kwd(k1,n)[a]: optional<A> or vec↵`
	`tor<A>`
	`a: Unused --> kwd(k1,n)[a]: Unused`

`kwd(k1,min, max)[a]`	`a: A --> kwd(k1,min, max)[a]: option↵`
	`al<A> or vector<A>`
	`a: Unused --> kwd(k1,min, max)[a]: Unused`

`kwd(k1,min, inf)[a]`	`a: A --> kwd(k1,min, inf)[a]: option↵`
	`al<A> or vector<A>`
	`a: Unused --> kwd(k1,min, inf)[a]: Unused`

Complexity

The overall complexity is defined by the complexity of its subject parser. The complexity of the keyword list construct `kwd` itself is O(N), where N is the number of repetitions executed.

The complexity of the keyword list itself determined by the complexity of the internal TST contents :

$O(\log n + k)$

Where k is the length of the string to be searched in a TST with n strings.

Example

Please refer to keyword_list.

Qi Seek Parser Directive

Description

The `seek[]` parser-directive skips all input until the subject parser matches. On failure, the current position is the end of the input.

Header

```
// forwards to <boost/spirit/repository/home/qi/directive/seek.hpp>
#include <boost/spirit/repository/include/qi_seek.hpp>
```

Also, see Include Structure.

Namespace

Name
`boost::spirit::repository::qi::seek`

Model of

> `UnaryParser`

Notation

a   A `Parser`.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `UnaryParser`.

Expression	Semantics
`seek[a]`	Advances until the parser `a` matches.

Attributes

See Compound Attribute Notation.

Expression	Attribute
`seek[a]`	`a: A --> seek[a]: A` `a: Unused --> seek[a]: Unused`

Complexity

> The overall complexity is defined by the complexity of its subject parser. The complexity of `seek` itself is O(N), where N is the number of unsuccessful matches.

> **Note**
>
> **seeking sequence with skipping**
>
> Using `seek[a >> b]` with skipping is inefficient, because when sequence fails, the backtracked position is non-skipped. The solution is to ensure the input will always be pre-skipped, for example:
>
> ```
> seek[lexeme[skip[a >> b]]]
> ```
>
> does the trick.

Example

The following example shows a simple use case of the `seek[]` directive, parsing C-style comment. (For the full source of the example, see seek.cpp)

Some namespace aliases:

```
namespace qi = boost::spirit::qi;
namespace repo = boost::spirit::repository;
```

The input string and its iterators:

```
std::string str("/*C-style comment*/");
iterator it = str.begin();
iterator end = str.end();
```

Parsing and showing the result:

```
if (qi::parse(it, end, "/*" >> repo::qi::seek["*/"]))
{
    std::cout << "------------------------------ \n";
    std::cout << "Parsing succeeded.\n";
    std::cout << "------------------------------\n";
}
else
{
    std::cout << "------------------------------ \n";
    std::cout << "Unterminated /* comment.\n";
    std::cout << "------------------------------ \n";
}
```

Qi Parser Non-terminals

Qi subrules

Description

The *Spirit.Qi* `subrule` is a component allowing to create a named parser, and to refer to it by name -- much like rules and grammars. It is in fact a fully static version of the rule.

The strength of subrules is performance. Replacing some rules with subrules can make a parser slightly faster (see Performance below for measurements). The reason is that subrules allow aggressive inlining by the C++ compiler, whereas the implementation of rules is based on a virtual function call which, depending on the compiler, can have some run-time overhead and stop inlining.

The weaknesses of subrules are:

- subrules can only be defined and used within the same parser expression. A subrule cannot be defined at one location, and then used in another location.

- subrules put a massive strain on the C++ compiler. They increase compile times and memory usage during compilation, and also increase the risk of hitting compiler limits and/or bugs.

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

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

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

The example above can be found here: ../../example/qi/calc1_sr.cpp

As shown in this code snippet (an extract from the calc1_sr example), subrules can be freely mixed with rules and grammars. Here, a group of 3 subrules (`expression`, `term`, `factor`) is assigned to a rule (named `entry`). This means that parts of a parser can use subrules (typically the innermost, most performance-critical parts), whereas the rest can use rules and grammars.

