Spirit v1.8.2 User's Guide

Table of Contents

Preface

"Examples of designs that meet most of the criteria for "goodness" (easy to understand, flexible, efficient) are a recursive-descent parser, which is traditional procedural code. Another example is the STL, which is a generic library of containers and algorithms depending crucially on both traditional procedural code and on parametric polymorphism."

Bjarne Stroustrup

History

A decade and a half ago, I wrote my first calculator in Pascal. It is one of my most unforgettable coding experiences. I was amazed how a mutually recursive set of functions can model a grammar specification. In time, the skills I acquired from that academic experience became very practical. Periodically I was tasked to do some parsing. For instance, whenever I need to perform any form of I/O, even in binary, I try to approach the task somewhat formally by writing a grammar using Pascal-like syntax diagrams and then write a corresponding recursive-descent parser. This worked very well.

The arrival of the Internet and the World Wide Web magnified this thousand-fold. At one point I had to write an HTML parser for a Web browser project. I got a recursive-descent HTML parser working based on the W3C formal specifications easily. I was certainly glad that HTML had a formal grammar specification. Because of the influence of the Internet, I then had to do more parsing. RFC specifications were everywhere. SGML, HTML, XML, even email addresses and those seemingly trivial URLs were all formally specified using small EBNF-style grammar specifications. This made me wish for a tool similar to big-time parser generators such as YACC and **ANTLR**, where a parser is built automatically from a grammar specification. Yet, I want it to be extremely small; small enough to fit in my pocket, yet scalable.

It must be able to practically parse simple grammars such as email addresses to moderately complex grammars such as XML and perhaps some small to medium-sized scripting languages. Scalability is a prime goal. You should be able to use it for small tasks such as parsing command lines without incurring a heavy payload, as you do when you are using YACC or PCCTS. Even now that it has evolved and matured to become a multi-module library, true to its original intent, Spirit can still be used for extreme micro-parsing tasks. You only pay for features that you need. The power of Spirit comes from its modularity and extensibility. Instead of giving you a sledgehammer, it gives you the right ingredients to create a sledgehammer easily. For instance, it does not really have a lexer, but you have all the raw ingredients to write one, if you need one.

The result was Spirit. Spirit was a personal project that was conceived when I was doing R&D in Japan. Inspired by the GoF's composite and interpreter patterns, I realized that I can model a recursive-descent parser with hierarchical-object composition of primitives (terminals) and composites (productions). The original version was implemented with run-time polymorphic classes. A parser is generated at run time by feeding in production rule strings such as `"prod ::= {‘A’ | ‘B’} ‘C’;"`A compile function compiled the parser, dynamically creating a hierarchy of objects and linking semantic actions on the fly. A very early text can be found **here**.

The version that we have now is a complete rewrite of the original Spirit parser using expression templates and static polymorphism, inspired by the works of Todd Veldhuizen (" **Expression Templates**", C++ Report, June 1995). Initially, the ***static-Spirit*** version was meant only to replace the core of the original ***dynamic-Spirit***. Dynamic-spirit needed a parser to implement itself anyway. The original employed a hand-coded recursive-descent parser to parse the input grammar specification strings.

After its initial "open-source" debut in May 2001, static-Spirit became a success. At around November 2001, the Spirit website had an activity percentile of 98%, making it the number one parser tool at Source Forge at the time. Not bad for such a niche project such as a parser library. The "static" portion of Spirit was forgotten and static-Spirit simply became Spirit. The framework soon evolved to acquire more dynamic features.

How to use this manual

The Spirit framework is organized in logical modules starting from the core. This documentation provides a user's guide and reference for each module in the framework. A simple and clear code example is worth a hundred lines of documentation; therefore, the user's guide is presented with abundant examples annotated and explained in step-wise manner. The user's guide is based on examples -lots of them.

As much as possible, forward information (i.e. citing a specific piece of information that has not yet been discussed) is avoided in the user's manual portion of each module. In many cases, though, it is unavoidable that advanced but related topics are interspersed with the normal flow of discussion. To alleviate this problem, topics categorized as "advanced" may be skipped at first reading.

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

Icons

✏	**Note**	Information provided is moderately important and should be noted by the reader.
⚠	**Alert**	Information provided is of utmost importance.
🔍	**Detail**	Information provided is auxiliary but will give the reader a deeper insight into a specific topic. May be skipped.
💡	**Tip**	A potentially useful and helpful piece of information.

Support

Please direct all questions to Spirit's mailing list. You can subscribe to the mailing list **here**. 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 an **NNTP news portal** (thanks to **www.gmane.org**). The news group mirrors the mailing list. Here are two links to the archives: via **gmane**, via **geocrawler**.

To my dear daughter Phoenix

Joel de Guzman
September 2002

Introduction

Spirit is an object-oriented recursive-descent parser generator framework implemented using template meta-programming techniques. Expression templates allow us to approximate the syntax of Extended Backus-Normal Form (EBNF) completely in C++.

The Spirit framework enables a target grammar to be written exclusively in C++. Inline EBNF grammar specifications can mix freely with other C++ code and, thanks to the generative power of C++ templates, are immediately executable. In retrospect, conventional compiler-compilers or parser-generators have to perform an additional translation step from the source EBNF code to C or C++ code.

A simple EBNF grammar snippet:

```
    group       ::= '(' expression ')'
    factor      ::= integer | group
    term        ::= factor (('*' factor) | ('/' factor))*
    expression  ::= term (('+' term) | ('-' term))*
```

is approximated using Spirit's facilities as seen in this code snippet:

```
    group       = '(' >> expression >> ')';
    factor      = integer | group;
    term        = factor >> *(('*' >> factor) | ('/' >> factor));
    expression  = term >> *(('+' >> term) | ('-' >> term));
```

Through the magic of expression templates, this is perfectly valid and executable C++ code. The production rule `expression` is in fact an object that has a member function parse that does the work given a source code written in the grammar that we have just declared. Yes, it's a calculator. We shall simplify for now by skipping the type declarations and the definition of the rule `integer` invoked by `factor`. The production rule `expression` in our grammar specification, traditionally called the start symbol, can recognize inputs such as:

```
    12345
    -12345
    +12345
    1 + 2
    1 * 2
    1/2 + 3/4
    1 + 2 + 3 + 4
    1 * 2 * 3 * 4
    (1 + 2) * (3 + 4)
    (-1 + 2) * (3 + -4)
    1 + ((6 * 200) - 20) / 6
    (1 + (2 + (3 + (4 + 5))))
```

Certainly we have done some modifications to the original EBNF syntax. This is done to conform to C++ syntax rules. Most notably we see the abundance of shift `>>` operators. Since there are no 'empty' operators in C++, it is simply not possible to write something like:

```
    a b
```

as seen in math syntax, for example, to mean multiplication or, in our case, as seen in EBNF syntax to mean sequencing (b should follow a). The framework uses the shift `>>` operator instead for this purpose. We take the `>>` operator, with arrows pointing to the right, to mean "is followed by". Thus we write:

```
    a >> b
```

The alternative operator `|` and the parentheses `()` remain as is. The assignment operator `=` is used in place of EBNF's `::=`. Last but not least, the Kleene star `*` which used to be a postfix operator in EBNF becomes a prefix. Instead of:

```
    a* //... in EBNF syntax,
```

we write:

```
    *a //... in Spirit.
```

since there are no postfix stars, "`*`", in C/C++. Finally, we terminate each rule with the ubiquitous semi-colon, "`;`".

Quick Start

Why would you want to use Spirit?

Spirit is designed to be a practical parsing tool. At the very least, the ability to generate a fully-working parser from a formal EBNF specification inlined in C++ significantly reduces development time. While it may be practical to use a full-blown, stand-alone parser such as YACC or ANTLR when we want to develop a computer language such as C or Pascal, it is certainly overkill to bring in the big guns when we wish to write extremely small micro-parsers. At that end of the spectrum, programmers typically approach the job at hand not as a formal parsing task but through ad hoc hacks using primitive tools such as `scanf`. True, there are tools such as regular-expression libraries (such as **boost regex**) or scanners (such as **boost tokenizer**), but these tools 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 prime objective is to make the tool easy to use. When one thinks of a parser generator, the usual reaction is "it must be big and complex with a steep learning curve." Not so. Spirit is designed to be fully scalable. The framework is structured in layers. This permits learning on an as-needed basis, after only learning the minimal core and basic concepts.

For development simplicity and ease in deployment, the entire framework consists of only header files, with no libraries to link against or build. Just put the spirit distribution in your include path, compile and run. Code size? -very tight. In the quick start example that we shall present in a short while, the code size is dominated by the instantiation of the `std::vector` and `std::iostream`.

Trivial Example #1

Create a parser that will parse a floating-point number.

```
    real_p
```

(You've got to admit, that's trivial!) The above code actually generates a Spirit `real_parser` (a built-in parser) which parses a floating point number. Take note that parsers that are meant to be used directly by the user end with "`_p`" in their names as a Spirit convention. Spirit has many pre-defined parsers and consistent naming conventions help you keep from going insane!

Trivial Example #2

Create a parser that will accept a line consisting of two floating-point numbers.

```
    real_p >> real_p
```

Here you see the familiar floating-point numeric parser `real_p` used twice, once for each number. What's that `>>` operator doing in there? Well, they had to be separated by something, and this was chosen as the "followed by" sequence operator. The above program creates a parser from two simpler parsers, glueing them together with the sequence operator. The result is a parser that is a composition of smaller parsers. Whitespace between numbers can implicitly be consumed depending on how the parser is invoked (see below).

Note: when we combine parsers, we end up with a "bigger" parser, But it's still a parser. Parsers can get bigger and bigger, nesting more and more, but whenever you glue two parsers together, you end up with one bigger parser. This is an important concept.

Trivial Example #3

Create a parser that will accept an arbitrary number of floating-point numbers. (Arbitrary means anything from zero to infinity)

```
    *real_p
```

This is like a regular-expression Kleene Star, though the syntax might look a bit odd for a C++ programmer not used to seeing the `*` operator overloaded like this. Actually, if you know regular expressions it may look odd too since the star is **before** the expression it modifies. C'est la vie. Blame it on the fact that we must work with the syntax rules of C++.

Any expression that evaluates to a parser may be used with the Kleene Star. Keep in mind, though, that due to C++ operator precedence rules you may need to put the expression in parentheses for complex expressions. The Kleene Star is also known as a Kleene Closure, but we call it the Star in most places.

Example #4 [A Just Slightly Less Trivial Example]

This example will create a parser that accepts a comma-delimited list of numbers and put the numbers in a vector.

Step 1. Create the parser

```
    real_p >> *(ch_p(',') >> real_p)
```

Notice `ch_p(',')`. It is a literal character parser that can recognize the comma `','`. In this case, the Kleene Star is modifying a more complex parser, namely, the one generated by the expression:

```
    (ch_p(',') >> real_p)
```

Note that this is a case where the parentheses are necessary. The Kleene star encloses the complete expression above.

Step 2. Using a Parser (now that it's created)

Now that we have created a parser, how do we use it? Like the result of any C++

temporary object, we can either store it in a variable, or call functions directly on it.

We'll gloss over some low-level C++ details and just get to the good stuff.

If `r` is a rule (don't worry about what rules exactly are for now. This will be discussed later. Suffice it to say that the rule is a placeholder variable that can hold a parser), then we store the parser as a rule like this:

```
    r = real_p >> *(ch_p(',') >> real_p);
```

Not too exciting, just an assignment like any other C++ expression you've used for years. The cool thing about storing a parser in a rule is this: rules are parsers, and now you can refer to it **by name**. (In this case the name is `r`). Notice that this is now a full assignment expression, thus we terminate it with a semicolon, "`;`".

That's it. We're done with defining the parser. So the next step is now invoking this parser to do its work. There are a couple of ways to do this. For now, we shall use the free `parse` function that takes in a `char const*`. The function accepts three arguments:

- The null-terminated `const char*` input
- The parser object
- Another parser called the **skip parser**

In our example, we wish to skip spaces and tabs. Another parser named `space_p` is included in Spirit's repertoire of predefined parsers. It is a very simple parser that simply recognizes whitespace. We shall use `space_p` as our skip parser. The skip parser is the one responsible for skipping characters in between parser elements such as the `real_p` and the `ch_p`.

Ok, so now let's parse!

```
    r = real_p >> *(ch_p(',') >> real_p);
    parse(str, r, space_p) // Not a full statement yet, patience...
```

The parse function returns an object (called `parse_info`) that holds, among other things, the result of the parse. In this example, we need to know:

- Did the parser successfully recognize the input `str`?
- Did the parser **fully** parse and consume the input up to its end?

To get a complete picture of what we have so far, let us also wrap this parser inside a function:

```
    bool
    parse_numbers(char const* str)
    {
        return parse(str, real_p >> *(',' >> real_p), space_p).full;
    }
```

Note in this case we dropped the named rule and inlined the parser directly in the call to parse. Upon calling parse, the expression evaluates into a temporary, unnamed parser which is passed into the parse() function, used, and then destroyed.

char and wchar_t operands

The careful reader may notice that the parser expression has *','* instead of `ch_p(',')` as the previous examples did. This is ok due to C++ syntax rules of conversion. There are `>>` operators that are overloaded to accept a `char` or `wchar_t` argument on its left or right (but not both). An operator may be overloaded if at least one of its parameters is a user-defined type. In this case, the `real_p` is the 2nd argument to `operator>>`, and so the proper overload of `>>` is used, converting *','* into a character literal parser.

The problem with omiting the `ch_p` call should be obvious: `'a' >> 'b'` is **not** a spirit parser, it is a numeric expression, right-shifting the ASCII (or another encoding) value of *'a'* by the ASCII value of *'b'*. However, both `ch_p('a') >> 'b'` and `'a' >> ch_p('b')` are Spirit sequence parsers for the letter *'a'* followed by *'b'*. You'll get used to it, sooner or later.

Take note that the object returned from the parse function has a member called `full` which returns true if both of our requirements above are met (i.e. the parser fully parsed the input).

Step 3. Semantic Actions

Our parser above is really nothing but a recognizer. It answers the question ***"did the input match our grammar?"***, but it does not remember any data, nor does it perform any side effects. Remember: we want to put the parsed numbers into a vector. This is done in an **action** that is linked to a particular parser. For example, whenever we parse a real number, we wish to store the parsed number after a successful match. We now wish to extract information from the parser. Semantic actions do this. Semantic actions may be attached to any point in the grammar specification. These actions are C++ functions or functors that are called whenever a part of the parser successfully recognizes a portion of the input. Say you have a parser **P**, and a C++ function **F**, you can make the parser call **F** whenever it matches an input by attaching **F**:

```
    P[&F]
```

Or if **F** is a function object (a functor):

```
    P[F]
```

The function/functor signature depends on the type of the parser to which it is attached. The parser `real_p` passes a single argument: the parsed number. Thus, if we were to attach a function **F** to `real_p`, we need **F** to be declared as:

```
    void F(double n);
```

For our example however, again, we can take advantage of some predefined semantic functors and functor generators (A functor generator is a function that returns a functor). For our purpose, Spirit has a functor generator `push_back_a(c)`. In brief, this semantic action, when called, **appends** the parsed value it receives from the parser it is attached to, to the container c.

Finally, here is our complete comma-separated list parser:

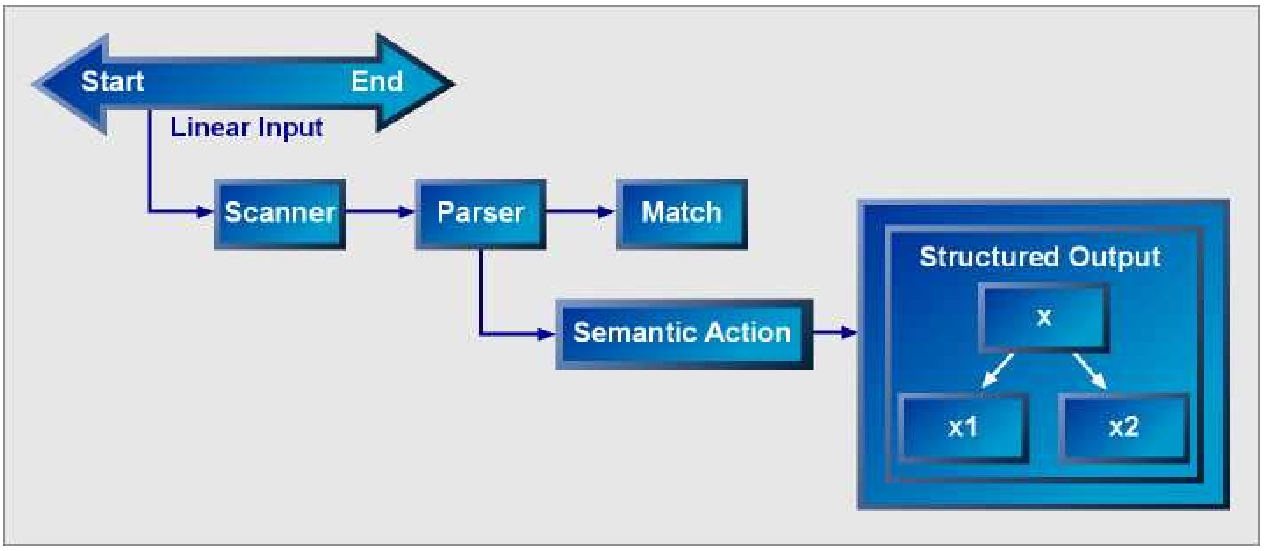
```
    bool
    parse_numbers(char const* str, vector<double>& v)
    {
        return parse(str,

            //  Begin grammar
            (
                real_p[push_back_a(v)] >> *(',' >> real_p[push_back_a(v)])
            )
            ,
            //  End grammar

            space_p).full;
    }
```

This is the same parser as above. This time with appropriate semantic actions attached to strategic places to extract the parsed numbers and stuff them in the vector v. The parse_numbers function returns true when successful.

The full source code can be **viewed here**. This is part of the Spirit distribution.

Basic Concepts

There are a few fundamental concepts that need to be understood well: 1) The **Parser**, 2) **Match**, 3) The **Scanner**, and 4) **Semantic Actions**. These basic concepts interact with one another, and the functionalities of each interweave throughout the framework to make it one coherent whole.



The Parser

Central to the framework is the parser. The parser does the actual work of recognizing a linear input stream of data read sequentially from start to end by the scanner. The parser attempts to match the input following a well-defined set of specifications known as grammar rules. The parser reports the success or failure to its client through a match object. When successful, the parser calls a client-supplied semantic action. Finally, the semantic action extracts structural information depending on the data passed by the parser and the hierarchical context of the parser it is attached to.

Parsers come in different flavors. The Spirit framework comes bundled with an extensive set of pre-defined parsers that perform various parsing tasks from the trivial to the complex. The parser, as a concept, has a public conceptual interface contract. Following the contract, anyone can write a conforming parser that will play along well with the framework's predefined components. We shall provide a blueprint detailing the conceptual interface of the parser later.

Clients of the framework generally do not need to write their own hand-coded parsers at all. Spirit has an immense repertoire of pre-defined parsers covering all aspects of syntax and semantic analysis. We shall examine this repertoire of parsers in the following sections. In the rare case where a specific functionality is not available, it is extremely easy to write a user-defined parser. The ease in writing a parser entity is the main reason for Spirit's extensibility.

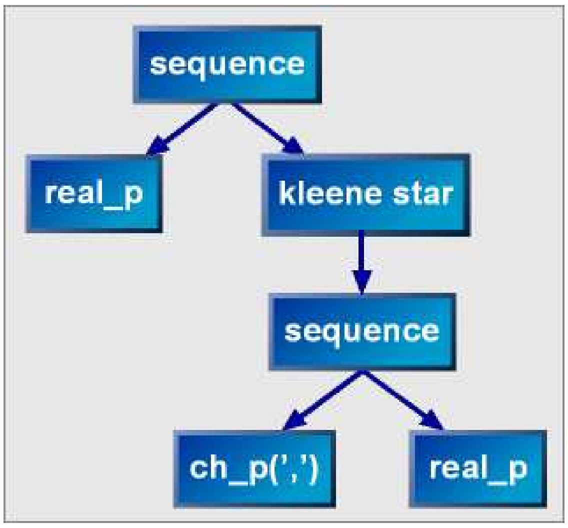
Primitives and Composites

Spirit parsers fall into two categories: **primitives** and **composites**. These two categories are more or less synonymous to terminals and non-terminals in parsing lingo. Primitives are non-decomposable atomic units. Composites on the other hand are parsers that are composed of other parsers which can in turn be a primitive or another composite. To illustrate, consider the Spirit expression:

```
    real_p >> *(',' >> real_p)
```

real_p is a primitive parser that can parse real numbers. The quoted comma ',' in the expression is a shortcut and is equivalent to ch_p(','), which is another primitive parser that recognizes single characters.

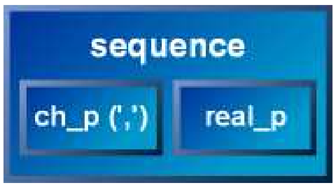
The expression above corresponds to the following parse tree:



The expression:

```
    ',' >> real_p
```

composes a **sequence** parser. The sequence parser is a composite parser comprising two parsers: the one on its left hand side (lhs), ch_p(',') ; and the other on its right hand side (rhs), real_p. This composite parser, when called, calls its lhs and rhs in sequence and reports a successful match only if both are successful.

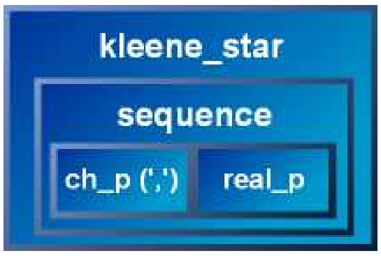


The sequence parser is a binary composite. It is composed of two parsers. There are unary composites as well. Unary composites hold only a single subject. Like the binary composite, the unary composite may change the behavior of its embedded subject. One particular example is the **Kleene star**. The Kleene star, when called to parse, calls its sole subject zero or more times. "Zero or more" implies that the Kleene star always returns a successful match, possibly matching the null string: "".

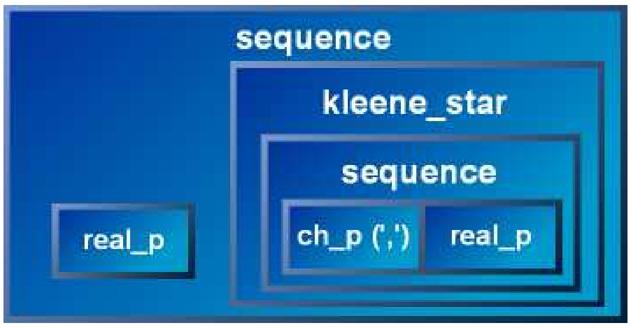
The expression:

```
    *(',' >> real_p)
```

wraps the whole sequence composite above inside a kleene_star.

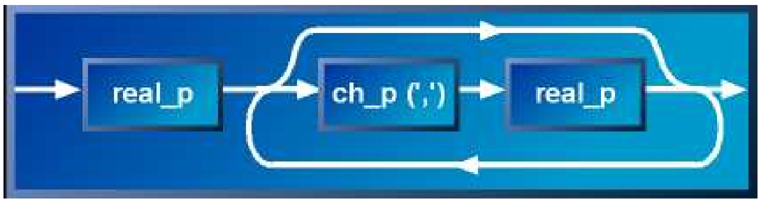


Finally, the full expression composes a real_p primitive parser and the kleene_star we have above into another higher level `sequence` parser composite.



A few simple classes, when composed and structured in a hierarchy, form a very powerful object-oriented recursive-descent parsing engine. These classes provide the infrastructure needed for the construction of more-complex parsers. The final parser composite is a non-deterministic recursive-descent parser with infinite look-ahead.

Top-down descent traverses the hierarchy. The outer `sequence` calls the leftmost `real_p` parser. If successful, the `kleene_star` is called next. The `kleene_star` calls the inner `sequence` repeatedly in a loop until it fails to match, or the input is exhausted. Inside, `ch_p(',')` and then `real_p` are called in sequence. The following diagram illustrates what is happening, somewhat reminiscent of Pascal syntax diagrams.



The flexibility of object embedding and composition combined with recursion opens up a unique approach to parsing. Subclasses are free to form aggregates and algorithms of arbitrary complexity. Complex parsers can be created with the composition of only a few primitive classes.

The framework is designed to be fully open-ended and extensible. New primitives or composites, from the trivial to the complex, may be added any time. Composition happens (statically) at compile time. This is possible through the expressive flexibility of C++ expression templates and template meta-programming.

The result is a composite composed of primitives and smaller composites. This embedding strategy gives us the ability to build hierarchical structures that fully model EBNF expressions of arbitrary complexity. Later on, we shall see more primitive and composite building blocks.

The Scanner

Like the parser, the scanner is also an abstract concept. The task of the scanner is to feed the sequential input data stream to the parser. The scanner is composed of two STL conforming forward iterators, first and last, where first is held by reference and last, by value. The first iterator is held by reference to allow re-positioning by the parser. A set of policies governs how the scanner behaves. Parsers extract data from the scanner and position the iterator appropriately through its member functions.

Knowledge of the intricacies of these policies is not required at all in most cases. However, knowledge of the scanner's basic API is required to write fully-conforming Spirit parsers. The scanner's API will be outlined in a separate section. In addition, for the power users and the adventurous among us, a full section will be devoted to covering the scanner policies. The scanner policies make Spirit very flexible and extensible. For instance, some of the policies may be modified to filter data. A practical example is a scanner policy that does not distinguish upper and lower case whereby making it useful for parsing case insensitive input. Another example is a scanner policy that strips white spaces from the input.

The Match

The parser has a conceptual parse member function taking in a scanner and returning a match object. The primary function of the match object is to report parsing success (or failure) back to the parser's caller; i.e., it evaluates to true if the parse function is successful, false otherwise. If the parse is successful, the match object may also be queried to report the number of characters matched (using `match.length()`). The length is non-negative if the match is successful, and the typical length of a parse failure is -1. A zero length is perfectly valid and still represents a successful match.

Parsers may have attribute data associated with it. For example, the real_p parser has a numeric datum associated with it. This attribute is the parsed number. This attribute is passed on to the returned match object. The match object may be queried to get this attribute. This datum is valid only when the match is successful.

Semantic Actions

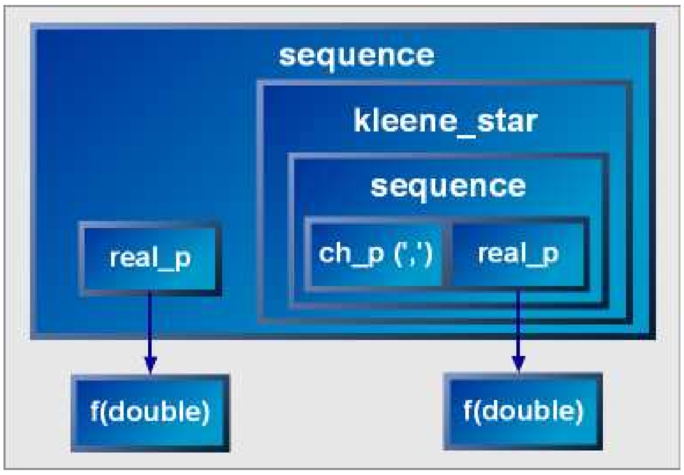
A composite parser forms a hierarchy. Parsing proceeds from the topmost parent parser which delegates and apportions the parsing task to its children recursively to its children's children and so on until a primitive is reached. By attaching semantic actions to various points in this hierarchy, in effect we can transform the flat linear input stream into a structured representation. This is essentially what parsers do.

Recall our example above:

```
    real_p >> *(',' >> real_p)
```

By hooking a function (or functor) into the real_p parsers, we can extract the numbers from the input:

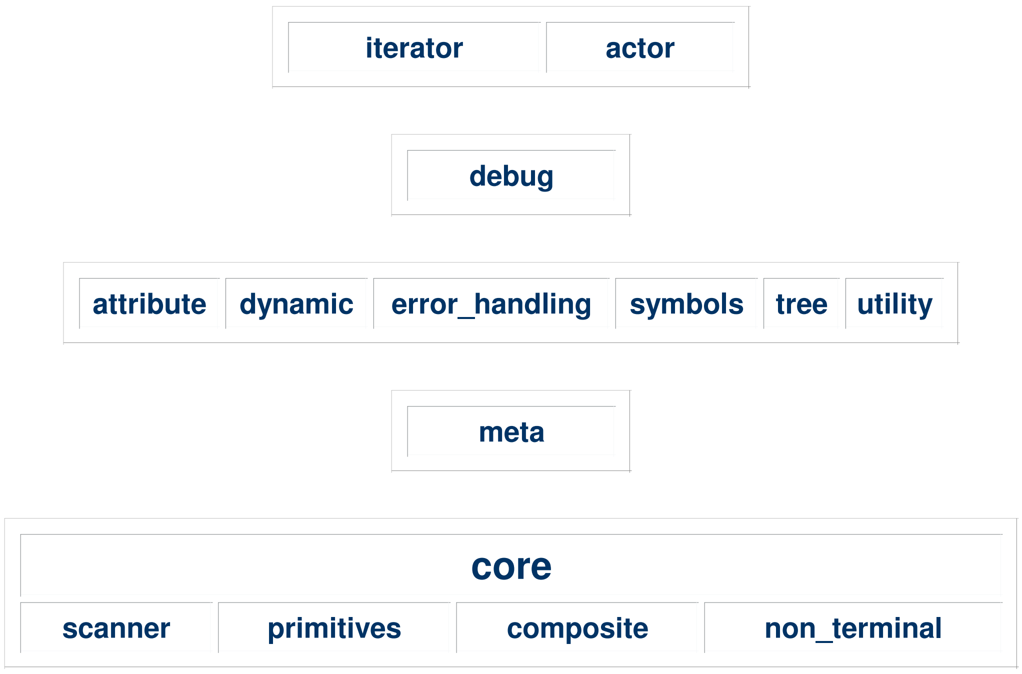
```
    real_p[&f] >> *(',' >> real_p[&f])
```



where `f` is a function that takes in a single argument. The `[&f]` hooks the parser with the function such that when `real_p` recognizes a valid number, the function `f` is called. It is up to the function then to do what is appropriate. For example, it can stuff the numbers in a vector. Or perhaps, if the grammar is changed slightly by replacing `','` with `'+'`, then we have a primitive calculator that computes sums. The function `f` then can then be made to add all incoming numbers.

Organization

The framework is highly modular and is organized in layers:



Spirit has four layers, plus an independent top layer. The independent layer, comprising of actor and iterator, does not rely on the other layers. The framework's architecture is completely orthogonal. The relationship among the layers is acyclic. Lower layers do not depend nor know the existence of upper layers. Modules in a layer do not depend on other modules in the same layer.

The client may use only the modules that she wants without incurring any compile time nor run time penalty. A minimalistic approach is to use only the core as is. The highly streamlined core is usable by itself. The core is sufficiently suitable for tasks such as micro parsing.

The **iterator** module is independent of Spirit and may be used in other non-Spirit applications. This module is a compilation of stand-alone iterators and iterator wrappers compatible with Spirit. Over time, these iterators have been found to be most useful for parsing with Spirit.

The **actor** module, also independent of Spirit, is a compilation of predefined semantic actions that covers the most common semantics processing tasks.

The **debug** module provides library wide parser debugging. This module hooks itself up transparently into the core non-intrusively and only when necessary.