Header

```
// forwards to <boost/spirit/repository/home/qi/nonterminal/subrule.hpp>
#include <boost/spirit/repository/include/qi_subrule.hpp>
```

Synopsis (declaration)

```
subrule<ID, A1, A2> sr(name);
```

Parameters (declaration)

Parameter	Description
ID	Required numeric argument. Gives the subrule a unique 'identification tag'.
A1, A2	Optional types, can be specified in any order. Can be one of 1. signature, 2. locals (see rules reference for more information on those parameters). Note that the skipper type need not be specified in the parameters, unlike with grammars and rules. Subrules will automatically use the skipper type which is in effect when they are invoked.
name	Optional string. Gives the subrule a name, useful for debugging and error handling.

Synopsis (usage)

Subrules are defined and used within groups, typically (and by convention) enclosed inside parentheses.

```
// Group containing N subrules
(
    sr1 = expr1
  , sr2 = expr2
  , ... // Any number of subrules
}
```

The IDs of all subrules defined within the same group must be different. It is an error to define several subrules with the same ID (or to define the same subrule multiple times) in the same group.

```
// Auto-subrules and inherited attributes
(
    srA %= exprA >> srB >> srC(c1, c2, ...) // Arguments to subrule srC
  , srB %= exprB
  , srC  = exprC
  , ...
)(a1, a2, ...)            // Arguments to group, i.e. to start subrule srA
```

Parameters (usage)

Parameter	Description
sr1, sr2	Subrules with different IDs.
expr1, expr2	Parser expressions. Can include sr1 and sr2, as well as any other valid parser expressions.
srA	Subrule with a synthesized attribute and inherited attributes.
srB	Subrule with a synthesized attribute.
srC	Subrule with inherited attributes.
exprA, exprB, exprC	Parser expressions.
a1, a2	Arguments passed to the subrule group. They are passed as inherited attributes to the group's start subrule, srA.
c1, c2	Arguments passed as inherited attributes to subrule srC.

Groups

A subrule group (a set of subrule definitions) is a parser, which can be used anywhere in a parser expression (in assignments to rules, as well as directly in arguments to functions such as parse). In a group, parsing proceeds from the start subrule, which is the first (topmost) subrule defined in that group. In the two groups in the synopsis above, sr1 and srA are the start subrules respectively -- for example when the first subrule group is called forth, the sr1 subrule is called.

A subrule can only be used in a group which defines it. Groups can be viewed as scopes: a definition of a subrule is limited to its enclosing group.

```
rule<char const*> r1, r2, r3;
subrule<1> sr1;
subrule<2> sr2;

r1 =
        ( sr1 = 'a' >> int_ )        // First group in r1.
    >>  ( sr2 = +sr1 )               // Second group in r1.
    //          ^^^
    // DOES NOT COMPILE: sr1 is not defined in this
    // second group, it cannot be used here (its
    // previous definition is out of scope).
;

r2 =
        ( sr1 = 'a' >> int_ )        // Only group in r2.
    >>  sr1
    //  ^^^
    // DOES NOT COMPILE: not in a subrule group,
    // sr1 cannot be used here (here too, its
    // previous definition is out of scope).
;

r3 =
        ( sr1 = 'x' >> double_ )     // Another group. The same subrule `sr1`
                                     // can have another, independent
                                     // definition in this group.
;
```

Attributes

A subrule has the same behavior as a rule with respect to attributes. In particular:

- the type of its synthesized attribute is the one specified in the subrule's signature, if any. Otherwise it is `unused_type`.

- the types of its inherited attributes are the ones specified in the subrule's signature, if any. Otherwise the subrule has no inherited attributes.

- an auto-subrule can be defined by assigning it with the `%=` syntax. In this case, the RHS parser's attribute is automatically propagated to the subrule's synthesized attribute.

- the Phoenix placeholders `_val`, `_r1`, `_r2`, ... are available to refer to the subrule's synthesized and inherited attributes, if present.

Locals

A subrule has the same behavior as a rule with respect to locals. In particular, the Phoenix placeholders `_a`, `_b`, ... are available to refer to the subrule's locals, if present.