The **attribute** module introduces advanced semantic action machinery with emphasis on extraction and passing of data up and down the parser hierarchy through inherited and synthesized attributes. Attributes may also be used to actually control the parsing. Parametric parsers are a form of dynamic parsers that changes their behavior at run time based on some attribute or data.

The **dynamic** module focuses on parsers with behavior that can be modified at run-time.

error_handling. The framework would not be complete without Error Handling. C++'s exception handling mechanism is a perfect match for Spirit due to its highly recursive functional nature. C++ Exceptions are used extensively by this module for handling errors.

The **symbols** module focuses on symbol table management. This module is rather basic now. The goal is to build a sub-framework that will be able to accommodate C++ style multiple scope mechanisms. C++ is a great model for the complexity of scoping that perhaps has no parallel in any other language. There are classes and inheritance, private, protected and public access restrictions, friends, namespaces, using declarations, using directives, Koenig lookup (Argument Dependent Lookup) and more. The symbol table functionality we have now will be the basis of a complete facility that will attempt to model this.

I wish that I could ever see, a structure as lovely as a tree...

Parse Tree and Abstract Syntax Tree (AST) generation are handled by the **Tree** module. There are advantages with Parse Trees and Abstract Syntax Trees over semantic actions. You can make multiple passes over the data without having to re-parse the input. You can perform transformations on the tree. You can evaluate things in any order you want, whereas with attribute schemes you have to process in a begin to end fashion. You do not have to worry about backtracking and action side effects that may occur with an ambiguous grammar.

The **utility** module is a set of commonly useful parsers and support classes that were found to be useful in handling common tasks such as list processing, comments, confix expressions, etc.

meta, provides metaprogramming facilities for advanced Spirit developers. This module facilitates compile-time and run-time introspection of Spirit parsers.

Primitives

The framework predefines some parser primitives. These are the most basic building blocks that the client uses to build more complex parsers. These primitive parsers are template classes, making them very flexible.

These primitive parsers can be instantiated directly or through a templatized helper function. Generally, the helper function is far simpler to deal with as it involves less typing.

We have seen the character literal parser before through the generator function `ch_p` which is not really a parser but, rather, a parser generator. Class `chlit<CharT>` is the actual template class behind the character literal parser. To instantiate a `chlit` object, you must explicitly provide the character type, `CharT`, as a template parameter which determines the type of the character. This type typically corresponds to the input type, usually `char` or `wchar_t`. The following expression creates a temporary parser object which will recognize the single letter ***'X'***.

```
    chlit<char>('X');
```

Using `chlit`'s generator function `ch_p` simplifies the usage of the `chlit<>` class (this is true of most Spirit parser classes since most have corresponding generator functions). It is convenient to call the function because the compiler will deduce the template type through argument deduction for us. The example above could be expressed less verbosely using the `ch_p` helper function.

```
    ch_p('X')  // equivalent to chlit<char>('X') object
```

Parser generators

Whenever you see an invocation of the parser generator function, it is equivalent to the parser itself. Therefore, we often call `ch_p` a character parser, even if, technically speaking, it is a function that generates a character parser.

The following grammar snippet shows these forms in action:

```
    // a rule can "store" a parser object.  They're covered
    // later, but for now just consider a rule as an opaque type
    rule<> r1, r2, r3;

    chlit<char> x('X');     // declare a parser named x

    r1 = chlit<char>('X');  //  explicit declaration
    r2 = x;                 //  using x
    r3 = ch_p('X')          //  using the generator
```

chlit and ch_p

Matches a single character literal. `chlit` has a single template type parameter which defaults to `char` (i.e. `chlit<>` is equivalent to `chlit<char>`). This type parameter is the character type that `chlit` will recognize when parsing. The function generator version deduces the template type parameters from the actual function arguments. The `chlit` class constructor accepts a single parameter: the character it will match the input against. Examples:

```
    r1 = chlit<>('X');
    r2 = chlit<wchar_t>(L'X');
    r3 = ch_p('X');
```

Going back to our original example:

```
    group = '(' >> expr >> ')';
    expr1 = integer | group;
    expr2 = expr1 >> *(('*' >> expr1) | ('/' >> expr1));
    expr  = expr2 >> *(('+' >> expr2) | ('-' >> expr2));
```

the character literals **`'('`**, **`')'`**, **`'+'`**, **`'-'`**, **`'*'`** and **`'/'`** in the grammar declaration are `chlit` objects that are implicitly created behind the scenes.

char operands

The reason this works is from two special templatized overloads of `operator>>` that takes a (`char`, `ParserT`), or (`ParserT`, `char`). These functions convert the character into a `chlit` object.

One may prefer to declare these explicitly as:

```
    chlit<> plus('+');
    chlit<> minus('-');
    chlit<> times('*');
    chlit<> divide('/');
    chlit<> oppar('(');
    chlit<> clpar(')');
```

range and range_p

A `range` of characters is created from a low/high character pair. Such a parser matches a single character that is in the `range`, including both endpoints. Like `chlit`, `range` has a single template type parameter which defaults to `char`. The `range` class constructor accepts two parameters: the character range (*from* and *to*, inclusive) it will match the input against. The function generator version is `range_p`. Examples:

```
    range<>('A','Z')    // matches 'A'..'Z'
    range_p('a','z')    // matches 'a'..'z'
```

Note, the first character must be "before" the second, according to the underlying character encoding characters. The range, like chlit is a single character parser.

Character mapping

Character mapping to is inherently platform dependent. It is not guaranteed in the standard for example that 'A' < 'Z', however, in many occasions, we are well aware of the character set we are using such as ASCII, ISO-8859-1 or Unicode. Take care though when porting to another platform.

strlit and str_p

This parser matches a string literal. `strlit` has a single template type parameter: an iterator type. Internally, `strlit` holds a begin/end iterator pair pointing to a string or a container of characters. The `strlit` attempts to match the current input stream with this string. The template type parameter defaults to `char const*`. `strlit` has two constructors. The first accepts a null-terminated character pointer. This constructor may be used to build `strlits` from quoted string literals. The second constructor takes in a first/last iterator pair. The function generator version is `str_p`. Examples:

```
    strlit<>("Hello World")
    str_p("Hello World")

    std::string msg("Hello World");
    strlit<std::string::const_iterator>(msg.begin(), msg.end());
```

Character and phrase level parsing

Typical parsers regard the processing of characters (symbols that form words or lexemes) and phrases (words that form sentences) as separate domains. Entities such as reserved words, operators, literal strings, numerical constants, etc., which constitute the terminals of a grammar are usually extracted first in a separate lexical analysis stage.

At this point, as evident in the examples we have so far, it is important to note that, contrary to standard practice, the Spirit framework handles parsing tasks at both the character level as well as the phrase level. One may consider that a lexical analyzer is seamlessly integrated in the Spirit framework.

Although the Spirit parser library does not need a separate lexical analyzer, there is no reason why we cannot have one. One can always have as many parser layers as needed. In theory, one may create a preprocessor, a lexical analyzer and a parser proper, all using the same framework.

chseq and chseq_p

Matches a character sequence. `chseq` has the same template type parameters and constructor parameters as strlit. The function generator version is chseq_p. Examples:

```
    chseq<>("ABCDEFG")
    chseq_p("ABCDEFG")
```

strlit is an implicit lexeme. That is, it works solely on the character level. chseq, strlit's twin, on the other hand, can work on both the character and phrase levels. What this simply means is that it can ignore white spaces in between the string characters. For example:

```
    chseq<>("ABCDEFG")
```

can parse:

```
    ABCDEFG
    A B C D E F G
    AB CD EFG
```

More character parsers

The framework also predefines the full repertoire of single character parsers:

Single character parsers

anychar_p	Matches any single character (including the null terminator: '\0')
alnum_p	Matches alpha-numeric characters
alpha_p	Matches alphabetic characters
blank_p	Matches spaces or tabs
cntrl_p	Matches control characters
digit_p	Matches numeric digits
graph_p	Matches non-space printing characters
lower_p	Matches lower case letters
print_p	Matches printable characters
punct_p	Matches punctuation symbols
space_p	Matches spaces, tabs, returns, and newlines
upper_p	Matches upper case letters
xdigit_p	Matches hexadecimal digits

negation ~

Single character parsers such as the chlit, range, anychar_p, alnum_p etc. can be negated. For example:

```
    ~ch_p('x')
```

matches any character except `'x'`. Double negation of a character parser cancels out the negation. `~~alpha_p` is equivalent to `alpha_p`.

eol_p

Matches the end of line (CR/LF and combinations thereof).

nothing_p

Never matches anything and always fails.

end_p

Matches the end of input (returns a sucessful match with 0 length when the input is exhausted)

Operators

Operators are used as a means for object composition and embedding. Simple parsers may be composed to form composites through operator overloading, crafted to approximate the syntax of an Extended Backus-Normal Form (EBNF) variant. An expression such as:

```
    a | b
```

actually yields a new parser type which is a composite of its operands, a and b. Taking this example further, if a and b were of type `chlit<>`, the result would have the composite type:

```
    alternative<chlit<>, chlit<> >
```

In general, for any binary operator, it will take its two arguments, parser1 and parser2, and create a new composed parser of the form

```
    op<parser1, parser2>
```

where parser1 and parser2 can be arbitrarily complex parsers themselves, with the only limitations being what your compiler imposes.

Set Operators

Set operators

`a	b`	Union	Match a or b. Also referred to as alternative
`a & b`	Intersection	Match a and b
`a - b`	Difference	Match a but not b. If both match and b's matched text is shorter than a's matched text, a successful match is made
`a ^ b`	XOR	Match a or b, but not both

Short-circuiting

Alternative operands are tried one by one on a first come first served basis starting from the leftmost operand. After a successfully matched alternative is found, the parser concludes its search, essentially short-circuiting the search for other potentially viable candidates. This short-circuiting implicitly gives the highest priority to the leftmost alternative.

Short-circuiting is done in the same manner as C or C++'s logical expressions; e.g. `if (x < 3 || y < 2)` where, if `x` evaluates to be less than 3, the `y < 2` test is not done at all. In addition to providing an implicit priority rule for alternatives which is necessary, given the non-deterministic nature of the Spirit parser compiler, short-circuiting improves the execution time. If the order of your alternatives is logically irrelevant, strive to put the (expected) most common choice first for maximum efficiency.

Intersections

Some researchers assert that the intersections (e.g. `a & b`) let us define context sensitive languages (**"XBNF"** [citing Leu-Weiner, 1973]). "The theory of defining a language as the intersection of a finite number of context free languages was developed by Leu and Weiner in 1973".

~ Operator

The complement operator `~` was originally put into consideration. Further understanding of its value and meaning leads us to uncertainty. The basic problem stems from the fact that `~a` will yield `U-a`, where `U` is the universal set of all strings. However, where it makes sense, some parsers can be complemented (see the **primitive character parsers** for examples).

Sequencing Operators

Sequencing operators

a >> b	Sequence	Match a and b in sequence
a && b	Sequential-and	Sequential-and. Same as above, match a and b in sequence
a || b	Sequential-or	Match a or b in sequence

The sequencing operator `>>` can alternatively be thought of as the sequential-and operator. The expression `a && b` reads as match a and b in sequence. Continuing this logic, we can also have a sequential-or operator where the expression `a || b` reads as match a or b and in sequence. That is, if both a and b match, it must be in sequence; this is equivalent to `a >> !b | b`.

Optional and Loops

Optional and Loops

*a	Kleene star	Match a zero (0) or more times
+a	Positive	Match a one (1) or more times
!a	Optional	Match a zero (0) or one (1) time
a % b	List	Match a list of one or more repetitions of a separated by occurrences of b. This is the same as `a >> *(b >> a)`. Note that `a` must not also match `b`

If we look more closely, take note that we generalized the optional expression of the form `!a` in the same category as loops. This is logical, considering that the optional matches the expression following it zero (0) or one (1) time.

Primitive type operands

For binary operators, one of the operands but not both may be a `char`, `wchar_t`, `char const*` or `wchar_t const*`. Where P is a parser object, here are some examples:

```
    P | 'x'
    P - L"Hello World"
    'x' >> P
    "bebop" >> P
```

It is important to emphasize that C++ mandates that operators may only be overloaded if at least one argument is a user-defined type. Typically, in an expression involving multiple operators, explicitly typing the leftmost operand as a parser is enough to cause propagation to all the rest of the operands to its right to be regarded as parsers. Examples:

```
    r = 'a' | 'b' | 'c' | 'd';          // ill formed
    r = ch_p('a')  | 'b' | 'c' | 'd';   // OK
```

The second case is parsed as follows:

```
    r → (((chlit<char> | char) | char) | char)

    a → (chlit<char> | char)
    r → (((a) | char) | char)

    b → (a | char)
    r → (((b)) | char)

    c → (b | char)
    r → (((c)))
```

Operator precedence and grouping

Since we are defining our meta-language in C++, we follow C/C++'s operator precedence rules. Grouping expressions inside the parentheses override this (e.g., `*(a | b)` reads: match a or b zero (0) or more times).

Numerics

Similar to chlit, strlit etc. numeric parsers are also primitives. Numeric parsers are placed on a section of their own to give this important building block better focus. The framework includes a couple of predefined objects for parsing signed and unsigned integers and real numbers. These parsers are fully parametric. Most of the important aspects of numeric parsing can be finely adjusted to suit. This includes the radix base, the minimum and maximum number of allowable digits, the exponent, the fraction etc. Policies control the real number parsers' behavior. There are some predefined policies covering the most common real number formats but the user can supply her own when needed.

uint_parser

This class is the simplest among the members of the numerics package. The uint_parser can parse unsigned integers of arbitrary length and size. The uint_parser parser can be used to parse ordinary primitive C/C++ integers or even user defined scalars such as bigints (unlimited precision integers). Like most of the classes in Spirit, the uint_parser is a template class. Template parameters fine tune its behavior. The uint_parser is so flexible that the other numeric parsers are implemented using it as the backbone.

```
    template <
        typename T = unsigned,
        int Radix = 10,
        unsigned MinDigits = 1,
        int MaxDigits = -1>
    struct uint_parser { /*...*/ };
```

uint_parser template parameters

T	The numeric base type of the numeric parser. Defaults to unsigned
Radix	The radix base. This can be either 2: binary, 8: octal, 10: decimal and 16: hexadecimal. Defaults to 10; decimal
MinDigits	The minimum number of digits allowable
MaxDigits	The maximum number of digits allowable. If this is -1, then the maximum limit becomes unbounded

Predefined uint_parsers

bin_p	`uint_parser<unsigned, 2, 1, -1> const`
oct_p	`uint_parser<unsigned, 8, 1, -1> const`
uint_p	`uint_parser<unsigned, 10, 1, -1> const`
hex_p	`uint_parser<unsigned, 16, 1, -1> const`

The following example shows how the uint_parser can be used to parse thousand separated numbers. The example can correctly parse numbers such as 1,234,567,890.

```
    uint_parser<unsigned, 10, 1, 3> uint3_p;        //  1..3 digits
    uint_parser<unsigned, 10, 3, 3> uint3_3_p;      //  exactly 3 digits
    ts_num_p = (uint3_p >> *(',' >> uint3_3_p));    //  our thousand separated numb
```

bin_p, oct_p, uint_p and hex_p are parser generator objects designed to be used within expressions. Here's an example of a rule that parses comma delimited list of numbers (We've seen this **before**):

```
    list_of_numbers = real_p >> *(',' >> real_p);
```

Later, we shall see how we can extract the actual numbers parsed by the numeric parsers. We shall deal with this when we get to the section on **specialized actions**.

int_parser

The `int_parser` can parse signed integers of arbitrary length and size. This is almost the same as the `uint_parser`. The only difference is the additional task of parsing the `'+'` or `'-'` sign preceding the number. The class interface is the same as that of the uint_parser.

A predefined int_parser

int_p	`int_parser<int, 10, 1, -1> const`

real_parser

The `real_parser` can parse real numbers of arbitrary length and size limited by its parametric type `T`. The `real_parser` is a template class with 2 template parameters. Here's the `real_parser` template interface:

```
    template<
        typename T = double,
        typename RealPoliciesT = ureal_parser_policies<T> >
    struct real_parser;
```

The first template parameter is its numeric base type `T`. This defaults to `double`.

Parsing special numeric types

Notice that the numeric base type `T` can be specified by the user. This means that we can use the numeric parsers to parse user defined numeric types such as `fixed_point` (fixed point reals) and `bigint` (unlimited precision integers).

The second template parameter is a class that groups all the policies and defaults to `ureal_parser_policies<T>`. Policies control the real number parsers' behavior. The default policies provided are designed to parse C/C++ style real numbers of the form **nnn.fff.Eeee** where **nnn** is the whole number part, **fff** is the fractional part, **E** is `'e'` or `'E'` and **eee** is the exponent optionally preceded by `'-'` or `'+'`. This corresponds to the following grammar, with the exception that plain integers without the decimal point are also accepted by default.

```
    floatingliteral
        =   fractionalconstant >> !exponentpart
        |  +digit_p >> exponentpart
        ;

    fractionalconstant
        =  *digit_p >> '.' >> +digit_p
        |  +digit_p >> '.'
        ;

    exponentpart
        =   ('e' | 'E') >> !('+' | '-') >> +digit_p
        ;
```

The default policies are provided to take care of the most common case (there are many ways to represent, and hence parse, real numbers). In most cases, the default setting of the `real_parser` is sufficient and can be used straight out of the box. Actually, there are four `real_parser`s pre-defined for immediate use:

Predefined real_parsers

ureal_p	`real_parser<double, ureal_parser_policies<double> > const`
real_p	`real_parser<double, real_parser_policies<double> > const`
strict_ureal_p	`real_parser<double, strict_ureal_parser_policies<double> > const`
strict_real_p	`real_parser<double, strict_real_parser_policies<double> > const`

We've seen `real_p` before. `ureal_p` is its unsigned variant.

Strict Reals

Integer numbers are considered a subset of real numbers, so `real_p` and `ureal_p` recognize integer numbers (without a dot) as real numbers. `strict_real_p` and `strict_ureal_p` are the equivalent parsers that **require** a dot to be present for a number to be considered a successful match.

Advanced: real_parser policies

The parser policies break down real number parsing into 6 steps:

1	**parse_sign**	Parse the prefix sign
2	**parse_n**	Parse the integer at the left of the decimal point
3	**parse_dot**	Parse the decimal point
4	**parse_frac_n**	Parse the fraction after the decimal point
5	**parse_exp**	Parse the exponent prefix (e.g. 'e')
6	**parse_exp_n**	Parse the actual exponent

And the interaction of these sub-parsing tasks is further controlled by these 3 policies:

1	**allow_leading_dot**	Allow a leading dot to be present (".1" becomes equivalent to "0.1")
2	**allow_trailing_dot**	Allow a trailing dot to be present ("1." becomes equivalent to "1.0")
3	**expect_dot**	Require a dot to be present (disallows "1" to be equivalent to "1.0")

[From here on, required reading: **The Scanner**, **In-depth The Parser** and **In-depth The Scanner**]

sign_parser and sign_p

Before we move on, a small utility parser is included here to ease the parsing of the '-' or '+' sign. While it is easy to write one:

```
    sign_p = (ch_p('+') | '-');
```

it is not possible to extract the actual sign (positive or negative) without resorting to semantic actions. The sign_p parser has a bool attribute returned to the caller through the match object which, after parsing, is set to **true** if the parsed sign is negative. This attribute detects if the negative sign has been parsed. Examples:

```
    bool is_negative;
    r = sign_p[assign_a(is_negative)];
```

or simply...

```
    // directly extract the result from the match result's value
    bool is_negative = sign_p.parse(scan).value();
```

The sign_p parser expects attached semantic actions to have a signature (see **Specialized Actions** for further detail) compatible with:

Signature for functions:

```
    void func(bool is_negative);
```

Signature for functors:

```
    struct ftor
    {
        void operator()(bool is_negative) const;
    };
```

ureal_parser_policies

```
    template <typename T>
    struct ureal_parser_policies
    {
        typedef uint_parser<T, 10, 1, -1>   uint_parser_t;
        typedef int_parser<T, 10, 1, -1>    int_parser_t;

        static const bool allow_leading_dot  = true;
        static const bool allow_trailing_dot = true;
        static const bool expect_dot         = false;

        template <typename ScannerT>
        static typename match_result<ScannerT, nil_t>::type
        parse_sign(ScannerT& scan)
        { return scan.no_match(); }

        template <typename ScannerT>
        static typename parser_result<uint_parser_t, ScannerT>::type
        parse_n(ScannerT& scan)
        { return uint_parser_t().parse(scan); }

        template <typename ScannerT>
        static typename parser_result<chlit<>, ScannerT>::type
        parse_dot(ScannerT& scan)
        { return ch_p('.').parse(scan); }

        template <typename ScannerT>
        static typename parser_result<uint_parser_t, ScannerT>::type
        parse_frac_n(ScannerT& scan)
        { return uint_parser_t().parse(scan); }

        template <typename ScannerT>
        static typename parser_result<chlit<>, ScannerT>::type
        parse_exp(ScannerT& scan)
        { return as_lower_d['e'].parse(scan); }

        template <typename ScannerT>
        static typename parser_result<int_parser_t, ScannerT>::type
        parse_exp_n(ScannerT& scan)
        { return int_parser_t().parse(scan); }
    };
```

The default ureal_parser_policies uses the lower level integer numeric parsers to do its job.

real_parser_policies

```
    template <typename T>
    struct real_parser_policies : public ureal_parser_policies<T>
    {
        template <typename ScannerT>
        static typename parser_result<sign_parser, ScannerT>::type
        parse_sign(ScannerT& scan)
        { return sign_p.parse(scan); }
    };
```

Notice how the real_parser_policies replaced **`parse_sign`** of the **u**real_parser_policies from which it is subclassed. The default real_parser_policies simply uses a `sign_p` instead of `scan.no_match()` in the `parse_sign` step.

strict_ureal_parser_policies and strict_real_parser_policies

```
    template <typename T>
    struct strict_ureal_parser_policies : public ureal_parser_policies<T>
    {
        static const bool expect_dot = true;
    };

    template <typename T>
    struct strict_real_parser_policies : public real_parser_policies<T>
    {
        static const bool expect_dot = true;
    };
```

Again, these policies replaced just the policies they wanted different from their superclasses.

Specialized real parser policies can reuse some of the defaults while replacing a few. For example, the following is a real number parser policy that parses thousands separated numbers with at most two decimal places and no exponent.

The full source code can be viewed **here**.

```
    template <typename T>
    struct ts_real_parser_policies : public ureal_parser_policies<T>
    {
        //  These policies can be used to parse thousand separated
        //  numbers with at most 2 decimal digits after the decimal
        //  point. e.g. 123,456,789.01

        typedef uint_parser<int, 10, 1, 2>  uint2_t;
        typedef uint_parser<T, 10, 1, -1>   uint_parser_t;
        typedef int_parser<int, 10, 1, -1>  int_parser_t;

        //////////////////////////////////  2 decimal places Max
        template <typename ScannerT>
        static typename parser_result<uint2_t, ScannerT>::type
        parse_frac_n(ScannerT& scan)
        { return uint2_t().parse(scan); }

        //////////////////////////////////  No exponent
        template <typename ScannerT>
        static typename parser_result<chlit<>, ScannerT>::type
        parse_exp(ScannerT& scan)
        { return scan.no_match(); }

        //////////////////////////////////  No exponent
        template <typename ScannerT>
        static typename parser_result<int_parser_t, ScannerT>::type
        parse_exp_n(ScannerT& scan)
        { return scan.no_match(); }

        //////////////////////////////////  Thousands separated numbers
        template <typename ScannerT>
        static typename parser_result<uint_parser_t, ScannerT>::type
        parse_n(ScannerT& scan)
        {
            typedef typename parser_result<uint_parser_t, ScannerT>::type RT;
            static uint_parser<unsigned, 10, 1, 3> uint3_p;
            static uint_parser<unsigned, 10, 3, 3> uint3_3_p;

            if (RT hit = uint3_p.parse(scan))
            {
                T n;
                typedef typename ScannerT::iterator_t iterator_t;
                iterator_t save = scan.first;
                while (match<> next = (',' >> uint3_3_p[assign_a(n)]).parse(scan))
                {
                    hit.value() *= 1000;
                    hit.value() += n;
                    scan.concat_match(hit, next);
                    save = scan.first;
                }
                scan.first = save;
                return hit;

                // Note: On erroneous input such as "123,45", the result should
                // be a partial match "123". 'save' is used to makes sure that
                // the scanner position is placed at the last *valid* parse
                // position.
            }
            return scan.no_match();
        }
    };
```

The Rule

The **rule** is a polymorphic parser that acts as a named place-holder capturing the behavior of an EBNF expression assigned to it. Naming an EBNF expression allows it to be referenced later. The `rule` is a template class parameterized by the type of the scanner (`ScannerT`), the rule's **context** and its **tag**. Default template parameters are provided to make it easy to use the rule.

```
    template<
        typename ScannerT = scanner<>,
        typename ContextT = parser_context<>,
        typename TagT = parser_address_tag>
    class rule;
```

Default template parameters are supplied to handle the most common case. `ScannerT` defaults to `scanner<>`, a plain vanilla scanner that acts on `char const*` iterators and does nothing special at all other than iterate through all the chars in the null terminated input a character at a time. The rule tag, `TagT`, typically used with **ASTs**, is used to identify a rule; it is explained **here**. In trivial cases, declaring a rule as `rule<>` is enough. You need not be concerned at all with the `ContextT` template parameter unless you wish to tweak the low level behavior of the rule. Detailed information on the `ContextT` template parameter is provided **elsewhere**.

Order of parameters

As of v1.8.0, the `ScannerT`, `ContextT` and `TagT` can be specified in any order. If a template parameter is missing, it will assume the defaults. Examples:

```
    rule<> rx1;
    rule<scanner<> > rx2;
    rule<parser_context<> > rx3;
    rule<parser_context<>, parser_address_tag> rx4;
    rule<parser_address_tag> rx5;
    rule<parser_address_tag, scanner<>, parser_context<> > rx6;
    rule<parser_context<>, scanner<>, parser_address_tag> rx7;
```

Multiple scanners

As of v1.8.0, rules can use one or more scanner types. There are cases, for instance, where we need a rule that can work on the phrase and character levels. Rule/scanner mismatch has been a source of confusion and is the no. 1 **FAQ**. To address this issue, we now have multiple scanner support. Example:

```
    typedef scanner_list<scanner<>, phrase_scanner_t> scanners;

    rule<scanners>  r = +anychar_p;
    assert(parse("abcdefghijk", r).full);
    assert(parse("a b c d e f g h i j k", r, space_p).full);
```

Notice how rule `r` is used in both the phrase and character levels.

By default support for multiple scanners is disabled. The macro BOOST_SPIRIT_RULE_SCANNERTYPE_LIMIT must be defined to the maximum number of scanners allowed in a scanner_list. The value must be greater than 1 to enable multiple scanners. Given the example above, to define a limit of two scanners for the list, the following line must be inserted into the source file before the inclusion of Spirit headers:

```
    #define BOOST_SPIRIT_RULE_SCANNERTYPE_LIMIT 2
```

See the techniques section for an **example** of a **grammar** using a multiple scanner enabled rule, **lexeme_scanner** and **as_lower_scanner.**

Rule Declarations

The rule class models EBNF's production rule. Example:

```
    rule<> a_rule = *(a | b) & +(c | d | e);
```

The type and behavior of the right-hand (rhs) EBNF expression, which may be arbitrarily complex, is encoded in the rule named a_rule. a_rule may now be referenced elsewhere in the grammar:

```
    rule<> another_rule = f >> g >> h >> a_rule;
```

Referencing rules

When a rule is referenced anywhere in the right hand side of an EBNF expression, the rule is held by the expression by reference. It is the responsibility of the client to ensure that the referenced rule stays in scope and does not get destructed while it is being referenced.

```
    a = int_p;
    b = a;
    c = int_p >> b;
```

Copying Rules

The rule is a weird C++ citizen, unlike any other C++ object. It does not have the proper copy and assignment semantics and cannot be stored and passed around by value. If you need to copy a rule you have to explicitly call its member function copy():

```
    r.copy();
```

However, be warned that copying a rule will not deep copy other referenced rules of the source rule being copied. This might lead to dangling references. Again, it is the responsibility of the client to ensure that all referenced rules stay in scope and does not get destructed while it is being referenced. Caveat emptor.

If you copy a rule, then you'll want to place it in a storage somewhere. The problem is how? The storage can't be another rule:

```
    rule<> r2 = r.copy(); // BAD!
```

because rules are weird and does not have the expected C++ copy-constructor and assignment semantics! As a general rule: **Don't put a copied rule into another rule!** Instead, use the **stored_rule** for that purpose.

Forward declarations

A `rule` may be declared before being defined to allow cyclic structures typically found in BNF declarations. Example:

```
    rule<> a, b, c;

    a = b | a;
    b = c | a;
```

Recursion

The right-hand side of a rule may reference other rules, including itself. The limitation is that direct or indirect left recursion is not allowed (this is an unchecked run-time error that results in an infinite loop). This is typical of top-down parsers. Example:

```
    a = a | b; // infinite loop!
```

What is left recursion?

Left recursion happens when you have a rule that calls itself before anything else. A top-down parser will go into an infinite loop when this happens. See the **FAQ** for details on how to eliminate left recursion.

Undefined rules

An undefined rule matches nothing and is semantically equivalent to `nothing_p`.

Redeclarations

Like any other C++ assignment, a second assignment to a rule is destructive and will redefine it. The old definition is lost. Rules are dynamic. A rule can change its definition anytime:

```
    r = a_definition;
    r = another_definition;
```

Rule `r` loses the old definition when the second assignment is made. As mentioned, an undefined rule matches nothing and is semantically equivalent to `nothing_p`.

Dynamic Parsers

Hosting declarative EBNF in imperative C++ yields an interesting blend. We have the best of both worlds. We have the ability to conveniently modify the grammar at run time using imperative constructs such as if, else statements. Example:

```
    if (feature_is_available)
        r = add_this_feature;
```

Rules are essentially dynamic parsers. A dynamic parser is characterized by its ability to modify its behavior at run time. Initially, an undefined rule matches nothing. At any time, the rule may be defined and redefined, thus, dynamically altering its behavior.

No start rule

Typically, parsers have what is called a start symbol, chosen to be the root of the grammar where parsing starts. The Spirit parser framework has no notion of a start symbol. Any rule can be a start symbol. This feature promotes step-wise creation of parsers. We can build parsers from the bottom up while fully testing each level or module up untill we get to the top-most level.