Example

Some includes:

```
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/repository/include/qi_subrule.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
```

Some using declarations:

```cpp
namespace qi = boost::spirit::qi;
namespace repo = boost::spirit::repository;
namespace ascii = boost::spirit::ascii;
```

A grammar containing only one rule, defined with a group of 5 subrules:

```cpp
template <typename Iterator>
struct mini_xml_grammar
  : qi::grammar<Iterator, mini_xml(), ascii::space_type>
{
    mini_xml_grammar()
      : mini_xml_grammar::base_type(entry)
    {
        using qi::lit;
        using qi::lexeme;
        using ascii::char_;
        using ascii::string;
        using namespace qi::labels;

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

          , node %= xml | text

          , text %= lexeme[+(char_ - '<')]

          , start_tag %=
                    '<'
                >>  !lit('/')
                >>  lexeme[+(char_ - '>')]
                >>  '>'

          , end_tag %=
                    "</"
                >>  string(_r1)
                >>  '>'
        );
    }

    qi::rule<Iterator, mini_xml(), ascii::space_type> entry;

    repo::qi::subrule<0, mini_xml(), qi::locals<std::string> > xml;
    repo::qi::subrule<1, mini_xml_node()> node;
    repo::qi::subrule<2, std::string()> text;
    repo::qi::subrule<3, std::string()> start_tag;
    repo::qi::subrule<4, void(std::string)> end_tag;
};
```

The definitions of the `mini_xml` and `mini_xml_node` data structures are not shown here. The full example above can be found here: ../../example/qi/mini_xml2_sr.cpp

Performance

This table compares run-time and compile-time performance when converting examples to subrules, with various compilers.

Table 2. Subrules performance

Example	Compiler	Speed (run-time)	Time (compile-time)	Memory (compile-time)
calc1_sr	gcc 4.4.1	+6%	n/a	n/a
calc1_sr	Visual C++ 2008 (VC9)	+5%	n/a	n/a
mini_xml2_sr	gcc 3.4.6	-1%	+54%	+32%
mini_xml2_sr	gcc 4.1.2	+5%	+58%	+25%
mini_xml2_sr	gcc 4.4.1	+8%	+20%	+14%
mini_xml2_sr	Visual C++ 2005 (VC8) SP1	+1%	+33%	+27%
mini_xml2_sr	Visual C++ 2008 (VC9)	+9%	+52%	+40%

The columns are:

- **Speed (run-time)**: speed-up of the parser resulting from the use of subrules (higher is better).

- **Time (compile-time)**: increase in compile time (lower is better).

- **Memory (compile-time)**: increase in compiler memory usage (lower is better).

Notes

Subrules push the C++ compiler hard. A group of subrules is a single C++ expression. Current C++ compilers cannot handle very complex expressions very well. One restricting factor is the typical compiler's limit on template recursion depth. Some, but not all, compilers allow this limit to be configured.

g++'s maximum can be set using a compiler flag: `-ftemplate-depth`. Set this appropriately if you use relatively complex subrules.

Qi Parser Operators

Keyword List Operator

Description

The keyword list operator, `kwd("k1")[a] / kwd("k2")[b]`, works tightly with the kwd, ikwd directives to effeciently match keyword lists. As long as one of the keywords specified through the kwd or ikwd directive matches, the keyword will be immediatly followed by the the keyword's associated subject parser. The parser will continue parsing input as long as the one of the keywords and it's associated parser succeed. Writing : (kwd("k1")[a] / kwd("k2")[b] / ...) is equivalent to: *("k1" > a | "k2" > b ...).

Header

```
// forwards to <boost/spirit/repository/home/qi/operator/keywords.hpp>
#include <boost/spirit/repository/include/qi_keywords.hpp>
```

Expression Semantics

Expression	Semantics
`kwd(k1)[a] / kwd(k2)[b]`	Match `lit(k1) > a` or `lit(k2) > b`, equivalent to `lit(k1) > a	lit(k2) > b`

Attributes

Expression	Attribute

`kwd("k1")[a] / kwd("k2")[b]`	```a: A, b: B --> (kwd(k1)[a] / kwd(k2)[b]): tuple<A, B>```
	```a: A, b: Un↵```
	```used --> (kwd(k1)[a] / kwd(k2)[b]): option↵```
	```al<A>```
	```a: Un↵```
	```used, b: B --> (kwd("k1")[a] / kwd(k2)[b]): op↵```
	```tional<B>```
	```a: Unused, b: Un↵```
	```used --> (kwd(k1)[a] / kwd(k2)[b]): Unused```
	
	```a: A, b: A --```
	```>(kwd(k1)[a] / kwd(k2)[b]): tuple<A, A>```

> **Note**
>
> The keyword list parser works tightly with the kwd and ikwd directives and can't be used without it. A compile time error will warn you of any mistakes. This parser collects all the kwd directives and extracts the keyword literals from the directives to internaly build a Ternary Search Tree (TST) to effectively parse the keywords. Because you we can't mix character types inside a TST you must take care not to mix wide strings with normal strings in the keyword you supply to a keyword list. Should it happen the compiler will trap the mistake for you.