Parser Tags

Rules may be tagged for identification purposes. This is necessary, especially when dealing with **parse trees and ASTs** to see which rule created a specific AST/parse tree node. Each rule has an ID of type parser_id. This ID can be obtained through the rule's id() member function:

```
    my_rule.id(); //  get my_rule's id
```

The parser_id class is declared as:

```
    class parser_id
    {
    public:
                        parser_id();
        explicit        parser_id(void const* p);
                        parser_id(std::size_t l);

        bool            operator==(parser_id const& x) const;
        bool            operator!=(parser_id const& x) const;
        bool            operator<(parser_id const& x) const;
        std::size_t     to_long() const;
    };
```

parser_address_tag

The rule's TagT template parameter supplies this ID. This defaults to parser_address_tag. The parser_address_tag uses the address of the rule as its ID. This is often not the most convenient, since it is not always possible to get the address of a rule to compare against.

parser_tag

It is possible to have specific constant integers to identify a rule. For this purpose, we can use the `parser_tag<N>`, where N is a constant integer:

```
    rule<parser_tag<123> > my_rule; //  set my_rule's id to 123
```

dynamic_parser_tag

The `parser_tag<N>` can only specifiy a **static ID**, which is defined at compile time. If you need the ID to be **dynamic** (changeable at runtime), you can use the `dynamic_parser_tag` class as the `TagT` template parameter. This template parameter enables the `set_id()` function, which may be used to set the required id at runtime:

```
    rule<dynamic_parser_tag> my_dynrule;
    my_dynrule.set_id(1234);    // set my_dynrule's id to 1234
```

If the `set_id()` function isn't called, the parser id defaults to the address of the rule as its ID, just like the `parser_address_tag` template parameter would do.

Epsilon

The **Epsilon** (`epsilon_p` and `eps_p`) is a multi-purpose parser that returns a zero length match.

Simple Form

In its simplest form, epsilon_p matches the null string and always returns a match of zero length:

```
    epsilon_p // always returns a zero-length match
```

This form is usually used to trigger a **semantic action** unconditionally. For example, it is useful in triggering error messages when a set of alternatives fail:

```
    r = A | B | C | eps_p[error]; // error if A, B, or C fails to match
```

Semantic Predicate

Semantic predicates allow you to attach a function anywhere in the grammar. In this role, the epsilon takes a 0-ary (nullary) function/functor. The run-time function/functor is typically a test that is called upon to resolve ambiguity in the grammar. A parse failure will be reported when the function/functor result evaluates to false. Otherwise an empty match will be reported. The general form is:

```
    eps_p(f) >> rest;
```

The nullary function `f` is called to do a semantic test (say, checking if a symbol is in the **symbol table**). If test returns `true`, `rest` will be evaluated. Otherwise, the production will return early with a no-match without ever touching `rest`.

Syntactic Predicate

Similar to Semantic predicates, Syntactic predicates assert a certain conditional syntax to be satisfied before evaluating another production. This time, epsilon_p accepts a (conditional) parser. The general form is:

```
    eps_p(p) >> rest;
```

If `p` is matched on the input stream then attempt to recognize `rest`. The parser `p` is called to do a syntax check. Regardless of `p`'s success, `eps_p(p)` will always return a zero length match (i.e. the input is not consumed). If test returns `true`, `rest` will be evaluated. Otherwise, the production will return early with a no-match without ever touching `rest`.

Example:

```
    eps_p('0') >> oct_p // note that '0' is actually a ch_p('0')
```

Epsilon here is used as a syntactic predicate. `oct_p` (see **numerics**) is parsed only if we see a leading `'0'`. Wrapping the leading `'0'` inside an epsilon makes the parser not consume anything from the input. If a `'0'` is seen, `epsilon_p` reports a successful match with zero length.

Primitive arguments

Epsilon allows primitive type arguments such as `char`, `int`, `wchar_t`, `char const*`, `wchar_t const*` and so on. Examples:

```
eps_p("hello")        //        same        as
eps_p(str_p("hello"))
eps_p('x') // same as eps_p(ch_p('x'))
```

Inhibiting Semantic Actions

In a syntactic predicate `eps_p(p)`, any semantic action directly or indirectly attached to the conditional parser `p` will not be called. However, semantic actions attached to epsilon itself will always be called. The following code snippets illustrates the behavior:

```
    eps_p(c[f])  // f not called
    eps_p(c)[f]  // f is called
    eps_p[f]     // f is called
```

Actually, the conditional parser `p` is implicitly wrapped in a **`no_actions_d`** directive:

```
    no_actions_d[p]
```

The conditional parser is required to be free from side-effects (semantic actions). The conditional parser's purpose is to resolve ambiguity by looking ahead in the input stream for a certain pattern. Ambiguity and semantic actions do not mix well. On an ambiguous grammar, backtracking happens. And when it happens, we cannot undo the effects of triggered semantic actions.

Negation

Operator `~` is defined for parsers constructed by `epsilon_p`/`eps_p`. It performs negation by complementing the results reported. `~~eps_p(x)` is identical to `eps_p(x)`.

Directives

Parser directives have the form: **directive[expression]**

A directive modifies the behavior of its enclosed expression, essentially *decorating* it. The framework pre-defines a few directives. Clients of the framework are free to define their own directives as needed. Information on how this is done will be provided later. For now, we shall deal only with predefined directives.

lexeme_d

Turns off white space skipping. At the phrase level, the parser ignores white spaces, possibly including comments. Use `lexeme_d` in situations where we want to work at the character level instead of the phrase level. Parsers can be made to work at the character level by enclosing the pertinent parts inside the lexeme_d directive. For example, let us complete the example presented in the **Introduction**. There, we skipped the definition of the `integer` rule. Here's how it is actually defined:

```
    integer = lexeme_d[ !(ch_p('+') | '-') >> +digit ];
```

The `lexeme_d` directive instructs the parser to work on the character level. Without it, the `integer` rule would have allowed erroneous embedded white spaces in inputs such as ***"1 2 345"*** which will be parsed as ***"12345"***.

as_lower_d

There are times when we want to inhibit case sensitivity. The `as_lower_d` directive converts all characters from the input to lower-case.

as_lower_d behavior

It is important to note that only the input is converted to lower case. Parsers enclosed inside the `as_lower_d` expecting upper case characters will fail to parse. Example: `as_lower_d['X']` will never succeed because it expects an upper case `'X'` that the `as_lower_d` directive will never supply.

For example, in Pascal, keywords and identifiers are case insensitive. Pascal ignores the case of letters in identifiers and keywords. Identifiers Id, ID and id are indistinguishable in Pascal. Without the as_lower_d directive, it would be awkward to define a rule that recognizes this. Here's a possibility:

```
    r = str_p("id") | "Id" | "iD" | "ID";
```

Now, try doing that with the case insensitive Pascal keyword ***"BEGIN"***. The as_lower_d directive makes this simple:

```
    r = as_lower_d["begin"];
```

Primitive arguments

The astute reader will notice that we did not explicitly wrap ***"begin"*** inside an str_p. Whenever appropriate, directives should be able to allow primitive types such as char, int, wchar_t, char const*, wchar_t const* and so on. Examples:

```
as_lower_d["hello"]      //      same      as
as_lower_d[str_p("hello")]
as_lower_d['x']        //       same       as
as_lower_d[ch_p('x')]
```

no_actions_d

There are cases where you want **semantic actions** not to be triggered. By enclosing a parser in the no_actions_d directive, all semantic actions directly or indirectly attached to the parser will not fire.

```
    no_actions_d[expression]
```

Tweaking the Scanner Type

How does lexeme_d, as_lower_d and no_actions_d work? These directives do their magic by tweaking the scanner policies. Well, you don't need to know what that means for now. Scanner policies are discussed **later**. However, it is important to note that when the scanner policy is tweaked, the result is a different scanner. Why is this important to note? The **rule** is tied to a particular scanner (one or more scanners, to be precise). If you wrap a rule inside a lexeme_d, as_lower_d or no_actions_d, the compiler will complain about **scanner mismatch** unless you associate the required scanner with the rule.

lexeme_scanner, as_lower_scanner and no_actions_scanner are your friends if the need to wrap a rule inside these directives arise. Learn bout these beasts in the next chapter on **The Scanner and Parsing**.

longest_d

Alternatives in the Spirit parser compiler are short-circuited (see **Operators**). Sometimes, this is not what is desired. The longest_d directive instructs the parser not to short-circuit alternatives enclosed inside this directive, but instead makes the parser try all possible alternatives and choose the one matching the longest portion of the input stream.

Consider the parsing of integers and real numbers:

```
    number = real | integer;
```

A number can be a real or an integer. This grammar is ambiguous. An input ***"1234"*** should potentially match both real and integer. Recall though that alternatives are short-circuited . Thus, for inputs such as above, the real alternative always wins. However, if we swap the alternatives:

```
    number = integer | real;
```

we still have a problem. Now, an input ***"123.456"*** will be partially matched by integer until the decimal point. This is not what we want. The solution here is either to fix the ambiguity by factoring out the common prefixes of real and integer or, if that is not possible nor desired, use the `longest_d` directive:

```
    number = longest_d[ integer | real ];
```

shortest_d

Opposite of the `longest_d` directive.

Multiple alternatives

The `longest_d` and `shortest_d` directives can accept two or more alternatives. Examples:

```
longest[ a | b | c ];
shortest[ a | b | c | d ];
```

limit_d

Ensures that the result of a parser is constrained to a given min..max range (inclusive). If not, then the parser fails and returns a no-match.

Usage:

```
    limit_d(min, max)[expression]
```

This directive is particularly useful in conjunction with parsers that parse specific scalar ranges (for example, **numeric parsers**). Here's a practical example. Although the numeric parsers can be configured to accept only a limited number of digits (say, 0..2), there is no way to limit the result to a range (say -1.0..1.0). This design is deliberate. Doing so would have undermined Spirit's design rule that ***"the client should not pay for features that she does not use"***. We would have stored the min, max values in the numeric parser itself, used or unused. Well, we could get by by using static constants configured by a non-type template parameter, but that is not acceptable because that way, we can only accommodate integers. What about real numbers or user defined numbers such as big-ints?

Example, parse time of the form **HH:MM:SS**:

```
    uint_parser<int, 10, 2, 2> uint2_p;

    r = lexeme_d
        [
                limit_d(0u, 23u)[uint2_p] >> ':'    //  Hours 00..23
            >>  limit_d(0u, 59u)[uint2_p] >> ':'    //  Minutes 00..59
            >>  limit_d(0u, 59u)[uint2_p]           //  Seconds 00..59
        ];
```

min_limit_d

Sometimes, it is useful to unconstrain just the maximum limit. This will allow for an interval that's unbounded in one direction. The directive min_limit_d ensures that the result of a parser is not less than minimun. If not, then the parser fails and returns a no-match.

Usage:

```
    min_limit_d(min)[expression]
```

Example, ensure that a date is not less than 1900

```
    min_limit_d(1900u)[uint_p]
```

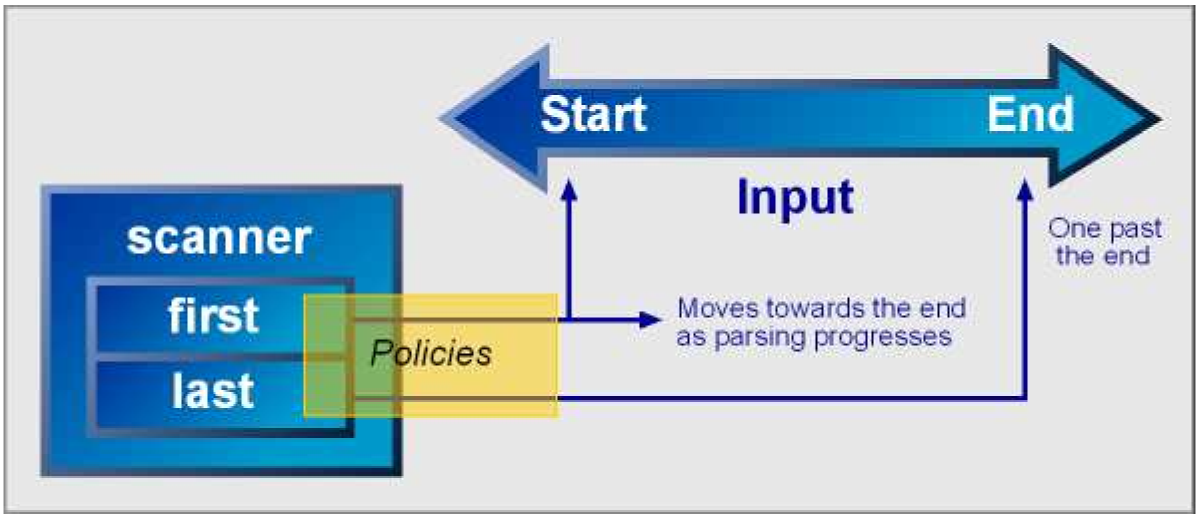
max_limit_d

Opposite of `min_limit_d`. Take note that `limit_d[p]` is equivalent to:

```
    min_limit_d(min)[max_limit_d(max)[p]]
```

The Scanner and Parsing

The **scanner**'s task is to feed the sequential input data stream to the parser. The scanner extracts data from the input, parceling, potentially modifying or filtering, and then finally relegating the result to individual parser elements on demand until the input is exhausted. The scanner is composed of two STL conforming forward iterators, first and last, where first is held by reference and last, by value. The first iterator is held by reference to allow it to be re-positioned. The following diagram illustrates what's happening:



The scanner manages various aspects of the parsing process through a set of policies. There are three sets of policies that govern:

- Iteration and filtering
- Recognition and matching
- Handling semantic actions

These policies are mostly hidden from view and users generally need not know about them. Advanced users might however provide their own policies that override the ones that are already in place to fine tune the parsing process to fit their own needs. We shall see how this can be done. This will be covered in further detail later.

The `scanner` is a template class expecting two parameters: `IteratorT`, the iterator type and `PoliciesT`, its set of policies. `IteratorT` defaults to `char const*` while `PoliciesT` defaults to `scanner_policies<>`, a predefined set of scanner policies that we can use straight out of the box.

```
    template<
        typename IteratorT  = char const*,
        typename PoliciesT  = scanner_policies<> >
    class scanner;
```

Spirit uses the same iterator concepts and interface formally defined by the C++ Standard Template Library (STL). We can use iterators supplied by STL's containers (e.g. `list`, `vector`, `string`, etc.) as is, or perhaps write our own. Iterators can be as simple as a pointer (e.g. `char const*`). At the other end of the spectrum, iterators can be quite complex; for instance, an iterator adapter that wraps a lexer such as LEX.

The Free Parse Functions

The framework provides a couple of free functions to make parsing a snap. These parser functions have two forms. The first form works on the **character level**. The second works on the **phrase level** and asks for a **skip parser**.

The **skip parser** is just about any parser primitive or composite. Its purpose is to move the scanner's `first` iterator to valid tokens by skipping white spaces. In C for instance, the tab ***'\t'***, the newline ***'\n'***, return ***'\r'***, space ***' '*** and characters inside comments ***/*...*/*** are considered as white spaces.

Character level parsing

```
    template <typename IteratorT, typename DerivedT>
    parse_info<IteratorT>
    parse
    (
        IteratorT const&        first,
        IteratorT const&        last,
        parser<DerivedT> const& p
    );
```

```
    template <typename CharT, typename DerivedT>
    parse_info<CharT const*>
    parse
    (
        CharT const*            str,
        parser<DerivedT> const& p
    );
```

There are two variants. The first variant accepts a `first`, `last` iterator pair like you do STL algorithms. The second variant accepts a null terminated string. The last argument is a parser `p` which will be used to parse the input.

Phrase level parsing

```
    template <typename IteratorT, typename ParserT, typename SkipT>
    parse_info<IteratorT>
    parse
    (
        IteratorT const&        first,
        IteratorT const&        last,
        parser<ParserT> const&  p,
        parser<SkipT> const&    skip
    );
```

```
    template <typename CharT, typename ParserT, typename SkipT>
    parse_info<CharT const*>
    parse
    (
        CharT const*            str,
        parser<ParserT> const&  p,
        parser<SkipT> const&    skip
    );
```

Like above, there are two variants. The first variant accepts a `first`, `last` iterator pair like you do STL algorithms. The second variant accepts a null terminated string. The argument `p` is the parser which will be used to parse the input. The last argument `skip` is the skip parser.

The parse_info structure

The functions above return a `parse_info` structure parameterized by the iterator type passed in. The parse_info struct has these members:

parse_info	
stop	Points to the final parse position (i.e The parser recognized and processed the input up to this point)
hit	True if parsing is successful. This may be full: the parser consumed all the input, or partial: the parser consumed only a portion of the input.
full	True when we have a full match (i.e The parser consumed all the input).
length	The number of characters consumed by the parser. This is valid only if we have a successful match (either partial or full).

The phrase_scanner_t and wide_phrase_scanner_t

For convenience, Spirit declares these typedefs:

```
    typedef scanner<char const*, unspecified> phrase_scanner_t;
    typedef scanner<wchar_t const*, unspecified> wide_phrase_scanner_t;
```

These are the exact scanner types used by Spirit on calls to the parse function passing in a `char const*` (C string) or a `wchar_t const*` (wide string) as the first parameter and a `space_p` as skip-parser (the third parameter). For instance, we can use these typedefs to declare some rules. Example:

```
    rule<phrase_scanner_t> my_rule;
    parse("abrakadabra", my_rule, space_p);
```

Direct parsing with Iterators

The free parse functions make it easy for us. By using them, we need not bother with the scanner intricacies. The free parse functions hide the dirty details. However, sometime in the future, we will need to get under the hood. It's nice that we know what we are dealing with when that need comes. We will need to go low-level and call the parser's parse member function directly.

If we wish to work on the **character level**, the procedure is quite simple:

```
    scanner<IteratorT> scan(first, last);

    if (p.parse(scan))
    {
        //  Parsed successfully. If first == last, then we have
        //  a full parse, the parser recognized the input in whole.
    }
    else
    {
        //  Parsing failure. The parser failed to recognize the input
    }
```

The scanner position on an unsucessful match

On a successful match, the input is advanced accordingly. But what happens on an unsuccessful match? Be warned. It might be intuitive to think that the scanner position is reset to its initial position prior to parsing. No, the position is not reset. On an unsuccessful match, the position of the scanner is **undefined**! Usually, it is positioned at the farthest point where the error was found somewhere down the recursive descent. If this behavior is not desired, you may need to position the scanner yourself. The **example in the numerics chapter** illustrates how the scanner position can be saved and later restored.

Where `p` is the parser we want to use, and `first`/`last` are the iterator pairs referring to the input. We just create a scanner given the iterators. The scanner type we will use here uses the default `scanner_policies<>`.

The situation is a bit more complex when we wish to work on the **phrase level**:

```
    typedef skip_parser_iteration_policy<SkipT> iter_policy_t;
    typedef scanner_policies<iter_policy_t> scanner_policies_t;
    typedef scanner<IteratorT, scanner_policies_t> scanner_t;

    iter_policy_t iter_policy(skip);
    scanner_policies_t policies(iter_policy);
    scanner_t scan(first, last, policies);

    if (p.parse(scan))
    {
        //  Parsed successfully. If first == last, then we have
        //  a full parse, the parser recognized the input in whole.
    }
    else
    {
        //  Parsing failure. The parser failed to recognize the input
    }
```

Where `SkipT` is the type of the skip-parser, `skip`. Again, `p` is the parser we want to use, and `first`/`last` are the iterator pairs referring to the input. Given a skip-parser type `SkipT`, `skip_parser_iteration_policy` creates a scanner iteration policy that skips over portions that are recognized by the skip-parser. This may then be used to create a scanner. The `scanner_policies` class wraps all scanner related policies including the iteration policies.

lexeme_scanner

When switching from phrase level to character level parsing, the `lexeme_d` (see **directives.html**) does its magic by disabling the skipping of white spaces. This is done by tweaking the **scanner**. However, when we do this, all parsers inside the lexeme gets a transformed scanner type. This should not be a problem in most cases. However, when rules are called inside the `lexeme_d`, the compiler will choke if the rule does not have the proper scanner type. If a rule must be used inside a `lexeme_d`, the rule's type must be:

```
    rule<lexeme_scanner<ScannerT>::type> r;
```

where `ScannerT` is the actual type of the scanner used. Take note that `lexeme_scanner` will only work for phrase level scanners.

as_lower_scanner

Similarly, the `as_lower_d` does its work by filtering and converting all characters received from the scanner to lower case. This is also done by tweaking the **scanner**. Then again, all parsers inside the `as_lower_d` gets a transformed scanner type. If a rule must be used inside a `as_lower_d`, the rule's type must be:

```
    rule<as_lower_scanner<ScannerT>::type> r;
```

where `ScannerT` is the actual type of the scanner used.

See the techniques section for an **example** of a **grammar** using a **multiple scanner enabled rule**, **lexeme_scanner** and **as_lower_scanner.**

no_actions_scanner

Again, `no_actions_d` directive tweaks the scanner to disable firing semantic actions. Like before, all parsers inside the `no_actions_d` gets a transformed scanner type. If a rule must be used inside a `no_actions_d`, the rule's type must be:

```
    rule<no_actions_scanner<ScannerT>::type> r;
```

where `ScannerT` is the actual type of the scanner used.

Be sure to add "`typename`" before `lexeme_scanner`, `as_lower_scanner` and `no_actions_scanner` when these are used inside a template class or function.

See **no_actions.cpp**. This is part of the Spirit distribution.

The Grammar

The **grammar** encapsulates a set of rules. The `grammar` class is a protocol base class. It is essentially an interface contract. The `grammar` is a template class that is parameterized by its derived class, `DerivedT`, and its context, `ContextT`. The template parameter ContextT defaults to `parser_context`, a predefined context.

You need not be concerned at all with the ContextT template parameter unless you wish to tweak the low level behavior of the grammar. Detailed information on the ContextT template parameter is provided **elsewhere**. The `grammar` relies on the template parameter DerivedT, a grammar subclass to define the actual rules.

Presented below is the public API. There may actually be more template parameters after `ContextT`. Everything after the `ContextT` parameter should not be of concern to the client and are strictly for internal use only.

```
    template<
        typename DerivedT,
        typename ContextT = parser_context<> >
    struct grammar;
```

Grammar definition

A concrete sub-class inheriting from `grammar` is expected to have a nested template class (or struct) named `definition`:

- It is a nested template class with a typename `ScannerT` parameter.
- Its constructor defines the grammar rules.
- Its constructor is passed in a reference to the actual grammar `self`.
- It has a member function named `start` that returns a reference to the start `rule`.

Grammar skeleton

```
    struct my_grammar : public grammar<my_grammar>
    {
        template <typename ScannerT>
        struct definition
        {
            rule<ScannerT>  r;
            definition(my_grammar const& self)  { r = /*..define here..*/; }
            rule<ScannerT> const& start() const { return r; }
        };
    };
```

Decoupling the scanner type from the rules that form a grammar allows the grammar to be used in different contexts possibly using different scanners. We do not care what scanner we are dealing with. The user-defined `my_grammar` can be used with **any** type of scanner. Unlike the rule, the grammar is not tied to a specific scanner type. See **"Scanner Business"** to see why this is important and to gain further understanding on this scanner-rule coupling problem.

Instantiating and using my_grammar

Our grammar above may be instantiated and put into action:

```
    my_grammar g;

    if (parse(first, last, g, space_p).full)
        cout << "parsing succeeded\n";
    else
        cout << "parsing failed\n";
```

`my_grammar` **IS-A** parser and can be used anywhere a parser is expected, even referenced by another rule:

```
    rule<>  r = g >> str_p("cool huh?");
```

Referencing grammars

Like the rule, the grammar is also held by reference when it is placed in the right hand side of an EBNF expression. It is the responsibility of the client to ensure that the referenced grammar stays in scope and does not get destructed while it is being referenced.

Full Grammar Example

Recalling our original calculator example, here it is now rewritten using a grammar:

```
    struct calculator : public grammar<calculator>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(calculator const& self)
            {
                group       = '(' >> expression >> ')';
                factor      = integer | group;
                term        = factor >> *(('*' >> factor) | ('/' >> factor));
                expression  = term >> *(('+' >> term) | ('-' >> term));
            }

            rule<ScannerT> expression, term, factor, group;

            rule<ScannerT> const&
            start() const { return expression; }
        };
    };
```

A fully working example with **semantic actions** can be **viewed here**. This is part of the Spirit distribution.

self

You might notice that the definition of the grammar has a constructor that accepts a const reference to the outer grammar. In the example above, notice that calculator::definition takes in a calculator const& self. While this is unused in the example above, in many cases, this is very useful. The self argument is the definition's window to the outside world. For example, the calculator class might have a reference to some state information that the definition can update while parsing proceeds through **semantic actions**.

Grammar Capsules

As a grammar becomes complicated, it is a good idea to group parts into logical modules. For instance, when writing a language, it might be wise to put expressions and statements into separate grammar capsules. The grammar takes advantage of the encapsulation properties of C++ classes. The declarative nature of classes makes it a perfect fit for the definition of grammars. Since the grammar is nothing more than a class declaration, we can conveniently publish it in header files. The idea is that once written and fully tested, a grammar can be reused in many contexts. We now have the notion of grammar libraries.

Reentrancy and multithreading

An instance of a grammar may be used in different places multiple times without any problem. The implementation is tuned to allow this at the expense of some overhead. However, we can save considerable cycles and bytes if we are certain that a grammar will only have a single instance. If this is desired, simply define BOOST_SPIRIT_SINGLE_GRAMMAR_INSTANCE before including any spirit header files.

```
    #define BOOST_SPIRIT_SINGLE_GRAMMAR_INSTANCE
```

On the other hand, if a grammar is intended to be used in multithreaded code, we should then define BOOST_SPIRIT_THREADSAFE before including any spirit header files. In this case it will also be required to link against **Boost.Threads**

```
    #define BOOST_SPIRIT_THREADSAFE
```

Using more than one grammar start rule

Sometimes it is desirable to have more than one visible entry point to a grammar (apart from the start rule). To allow additional start points, Spirit provides a helper template grammar_def, which may be used as a base class for the definition subclass of your grammar. Here's an example:

```
    // this header has to be explicitly included
    #include <boost/spirit/utility/grammar_def.hpp>

    struct calculator2 : public grammar<calculator2>
    {
        enum
        {
            expression = 0,
            term = 1,
            factor = 2,
        };

        template <typename ScannerT>
        struct definition
        : public grammar_def<rule<ScannerT>, same, same>
        {
            definition(calculator2 const& self)
            {
                group       = '(' >> expression >> ')';
                factor      = integer | group;
                term        = factor >> *(('*' >> factor) | ('/' >> factor));
                expression  = term >> *(('+' >> term) | ('-' >> term));

                this->start_parsers(expression, term, factor);
            }

            rule<ScannerT> expression, term, factor, group;
        };
    };
```

The grammar_def template has to be instantiated with the types of all the rules you wish to make visible from outside the grammar:

```
    grammar_def<rule<ScannerT>, same, same>
```

The shorthand notation same is used to indicate that the same type be used as specified by the previous template parameter (e.g. rule<ScannerT>). Obviously, same may not be used as the first template parameter.

grammar_def start types

It may not be obvious, but it is interesting to note that aside from rule<>s, any parser type may be specified (e.g. chlit<>, strlit<>, int_parser<>, etc.).

Using the grammar_def class, there is no need to provide a start()member function anymore. Instead, you'll have to insert a call to the this->start_parsers() (which is a member function of the grammar_def template) to define the start symbols for your grammar. Note that the number and the sequence of the rules used as the parameters to the start_parsers() function should match the types specified in the grammar_def template:

```
    this->start_parsers(expression, term, factor);
```

The grammar entry point may be specified using the following syntax:

```
    g.use_parser<N>() // Where g is your grammar and N is the Nth entry.
```

This sample shows how to use the term rule from the calculator2 grammar above:

```
    calculator2 g;

    if (parse(
                first, last,
                g.use_parser<calculator2::term>(),
                space_p
            ).full)
    {
        cout << "parsing succeeded\n";
    }
    else {
        cout << "parsing failed\n";
    }
```

The template parameter for the `use_parser<>` template type should be the zero based index into the list of rules specified in the `start_parsers()` function call.

> **`use_parser<0>`**
>
> Note, that using *0* (zero) as the template parameter to `use_parser` is equivalent to using the start rule, exported by conventional means through the `start()` function, as shown in the first **`calculator`** sample above. So this notation may be used even for grammars exporting one rule through its `start()` function only. On the other hand, calling a `grammar` without the `use_parser` notation will execute the rule specified as the first parameter to the `start_parsers()` function.

The maximum number of usable start rules is limited by the preprocessor constant:

```
    BOOST_SPIRIT_GRAMMAR_STARTRULE_TYPE_LIMIT // defaults to 3
```

Subrules

Spirit is implemented using expression templates. This is a very powerful technique. Along with its power comes some complications. We almost take for granted that when we write `i | j >> k` where `i`, `j` and `k` are all integers the result is still an integer. Yet, with expression templates, the same expression `i | j >> k` where `i`, `j` and `k` are of type `T`, the result is a complex composite type [see **Basic Concepts**]. Spirit expressions, which are combinations of primitives and composites yield an infinite set of new types. One problem is that C++ offers no easy facility to deduce the type of an arbitrarily complex expression that yields a complex type. Thus, while it is easy to write:

```
    int r = i | j >> k; // where i, j, and k are ints
```

Expression templates yield an endless supply of types. Without the **rule**, there is no easy way to do this in C++ if `i`, `j` and `k` are Spirit parsers:

```
    <what_type???> r = i | j >> k; // where i, j, and k are Spirit parsers
```

If `i`, `j` and `k` are all `chlit<>` objects, the type that we want is:

```
    typedef
        alternative<
            chlit<>      //  i
          , sequence<
                chlit<>  //  j
              , chlit<>  //  k
            >
        >
    rule_t;

    rule_t r = i | j >> k; // where i, j, and k are chlit<> objects
```

We deliberately formatted the type declaration nicely to make it understandable. Try that with a more complex expression. While it can be done, explicitly spelling out the type of a Spirit expression template is tedious and error prone. The right hand side (rhs) has to mirror the type of the left hand side (lhs). (Yet, if you still wish to do it, see this **link** for a technique).

typeof and auto

Some compilers already support the `typeof` keyword. This can be used to free us from having to explicitly type the type (pun intentional). Using the `typeof`, we can rewrite the Spirit expression above as:

```
typeof(i | j >> k) r = i | j >> k;
```

While this is better than having to explicitly declare a complex type, it is redundant, error prone and still an eye sore. The expression is typed twice. The only way to simplify this is to introduce a macro (See this **link** for more information).

David Abrahams proposed in comp.std.c++ to reuse the `auto` keyword for type deduced variables. This has been extensibly discussed in **boost.org**. Example:

```
auto r = i | j >> k;
```

Once such a C++ extension is accepted into the standard, this would be a neat solution and a nice fit for our purpose. It's not a complete solution though since there are still situations where we do not know the rhs beforehand; for instance when pre-declaring cyclic dependent rules.

Fortunately, rules come to the rescue. Rules can capture the type of the expression assigned to it. Thus:

```
    rule<> r = i | j >> k;  // where i, j, and k are chlit<> objects
```

It might not be apparent but behind the scenes, plain rules are actually implemented using a pointer to a runtime polymorphic abstract class that holds the dynamic type of the parser assigned to it. When a Spirit expression is assigned to a rule, its type is encapsulated in a concrete subclass of the abstract class. A virtual parse function delegates the parsing to the encapsulated object.

Rules have drawbacks though:

- It is coupled to a specific scanner type. The rule is tied to a specific scanner [see **The Scanner Business**].
- The rule's parse member function has a virtual function call overhead that cannot be inlined.

Static rules: subrules

The subrule is a fully static version of the rule. The subrule does not have the drawbacks listed above.