> **Note**
>
> The kwd directive also works a bit like the repeat directive and can be used to formulate additional contraints on the number of times a keyword can occur while parsing a keyword list.

> **Note**
>
> The kwd and ikwd directives can be mixed inside a keyword list. This has however a small overhead and should be avoided when possible.

Complexity

The overall complexity of the keyword list parser is defined by the sum of the complexities of its elements. The complexity of the keyword list itself is determined by the complexity of the internal TST contents :

O(log n+k)

Where k is the length of the string to be searched in a TST with n strings.

Example

> **Note**
>
> The test harness for the example(s) below is presented in the Basics Examples section.

Declare a small data structure representing a person:

```cpp
// Data structure definitions to test the kwd directive
// and the keywords list operator

struct person {
    std::string name;
    int age;
    double size;
    std::vector<std::string> favorite_colors;

};

std::ostream &operator<<(std::ostream &os, const person &p)
{
    os<<"Person : "<<p.name<<", "<<p.age<<", "<<p.size<<std::endl;
    std::copy(p.favorite_colors.begin(),p.favorite_colors.end(),std::ostream_iterat↵
or<std::string>(os,"\n"));
    return os;
}

BOOST_FUSION_ADAPT_STRUCT( person,
    (std::string, name)
    (int, age)
    (double, size)
    (std::vector<std::string>, favorite_colors)
)
```

Some using declarations:

```cpp
using boost::spirit::repository::qi::kwd;
using boost::spirit::qi::inf;
using boost::spirit::ascii::space_type;
using boost::spirit::ascii::char_;
using boost::spirit::qi::double_;
using boost::spirit::qi::int_;
using boost::spirit::qi::rule;
```

Now let's declare a keyword parser:

```cpp
no_constraint_person_rule %=
    kwd("name")['=' > parse_string ]
  / kwd("age")  ['=' > int_]
  / kwd("size")  ['=' > double_ > 'm']
  ;
```

A couple of input string variations run on the same parser:

Parsing a keyword list:

```
// Let's declare a small list of people for which we want to collect information.
person John,Mary,Mike,Hellen,Johny;
test_phrase_parser_attr(
                "name = \"John\" \n age = 10 \n size = 1.69m "
                ,no_constraint_person_rule
                ,John);  // full in orginal order
std::cout<<John;

test_phrase_parser_attr(
                "age = 10 \n size = 1.69m \n name = \"Mary\""
                ,no_constraint_person_rule
                ,Mary);  // keyword oder doesn't matter
std::cout<<Mary;

test_phrase_parser_attr(
                "size = 1.69m \n name = \"Mike\" \n age = 10 "
                ,no_constraint_person_rule
                ,Mike);  // still the same result

std::cout<<Mike;
```

The code above will print:

```
Person : John, 10, 1.69
Person : Mary, 10, 1.69
Person : Mike, 10, 1.69
```

Now let's delcare a parser with some occurence constraints:

The parser definition below uses the kwd directive occurence constraint variants to make sure that the name and age keyword occur only once and allows the favorite color entry to appear 0 or more times.

```
constraint_person_rule %=
    kwd("name",1)                   ['=' > parse_string ]
  / kwd("age"    ,1)                 ['=' > int_]
  / kwd("size"   ,1)                 ['=' > double_ > 'm']
  / kwd("favorite color",0,inf) [ '=' > parse_string ]
  ;
```

And see how it works in these two cases:

```
  // Here all the give constraint are resepected : parsing will succeed.
  test_phrase_parser_attr(
    "name = \"Hellen\" \n age = 10 \n size = 1.80m \n favorite color = \"blue\" \n favorite col↵
or = \"green\" "
    ,constraint_person_rule
    ,Hellen);
  std::cout<<Hellen;

// Parsing this string will fail because the age and size minimum occurence requirements aren't ↵
met.
test_phrase_parser_attr(
    "name = \"Johny\"  \n favorite color = \"blue\" \n favorite color = \"green\" "
    ,constraint_person_rule
    ,Johny );
```