- The subrule is not tied to a specific scanner so just about any scanner type may be used
- The subrule also allows aggressive inlining since there are no virtual function calls

```
    template<int ID, typename ContextT = parser_context<> >
    class subrule;
```

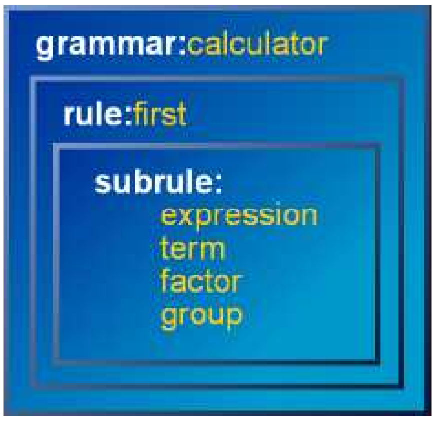
The first template parameter gives the subrule an identification tag. Like the **rule**, there is a ContextT template parameter that defaults to `parser_context`. You need not be concerned at all with the `ContextT` template parameter unless you wish to tweak the low level behavior of the subrule. Detailed information on the `ContextT` template parameter is provided **elsewhere**.

Presented above is the public API. There may actually be more template parameters after `ContextT`. Everything after the `ContextT` parameter should not be of concern to the client and are strictly for internal use only.

Apart from a few minor differences, the subrule follows the usage and syntax of the rule closely. Here's the calculator grammar using subrules:

```
    struct calculator : public grammar<calculator>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(calculator const& self)
            {
                first =
                (
                    expression  = term >> *(('+' >> term) | ('-' >> term)),
                    term        = factor >> *(('*' >> factor) | ('/' >> factor)),
                    factor      = integer | group,
                    group       = '(' >> expression >> ')'
                );
            }

            subrule<0>  expression;
            subrule<1>  term;
            subrule<2>  factor;
            subrule<3>  group;

            rule<ScannerT> first;
            rule<ScannerT> const&
            start() const { return first; }
        };
    };
```

A fully working example with **semantic actions** can be **viewed here**. This is part of the Spirit distribution.



The subrule as an efficient version of the rule. Compiler optimizations such as aggressive inlining help reduce the code size and increase performance significantly.

The subrule is not a panacea however. Subrules push the C++ compiler hard to its knees. For example, current compilers have a limit on recursion depth that may not be exceeded. Don't even think about writing a full pascal grammar using subrules alone. A grammar using subrules is a single C++ expression. Current C++ compilers cannot handle very complex expressions very well. Finally, a plain rule is still needed to act as place holder for subrules.

The code above is a good example of the recommended way to use subrules. Notice the hierarchy. We have a grammar that encapsulates the whole calculator. The start rule is a plain rule that holds the set of subrules. The subrules in turn defines the actual details of the grammar.

Template instantiation depth

Spirit pushes the C++ compiler hard. Current C++ compilers cannot handle very complex heavily nested 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 have a relatively complex grammar.

Microsoft Visual C++ can take greater than 1000 for both template class and function instantiation depths. However, the linker chokes with deep template function instantiation unless inline recursion depth is set using these pragmas:

```
#pragma inline_depth(255)
#pragma inline_recursion(on)
```

Perhaps this limitations no longer applies to more current versions of these compilers. Be sure to check your compiler documentation.

This setup gives a good balance. The subrules do all the work. Each grammar will have only one rule: `first`. The rule is used just to hold the subrules and make them visible to the grammar.

The subrule definition

Like the rule, the expression after assignment operator `=` defines the subrule:

```
    identifier = expression
```

Unlike rules, subrules may be defined only once. Redefining a subrule is illegal and will result to a compile time assertion.

Separators [,]

While rules are terminated by the semicollon `';'`. Subrules are not terminated but are separated by the comma: `','`. Like Pascal statements, the last subrule in a group may not have a trailing comma.

```
    a = ch_p('a'),
    b = ch_p('b'),
    c = ch_p('c'), // BAD, trailing comma
```

```
    a = ch_p('a'),
    b = ch_p('b'),
    c = ch_p('c')  // OK
```

The start subrule

Unlike rules, parsing proceeds from the start subrule. The first (topmost) subrule in a group of subrules is called the **start subrule**. In our example above, `expression` is the start subrule. When a group of subrules is called forth, the start subrule `expression` is called first.

IDs

Each subrule has a corresponding ID; an integral constant that uniquely specifies the subrule. Our example above has four subrules. They are declared as:

```
    subrule<0>  expression;
    subrule<1>  term;
    subrule<2>  factor;
    subrule<3>  group;
```

Aliases

It is possible to have subrules with similar IDs. A subrule with a similar ID to will be an alias of the other. Both subrules may be used interchangeably.

```
    subrule<0>  a;
    subrule<0>  alias;  // alias of a
```

Groups: scope and nesting

The scope of a subrule and its definition is the enclosing group, typically (and by convention) enclosed inside the parentheses. IDs outside a scope are not directly visible. Inner subrule groups can be nested by enclosing each sub-group inside another set of parentheses. Each group is unique and acts independently. Consequently, while it may not be advisable to do so, a subrule in a group may share the same ID as a subrule in another group since both groups are independent of each other.

```
    subrule<0> a;
    subrule<1> b;
    subrule<0> c;
    subrule<1> d;

    (                       // outer subrule group, scope of a and b
        a = ch_p('a'),
        b =
        (                   // inner subrule group, scope of b and c
            c = ch_p('c'),
            d = ch_p('d')
        )
    )
```

Subrule IDs need to be unique only within a group. A grammar is an implicit group. Furthermore, even subrules in a grammar may have the same IDs without clashing if they are inside a group. Subrules may be explicitly grouped using the parentheses. Parenthesized groups have unique scopes. In the code above, the outer subrule group defines the subrules `a` and `b` while the inner subrule group defines the subrules `c` and `d`. Notice that the definition of `b` is the inner subrule.

Semantic Actions

Semantic actions have the form: **expression[action]**

Ultimately, after having defined our grammar and having generated a corresponding parser, we will need to produce some output and do some work besides syntax analysis; unless, of course, what we want is merely to check for the conformance of an input with our grammar, which is very seldom the case. Semantic actions may be attached to any expression at any level within the parser hierarchy. An action is a C/C++ function or function object that will be called if a match is found in the particular context where it is attached. The action function serves as a hook into the parser and may be used to, for example:

- Generate output from the parser (ASTs, for example)
- Report warnings or errors
- Manage symbol tables

Generic Semantic Actions (Transduction Interface)

A generic semantic action can be any free function or function object that is compatible with the interface:

```
    void f(IteratorT first, IteratorT last);
```

where `IteratorT` is the type of iterator used, `first` points to the current input and `last` points to one after the end of the input (identical to STL iterator ranges). A function object (functor) should have a member `operator()` with the same signature as above:

```
    struct my_functor
    {
        void operator()(IteratorT first, IteratorT last) const;
    };
```

Iterators pointing to the matching portion of the input are passed into the function/functor.

In general, semantic actions accept the first-last iterator pair. This is the transduction interface. The action functions or functors receive the unprocessed data representing the matching production directly from the input. In many cases, this is sufficient. Examples are source to source translation, pre-processing, etc.

Example:

```
    void
    my_action(char const* first, char const* last)
    {
        std::string str(first, last);
        std::cout << str << std::endl;
    }

    rule<> myrule = (a | b | *(c >> d))[&my_action];
```

The function `my_action` will be called whenever the expression `(a | b | *(c >> d)` matches a portion of the input stream while parsing. Two iterators, `first` and `last`, are passed into the function. These iterators point to the start and end, respectively, of the portion of input stream where the match is found.

Const-ness:

With functors, take note that the `operator()` should be `const`. This implies that functors are immutable. One may wish to have some member variables that are modified when the action gets called. This is not a good idea. First of all, functors are preferably lightweight. Functors are passed around a lot and it would incur a lot of overhead if the functors are heavily laden. Second, functors are passed by value. Thus, the actual functor object that finally attaches to the parser, will surely not be the original instance supplied by the client. What this means is that changes to a functor's state will not affect the original functor that the client passed in since they are distinct copies. If a functor needs to update some state variables, which is often the case, it is better to use references to external data. The following example shows how this can be done:

```
    struct my_functor
    {
        my_functor(std::string& str_)
        : str(str_) {}

        void
        operator()(IteratorT first, IteratorT last) const
        {
            str.assign_a(first, last);
        }

        std::string& str;
    };
```

Full Example:

Here now is our calculator enhanced with semantic actions:

```
    namespace
    {
        void    do_int(char const* str, char const* end)
        {
            string  s(str, end);
            cout << "PUSH(" << s << ')' << endl;
        }

        void    do_add(char const*, char const*)    { cout << "ADD\n"; }
        void    do_subt(char const*, char const*)   { cout << "SUBTRACT\n"; }
        void    do_mult(char const*, char const*)   { cout << "MULTIPLY\n"; }
        void    do_div(char const*, char const*)    { cout << "DIVIDE\n"; }
        void    do_neg(char const*, char const*)    { cout << "NEGATE\n"; }
    }
```

We augment our grammar with semantic actions:

```
    struct calculator : public grammar<calculator>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(calculator const& self)
            {
                expression
                =   term
                    >> *(   ('+' >> term)[&do_add]
                        |   ('-' >> term)[&do_subt]
                        )
                ;

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

            factor
                =   lexeme_d[(+digit_p)[&do_int]]
                |   '(' >> expression >> ')'
                |   ('-' >> factor)[&do_neg]
                |   ('+' >> factor)
                ;
        }

        rule<ScannerT> expression, term, factor;

        rule<ScannerT> const&
        start() const { return expression; }
    };
};
```

Feeding in the expression `(-1 + 2) * (3 + -4)`, for example, to the rule `expression` will produce the expected output:

```
-1
2
ADD
3
-4
ADD
MULT
```

which, by the way, is the Reverse Polish Notation (RPN) of the given expression, reminiscent of some primitive calculators and the language Forth.

View the complete source code here. This is part of the Spirit distribution.

Specialized Actions

In general, semantic actions accept the first-last iterator pair. There are situations though where we might want to pass data in its processed form. A concrete example is the numeric parser. It is unwise to pass unprocessed data to a semantic action attached to a numeric parser and just throw away what has been parsed by the parser. We want to pass the actual parsed number.

The function and functor signature of a semantic action varies depending on the parser where it is attached to. The following table lists the parsers that accept unique signatures.

Unless explicitly stated in the documentation of a specific parser type, parsers not included in the list by default expect the generic signature as explained above.

Numeric Actions

Applies to:

- uint_p
- int_p
- ureal_p
- real_p

Signature for functions:

```
    void func(NumT val);
```

Signature for functors:

```
    struct ftor
    {
        void operator()(NumT val) const;
    };
```

Where `NumT` is any primitive numeric type such as `int`, `long`, `float`, `double`, etc., or a user defined numeric type such as big_int. `NumT` is the same type used as template parameter to `uint_p`, `int_p`, `ureal_p` or `real_p`. The parsed number is passed into the function/functor.

Character Actions

Applies to:

- chlit, ch_p
- range, range_p
- anychar
- alnum, alpha
- cntrl, digit
- graph, lower
- print, punct
- space, upper
- xdigit

Signature for functions:

```
    void func(CharT ch);
```

Signature for functors:

```
    struct ftor
    {
        void operator()(CharT ch) const;
    };
```

Where `CharT` is the value_type of the iterator used in parsing. A `char const*` iterator for example has a `value_type` of `char`. The matching character is passed into the function/functor.

Cascading Actions

Actions can be cascaded. Cascaded actions also inherit the function/functor interface of the original. For example:

```
    uint_p[fa][fb][fc]
```

Here, the functors `fa`, `fb` and `fc` all expect the signature `void operator()(unsigned n) const`.

Directives and Actions

Directives inherit the the function/functor interface of the subject it is enclosing. Example:

```
    as_lower_d[ch_p('x')][f]
```

Here, the functor `f` expects the signature `void operator()(char ch) const`, assuming that the iterator used is a `char const*`.

In-depth: The Parser

What makes Spirit tick? Now on to some details... The parser class is the most fundamental entity in the framework. A parser accepts a scanner comprised of a first-last iterator pair and returns a match object as its result. The iterators delimit the data currently being parsed. The match object evaluates to true if the parse succeeds, in which case the input is advanced accordingly. Each parser can represent a specific pattern or algorithm, or it can be a more complex parser formed as a composition of other parsers.

All parsers inherit from the base template class, parser:

```
template <typename DerivedT>
struct parser
{
    /*...*/

    DerivedT& derived();
    DerivedT const& derived() const;
};
```

This class is a protocol base class for all parsers. The parser class does not really know how to parse anything but instead relies on the template parameter `DerivedT` to do the actual parsing. This technique is known as the **"Curiously Recurring Template Pattern"** in template meta-programming circles. This inheritance strategy gives us the power of polymorphism without the virtual function overhead. In essence this is a way to implement **compile time polymorphism**.

parser_category_t

Each derived parser has a typedef `parser_category_t` that defines its category. By default, if one is not specified, it will inherit from the base parser class which typedefs its parser_category_t as `plain_parser_category`. Some template classes are provided to distinguish different types of parsers. The following categories are the most generic. More specific types may inherit from these.

Parser categories

`plain_parser_category`	Your plain vanilla parser
`binary_parser_category`	A parser that has subject a and b (e.g. alternative)
`unary_parser_category`	A parser that has single subject (e.g. kleene star)
`action_parser_category`	A parser with an attached semantic action

```
    struct plain_parser_category {};
    struct binary_parser_category       : plain_parser_category {};
    struct unary_parser_category        : plain_parser_category {};
    struct action_parser_category       : unary_parser_category {};
```

embed_t

Each parser has a typedef `embed_t`. This typedef specifies how a parser is embedded in a composite. By default, if one is not specified, the parser will be embedded by value. That is, a copy of the parser is placed as a member variable of the composite. Most parsers are embedded by value. In certain situations however, this is not desirable or possible. One particular example is the **rule**. The rule, unlike other parsers is embedded by reference.

The match

The match holds the result of a parser. A match object evaluates to true when a succesful match is found, otherwise false. The length of the match is the number of characters (or tokens) that is successfully matched. This can be queried through its `length()` member function. A negative value means that the match is unsucessful.

Each parser may have an associated attribute. This attribute is also returned back to the client on a successful parse through the match object. We can get this attribute via the match's `value()` member function. Be warned though that the match's attribute may be invalid, in which case, getting the attribute will result in an exception. The member function `has_valid_attribute()` can be queried to know if it is safe to get the match's attribute. The attribute may be set anytime through the member function `value(v)`where `v` is the new attribute value.

A match attribute is valid:

- on a successful match
- when its value is set through the `value(val)` member function
- if it is assigned or copied from a compatible match object (e.g. `match<double>` from `match<int>`) with a valid attribute. A match object `A` is compatible with another match object `B` if the attribute type of `A` can be assigned from the attribute type of `B` (i.e. `a = b;` must compile).

The match attribute is undefined:

- on an unsuccessful match
- when an attempt to copy or assign from another match object with an incompatible attribute type (e.g. `match<std::string>` from `match<int>`).

The match class:

```
    template <typename T>
    class match
    {
    public:

        /*...*/

        typedef T attr_t;

                    operator safe_bool() const; // convertible to a bool
        int         length() const;
        bool        has_valid_attribute() const;
        void        value(T const&) const;
        T const&    value();
    };
```

match_result

It has been mentioned repeatedly that the parser returns a match object as its result. This is a simplification. Actually, for the sake of genericity, parsers are really not hard-coded to return a match object. More accurately, a parser returns an object that adheres to a conceptual interface, of which the match is an example. Nevertheless, we shall call the result type of a parser a match object regardless if it is actually a match class, a derivative or a totally unrelated type.