Parsing the first string will succeed but fail for the second string as the occurence constraints aren't met. This code should print:

```
Person : Hellen, 10, 1.8
blue
green
```

Karma Components

Karma Generator Directives

Karma Confix Generator

Description

The *Spirit.Karma* `confix` generator is a generator directive component allowing to embed any generated ouput inside an opening (a prefix) and a closing (a suffix). A simple example is a C comment: `/* This is a C comment */` which can be generated using the `confix` generator as: `confix("/*", "*/")["This is a C comment"]`. The general syntax for using the `confix` is:

```
confix(prefix, suffix)[subject]
```

which results in generating a sequence equivalent to

```
prefix << subject << suffix
```

Using the `confix` component instead of the explicit sequence has the advantage of being able to encapsulate the prefix and the suffix into a separate generator construct. The following code snippet illustrates the idea:

```cpp
// Define a metafunction allowing to compute the type of the confix()
// construct
namespace traits
{
    using namespace boost::spirit;

    template <typename Prefix, typename Suffix = Prefix>
    struct confix_spec
      : spirit::result_of::terminal<repository::tag::confix(Prefix, Suffix)>
    {};
};

// Define a helper function allowing to create a confix() construct from
// arbitrary prefix and suffix generators
template <typename Prefix, typename Suffix>
typename traits::confix_spec<Prefix, Suffix>::type
confix_spec(Prefix const& prefix, Suffix const& suffix)
{
    using namespace boost::spirit;
    return repository::confix(prefix, suffix);
}

// Define a helper function to  construct a HTML tag from the tag name
inline typename traits::confix_spec<std::string>::type
tag (std::string const& tagname)
{
    return confix_spec("<" + tagname + ">", "</" + tagname + ">");
}

// Define generators for different HTML tags the HTML tag
typedef traits::confix_spec<std::string>::type ol = tag("ol");      // <ol>...</ol>
typedef traits::confix_spec<std::string>::type li = tag("li");      // <li>...</li>
```

Now, for instance, the above definitions allow to generate the HTML 'ol' tag using a simple: `ol["Some text"]` (which results in `<ol>Some text</ol>`).

Header

```
// forwards to <boost/spirit/repository/home/karma/directive/confix.hpp>
#include <boost/spirit/repository/include/karma_confix.hpp>
```

Synopsis

```
confix(prefix, suffix)[subject]
```

Parameters

Parameter	Description
prefix	The generator construct to use to format the opening (the prefix). The prefix is the part generated *before* any output as generated by the subject.
suffix	The generator construct to use to format the ending (the suffix). The suffix is the part generated *after* any output as generated by the subject.
subject	The generator construct to use to format the actual output in between the prefix and suffix parts.

All three parameters can be arbitrary complex generators themselves.

Attribute

The confix component exposes the attribute type of its subject as its own attribute type. If the subject does not expose any attribute (the type is unused_type), then the confix does not expose any attribute either.

```
a: A --> confix(p, s)[a]: A
```

> **Note**
>
> This notation is used all over the Spirit documentation and reads as: Given, a is generator, and A is the type of the attribute of generator a, then the type of the attribute exposed by confix(p, s)[a] will be A as well.

Example

The following example shows simple use cases of the confix generator. We will illustrate its usage by generating different comment styles and a function prototype (for the full example code see here: confix.cpp)

Prerequisites

In addition to the main header file needed to include the core components implemented in *Spirit.Karma* we add the header file needed for the new confix generator.

```
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/repository/include/karma_confix.hpp>
```

To make all the code below more readable we introduce the following namespaces.

```
using namespace boost::spirit;
using namespace boost::spirit::ascii;
using boost::spirit::repository::confix;
```

Generating Different Comment Styles

We will show how to generate different comment styles. First we will generate a C++ comment:

```
// C++ comment
std::cout <<
    karma::format_delimited(
        confix("//", eol)[string],          // format description
        space,                              // delimiter
        "This is a comment"                 // data
    ) << std::endl;
```

This code snippet will obviouly generate `// This is a comment \n` . Similarily generating a 'C'-style comment proves to be straightforward:

```
// C comment
std::cout <<
    karma::format_delimited(
        confix("/*", "*/")[string],         // format description
        space,                              // delimiter
        "This is a comment"                 // data
    ) << std::endl;
```

which again will generate `/* This is a comment */` .

Generating a Function Prototype

Generating a function prototype given a function name a vector or parameter names is simple as well:

```
// Generate a function call with an arbitrary parameter list
std::vector<std::string> parameters;
parameters.push_back("par1");
parameters.push_back("par2");
parameters.push_back("par3");

std::cout <<
    karma::format(
        string << confix('(', ')')[string % ','],   // format description
        "func",                                       // function name
        parameters                                    // parameter names
    ) << std::endl;
```

which generates the expected output: `func(par1,par2,par3)`.

Karma Generator Non-terminals

Karma subrules

Description

The *Spirit.Karma* `subrule` is a component allowing to create a named generator, and to refer to it by name -- much like rules and grammars. It is in fact a fully static version of the rule.

The strength of subrules is performance. Replacing some rules with subrules can make a generator slightly faster (see Performance below for measurements). The reason is that subrules allow aggressive inlining by the C++ compiler, whereas the implementation of rules is based on a virtual function call which, depending on the compiler, can have some run-time overhead and stop inlining.

The weaknesses of subrules are:

- subrules can only be defined and used within the same generator expression. A subrule cannot be defined at one location, and then used in another location.

- subrules put a massive strain on the C++ compiler. They increase compile times and memory usage during compilation, and also increase the risk of hitting compiler limits and/or bugs.

```
entry %= (
    ast_node %= int_ | binary_node | unary_node

  , binary_node %= '(' << ast_node << char_ << ast_node << ')'

  , unary_node %= '(' << char_ << ast_node << ')'
);
```

The example above can be found here: ../../example/karma/calc2_ast_dump_sr.cpp

As shown in this code snippet (an extract from the calc2_ast_dump_sr example), subrules can be freely mixed with rules and grammars. Here, a group of 3 subrules (`ast_node`, `binary_node`, `unary_node`) is assigned to a rule (named `entry`). This means that parts of a generator can use subrules (typically the innermost, most performance-critical parts), whereas the rest can use rules and grammars.

Header

```
// forwards to <boost/spirit/repository/home/karma/nonterminal/subrule.hpp>
#include <boost/spirit/repository/include/karma_subrule.hpp>
```

Synopsis (declaration)

```
subrule<ID, A1, A2> sr(name);
```

Parameters (declaration)

Parameter	Description
ID	Required numeric argument. Gives the subrule a unique 'identification tag'.
A1, A2	Optional types, can be specified in any order. Can be one of 1. signature, 2. locals (see rules reference for more information on those parameters). Note that the delimiter type need not be specified in the parameters, unlike with grammars and rules. Subrules will automatically use the delimiter type which is in effect when they are invoked.
name	Optional string. Gives the subrule a name, useful for debugging and error handling.

Synopsis (usage)

Subrules are defined and used within groups, typically (and by convention) enclosed inside parentheses.

```
// Group containing N subrules
(
    sr1 = expr1
  , sr2 = expr2
  , ... // Any number of subrules
}
```

The IDs of all subrules defined within the same group must be different. It is an error to define several subrules with the same ID (or to define the same subrule multiple times) in the same group.

```
// Auto-subrules and inherited attributes
(
    srA %= exprA << srB << srC(c1, c2, ...) // Arguments to subrule srC
  , srB %= exprB
  , srC  = exprC
  , ...
)(a1, a2, ...)            // Arguments to group, i.e. to start subrule srA
```

Parameters (usage)

Parameter	Description
sr1, sr2	Subrules with different IDs.
expr1, expr2	Generator expressions. Can include sr1 and sr2, as well as any other valid generator expressions.
srA	Subrule with a synthesized attribute and inherited attributes.
srB	Subrule with a synthesized attribute.
srC	Subrule with inherited attributes.
exprA, exprB, exprC	Generator expressions.
a1, a2	Arguments passed to the subrule group. They are passed as inherited attributes to the group's start subrule, srA.
c1, c2	Arguments passed as inherited attributes to subrule srC.

Groups

A subrule group (a set of subrule definitions) is a generator, which can be used anywhere in a generator expression (in assignments to rules, as well as directly in arguments to functions such as generate). In a group, generation proceeds from the start subrule, which is the first (topmost) subrule defined in that group. In the two groups in the synopsis above, sr1 and srA are the start subrules respectively -- for example when the first subrule group is called forth, the sr1 subrule is called.

A subrule can only be used in a group which defines it. Groups can be viewed as scopes: a definition of a subrule is limited to its enclosing group.

```
rule<outiter_type> r1, r2, r3;
subrule<1> sr1;
subrule<2> sr2;

r1 =
        ( sr1 = 'a' << space )      // First group in r1.
    <<  ( sr2 = +sr1 )              // Second group in r1.
    //            ^^^
    // DOES NOT COMPILE: sr1 is not defined in this
    // second group, it cannot be used here (its
    // previous definition is out of scope).
;

r2 =
        ( sr1 = 'a' << space )      // Only group in r2.
    <<  sr1
    //  ^^^
    // DOES NOT COMPILE: not in a subrule group,
    // sr1 cannot be used here (here too, its
    // previous definition is out of scope).
;

r3 =
        ( sr1 = space << 'x' )      // Another group. The same subrule `sr1`
                                    // can have another, independent
                                    // definition in this group.
;
```

Attributes

A subrule has the same behavior as a rule with respect to attributes. In particular:

- the type of its synthesized attribute is the one specified in the subrule's signature, if any. Otherwise it is `unused_type`.

- the types of its inherited attributes are the ones specified in the subrule's signature, if any. Otherwise the subrule has no inherited attributes.

- an auto-subrule can be defined by assigning it with the `%=` syntax. In this case, the subrule's synthesized attribute is automatically propagated to the RHS generator's attribute.

- the Phoenix placeholders `_val`, `_r1`, `_r2`, ... are available to refer to the subrule's synthesized and inherited attributes, if present.

Locals

A subrule has the same behavior as a rule with respect to locals. In particular, the Phoenix placeholders `_a`, `_b`, ... are available to refer to the subrule's locals, if present.

Example

Some includes:

```
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/repository/include/karma_subrule.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/spirit/include/phoenix_fusion.hpp>
```

Some using declarations:

```
using namespace boost::spirit;
using namespace boost::spirit::ascii;
namespace repo = boost::spirit::repository;
```

A grammar containing only one rule, defined with a group of 2 subrules:

```
template <typename OutputIterator>
struct mini_xml_generator
  : karma::grammar<OutputIterator, mini_xml()>
{
    mini_xml_generator() : mini_xml_generator::base_type(entry)
    {
        entry %= (
            xml =
                    '<'  << string[_1 = at_c<0>(_val)] << '>'
                <<          (*node)[_1 = at_c<1>(_val)]
                <<  "</" << string[_1 = at_c<0>(_val)] << '>'

          , node %= string | xml
        );
    }

    karma::rule<OutputIterator, mini_xml()> entry;

    repo::karma::subrule<0, mini_xml()> xml;
    repo::karma::subrule<1, mini_xml_node()> node;
};
```

The definitions of the `mini_xml` and `mini_xml_node` data structures are not shown here. The full example above can be found here: ../../example/karma/mini_xml_karma_sr.cpp

Performance

For comparison of run-time and compile-time performance when using subrules, please see the Performance section of *Spirit.Qi* subrules (the implementation of *Spirit.Karma* and *Spirit.Qi* subrules is very similar, so performance is very similar too).

Notes

Subrules push the C++ compiler hard. A group of subrules is a single C++ expression. Current C++ compilers cannot handle very complex expressions very well. One restricting factor is the typical compiler's limit on template recursion depth. Some, but not all, compilers allow this limit to be configured.

g++'s maximum can be set using a compiler flag: `-ftemplate-depth`. Set this appropriately if you use relatively complex subrules.

Acknowledgments

The Spirit repository is the result of the contributions of active members of the Spirit community. We would like to express our thanks to all who directly contributed and to everybody directly or indirectly involved in the discussions, which led to the creation of the parser and generator components.

The following people have directly contributed code to this repository:

Aaron Graham wrote the `advance` parser component, which allows the parser to skip (advance) through a specified number of iterations without performing unnecessary work.

Chris Hoeppler submitted the `confix` parser directive allowing to embed a parser (the subject) inside an opening (the prefix) and a closing sequence (the suffix).

Francois Barel contributed the `subrule` parser and `subrule` generator components, allowing to create a named parser or generator, and to refer to it by name. These components are in fact fully static versions of the corresponding `rule` component.

Thomas Bernard contributed the `keyword_list` and `kwd()[]` parser components, allowing to define keyword parsers.