Meta-functions

What are meta-functions? We all know how functions look like. In simplest terms, a function accepts some arguments and returns a result. Here is the function we all love so much:

```
int identity_func(int arg)
{ return arg; } // return the argument arg
```

Meta-functions are essentially the same. These beasts also accept arguments and return a result. However, while functions work at runtime on values, meta-functions work at compile time on types (or constants, but we shall deal only with types). The meta-function is a template class (or struct). The template parameters are the arguments to the meta-function and a typedef within the class is the meta-function's return type. Here is the corresponding meta-function:

```
template <typename ArgT>
struct identity_meta_func
{ typedef ArgT type; } // return the argument
ArgT
```

The meta-function above is invoked as:

```
typename identity_meta_func<ArgT>::type
```

By convention, meta-functions return the result through the typedef `type`. Take note that `typename` is only required within templates.

The actual match type used by the parser depends on two types: the parser's attribute type and the scanner type. `match_result` is the meta-function that returns the desired match type given an attribute type and a scanner type.

Usage:

```
    typename match_result<ScannerT, T>::type
```

The meta-function basically answers the question "given a scanner type `ScannerT` and an attribute type `T`, what is the desired match type?" [`typename` is only required within templates].

The parse member function

Concrete sub-classes inheriting from parser must have a corresponding member function `parse(...)` compatible with the conceptual Interface:

```
    template <typename ScannerT>
    RT
    parse(ScannerT const& scan) const;
```

where `RT` is the desired return type of the parser.

The parser result

Concrete sub-classes inheriting from parser in most cases need to have a nested meta-function `result` that returns the result `type` of the parser's parse member function, given a scanner type. The meta-function has the form:

```
    template <typename ScannerT>
    struct result
    {
        typedef RT type;
    };
```

where `RT` is the desired return type of the parser. This is usually, but not always, dependent on the template parameter `ScannerT`. For example, given an attribute type `int`, we can use the match_result metafunction:

```
    template <typename ScannerT>
    struct result
    {
        typedef typename match_result<ScannerT, int>::type type;
    };
```

If a parser does not supply a result metafunction, a default is provided by the base parser class. The default is declared as:

```
    template <typename ScannerT>
    struct result
    {
        typedef typename match_result<ScannerT, nil_t>::type type;
    };
```

Without a result metafunction, notice that the parser's default attribute is `nil_t` (i.e. the parser has no attribute).

parser_result

Given a a scanner type `ScannerT` and a parser type `ParserT`, what will be the actual result of the parser? The answer to this question is provided to by the `parser_result` meta-function.

Usage:

```
    typename parser_result<ParserT, ScannerT>::type
```

In general, the meta-function just forwards the invocation to the parser's result meta-function:

```
    template <typename ParserT, typename ScannerT>
    struct parser_result
    {
        typedef typename ParserT::template result<ScannerT>::type type;
    };
```

This is similar to a global function calling a member function. Most of the time, the usage above is equivalent to:

```
    typename ParserT::template result<ScannerT>::type
```

Yet, this should not be relied upon to be true all the time because the parser_result metafunction might be specialized for specific parser and/or scanner types.

The parser_result metafunction makes the signature of the required parse member function almost canonical:

```
    template <typename ScannerT>
    typename parser_result<self_t, ScannerT>::type
    parse(ScannerT const& scan) const;
```

where `self_t` is a typedef to the parser.

parser class declaration

```
    template <typename DerivedT>
    struct parser
    {
        typedef DerivedT                embed_t;
        typedef DerivedT                derived_t;
        typedef plain_parser_category   parser_category_t;

        template <typename ScannerT>
        struct result
        {
            typedef typename match_result<ScannerT, nil_t>::type type;
        };

        DerivedT& derived();
        DerivedT const& derived() const;

        template <typename ActionT>
        action<DerivedT, ActionT>
        operator[](ActionT const& actor) const;
    };
```

In-depth: The Scanner

Basic Scanner API

class scanner

`value_t`	typedef: The value type of the scanner's iterator
`ref_t`	typedef: The reference type of the scanner's iterator
`bool at_end() const`	Returns true if the input is exhausted
`value_t operator*() const`	Dereference/get a `value_t` from the input
`scanner const& operator++()`	move the scanner forward
`IteratorT& first`	The iterator pointing to the current input position. Held by reference
`IteratorT const last`	The iterator pointing to the end of the input. Held by value

The basic behavior of the scanner is handled by policies. The actual execution of the scanner's public member functions listed in the table above is implemented by the scanner policies.

Three sets of policies govern the behavior of the scanner. These policies make it possible to extend Spirit non-intrusively. The scanner policies allow the core-functionality to be extended without requiring any potentially destabilizing changes to the code. A library writer might provide her own policies that override the ones that are already in place to fine tune the parsing process to fit her own needs. Layers above the core might also want to take advantage of this policy based machanism. Abstract syntax tree generation, debuggers and lexers come to mind.

There are three sets of policies that govern:

- Iteration and filtering
- Recognition and matching
- Handling semantic actions

iteration_policy

Here are the default policies that govern iteration and filtering:

```
    struct iteration_policy
    {
        template <typename ScannerT>
        void
        advance(ScannerT const& scan) const
        { ++scan.first; }
        template <typename ScannerT>
        bool at_end(ScannerT const& scan) const
        { return scan.first == scan.last; }

        template <typename T>
        T filter(T ch) const
        { return ch; }

        template <typename ScannerT>
        typename ScannerT::ref_t
        get(ScannerT const& scan) const
        { return *scan.first; }
    };
```

Iteration and filtering policies

advance	Move the iterator forward
at_end	Return true if the input is exhausted
filter	Filter a character read from the input
get	Read a character from the input

The following code snippet demonstrates a simple policy that converts all characters to lower case:

```
    struct inhibit_case_iteration_policy : public iteration_policy
    {
        template <typename CharT>
        CharT filter(CharT ch) const
        {
            return std::tolower(ch);
        }
    };
```

match_policy

Here are the default policies that govern recognition and matching:

```
    struct match_policy
    {
        template <typename T>
        struct result
        {
            typedef match<T> type;
        };

        const match<nil_t>
        no_match() const
        {
            return match<nil_t>();
        }

        const match<nil_t>
        empty_match() const
        {
            return match<nil_t>(0, nil_t());
        }
        template <typename AttrT, typename IteratorT>
        match<AttrT>
        create_match(
            std::size_t         length,
            AttrT const&        val,
            IteratorT const&    /*first*/,
            IteratorT const&    /*last*/) const
        {
            return match<AttrT>(length, val);
        }

        template <typename MatchT, typename IteratorT>
        void
        group_match(
            MatchT&             /*m*/,
            parser_id const&    /*id*/,
            IteratorT const&    /*first*/,
            IteratorT const&    /*last*/) const {}

        template <typename Match1T, typename Match2T>
        void
        concat_match(Match1T& l, Match2T const& r) const
        {
            l.concat(r);
        }
    };
```

Recognition and matching

result	A metafunction that returns a match type given an attribute type (see In-depth: The Parser)
no_match	Create a failed match
empty_match	Create an empty match. An empty match is a successful epsilon match (matching length == 0)
create_match	Create a match given the matching length, an attribute and the iterator pair pointing to the matching portion of the input
group_match	For non terminals such as rules, this is called after a successful match has been made to allow post processing
concat_match	Concatenate two match objects

action_policy

The action policy has only one function for handling semantic actions:

```
    struct action_policy
    {
        template <typename ActorT, typename AttrT, typename IteratorT>
        void
        do_action(
            ActorT const&       actor,
            AttrT const&        val,
            IteratorT const&    first,
            IteratorT const&    last) const;
    };
```

The default action policy forwards to:

```
    actor(first, last);
```

If the attribute `val` is of type nil_t. Otherwise:

```
    actor(val);
```

scanner_policies mixer

The class `scanner_policies` combines the three scanner policy classes above into one:

```
    template <
        typename IterationPolicyT   = iteration_policy,
        typename MatchPolicyT       = match_policy,
        typename ActionPolicyT      = action_policy>
    struct scanner_policies;
```

This *mixer* class inherits from all the three policies. This scanner_policies class is then used to parameterize the scanner:

```
    template <
        typename IteratorT = char const*,
        typename PoliciesT = scanner_policies<> >
    class scanner;
```

The scanner in turn inherits from the PoliciesT.

Rebinding Policies

The scanner can be made to rebind to a different set of policies anytime. It has a member function `change_policies(new_policies)`. Given a new set of policies, this member function creates a new scanner with the new set of policies. The result type of the *rebound* scanner can be can be obtained by calling the metafunction:

```
    rebind_scanner_policies<ScannerT, PoliciesT>::type
```

Rebinding Iterators

The scanner can also be made to rebind to a different iterator type anytime. It has a member function `change_iterator(first, last)`. Given a new pair of iterator of type different from the ones held by the scanner, this member function creates a new scanner with the new pair of iterators. The result type of the *rebound* scanner can be can be obtained by calling the metafunction:

```
    rebind_scanner_iterator<ScannerT, IteratorT>::type
```

In-depth: The Parser Context

Overview

The parser's **context** is yet another concept. An instance (object) of the `context` class is created before a non-terminal starts parsing and is destructed after parsing has concluded. A non-terminal is either a `rule`, a `subrule`, or a `grammar`. Non-terminals have a `ContextT` template parameter. The following pseudo code depicts what's happening when a non-terminal is invoked:

```
    return_type
    a_non_terminal::parse(ScannerT const& scan)
    {
        context_t ctx(/**/);
        ctx.pre_parse(/**/);

        //  main parse code of the non-terminal here...

        return ctx.post_parse(/**/);
    }
```

The context is provided for extensibility. Its main purpose is to expose the start and end of the non-terminal's parse member function to accommodate external hooks. We can extend the non-terminal in a multitude of ways by writing specialized context classes, without modifying the class itself. For example, we can make the non-terminal emit debug diagnostics information by writing a context class that prints out the current state of the scanner at each point in the parse traversal where the non-terminal is invoked.

Example of a parser context that prints out debug information:

```
    pre_parse:      non-terminal XXX is entered. The current state of the input
                    is "hello world, this is a test"

    post_parse:     non-terminal XXX has concluded, the non-terminal matched "h
                    The current state of the input is ", this is a test"
```

Most of the time, the context will be invisible from the user's view. In general, clients of the framework need not deal directly nor even know about contexts. Power users, however, might find some use of contexts. Thus, this is part of the public API. Other parts of the framework in other layers above the core take advantage of the context to extend non-terminals.

Class declaration

The `parser_context` class is the default context class that the non-terminal uses.

```
    template <typename AttrT = nil_t>
    struct parser_context
    {
        typedef AttrT attr_t;
        typedef implementation_defined base_t;
        typedef parser_context_linker<parser_context<AttrT> > context_linker_t;
        template <typename ParserT>
        parser_context(ParserT const& p) {}

        template <typename ParserT, typename ScannerT>
        void
        pre_parse(ParserT const& p, ScannerT const& scan) {}

        template <typename ResultT, typename ParserT, typename ScannerT>
        ResultT&
        post_parse(ResultT& hit, ParserT const& p, ScannerT const& scan)
        { return hit; }
    };
```

The non-terminal's `ContextT` template parameter is a concept. The `parser_context` class above is the simplest model of this concept. The default `parser_context`'s `pre_parse` and `post_parse` member functions are simply no-ops. You can think of the non-terminal's `ContextT` template parameter as the policy that governs how the non-terminal will behave before and after parsing. The client can supply her own context policy by passing a user defined context template parameter to a particular non-terminal.

Parser Context Policies

attr_t	typedef: the attribute type of the non-terminal. See the **match**.
base_t	typedef: the base class of the non-terminal. The non-terminal inherits from this class.
context_linker_t	typedef: this class type opens up the possibility for Spirit to plug in additional functionality into the non-terminal parse function or even bypass the given context. This should simply be typedefed to `parser_context_linker<T>` where T is the type of the user defined context class.
constructor	Construct the context. The non-terminal is passed as an argument to the constructor.
pre_parse	Do something prior to parsing. The non-terminal and the current scanner are passed as arguments.
post_parse	Do something after parsing. This is called regardless of the parse result. A reference to the parser's result is passed in. The context has the power to modify this. The non-terminal and the current scanner are also passed as arguments.

The `base_t` deserves further explanation. Here goes... The context is strictly a stack based class. It is created before parsing and destructed after the non-terminal's parse member function exits. Sometimes, we need auxiliary data that exists throughout the full lifetime of the non-terminal host. Since the non-terminal inherits from the context's `base_t`, the context itself, when created, gets access to this upon construction when the non-terminal is passed as an argument to the constructor. Ditto on `pre_parse` and `post_parse`.

The non-terminal inherits from the context's `base_t` typedef. The sole requirement is that it is a class that is default constructible. The copy-construction and assignment requirements depends on the host. If the host requires it, so does the context's `base_t`. In general, it wouldn't hurt to provide these basic requirements.

Non-default Attribute Type

Right out of the box, the `parser_context` class may be paramaterized with a type other than the default `nil_t`. The following code demonstrates the usage of the `parser_context` template with an explicit argument to declare rules with match results different from `nil_t`:

```
    rule<parser_context<int> > int_rule = int_p;

    parse(
        "123",
        // Using a returned value in the semantic action
        int_rule[cout << arg1 << endl]
    );
```

In this example, `int_rule` is declared with `int` attribute type. Hence, the `int_rule` variable can hold any parser which returns an `int` value (for example `int_p` or `bin_p`). The important thing to note is that we can use the returned value in the semantic action bound to the `int_rule`.

See **parser_context.cpp** in the examples. This is part of the Spirit distribution.

An Example

As an example let's have a look at the Spirit parser context, which inserts some debug output to the parsing process:

```
    template<typename ContextT>
    struct parser_context_linker : public ContextT
    {
        typedef ContextT base_t;

        template <typename ParserT>
        parser_context_linker(ParserT const& p)
        : ContextT(p) {}

    // This is called just before parsing of this non-terminal
        template <typename ParserT, typename ScannerT>
        void pre_parse(ParserT const& p, ScannerT &scan)
        {
        // call the pre_parse function of the base class
            this->base_t::pre_parse(p, scan);

#if BOOST_SPIRIT_DEBUG_FLAGS & BOOST_SPIRIT_DEBUG_FLAGS_NODES
            if (trace_parser(p.derived())) {
            // print out pre parse info
                impl::print_node_info(
                false, scan.get_level(), false,
                parser_name(p.derived()),
                scan.first, scan.last);
            }
            scan.get_level()++;  // increase nesting level
#endif
        }
    // This is called just after parsing of the current non-terminal
        template <typename ResultT, typename ParserT, typename ScannerT>
        ResultT& post_parse(
            ResultT& hit, ParserT const& p, ScannerT& scan)
        {

#if BOOST_SPIRIT_DEBUG_FLAGS & BOOST_SPIRIT_DEBUG_FLAGS_NODES
            --scan.get_level();  // decrease nesting level
            if (trace_parser(p.derived())) {
                impl::print_node_info(
                    hit, scan.get_level(), true,
                    parser_name(p.derived()),
                    scan.first, scan.last);
            }
#endif
        // call the post_parse function of the base class
            return this->base_t::post_parse(hit, p, scan);
        }
    };
```

During debugging (`BOOST_SPIRIT_DEBUG` is defined) this parser context is injected into the derivation hierarchy of the current `parser_context`, which was originally specified to be used for a concrete parser, so the template parameter `ContextT` represents the original `parser_context`. For this reason the `pre_parse` and `post_parse` functions call it's counterparts from the base class. Additionally these functions call a special `print_node_info` function, which does the actual output of the parser state info of the current non-terminal. For more info about the printed information, you may want to have a look at the topic **Debugging**.

Predefined Actors

Actors

The framework has a number of predefined semantic action functors. Experience shows that these functors are so often used that they were included as part of the core framework to spare the user from having to reinvent the same functionality over and over again.

Quick example: `assign_a` actor

```
    int i, j;
    std::string s;
    r = int_p[assign_a(i)] >> (+alpha_p)[assign_a(s)] >> int_p[assign_a(j,i)];
```

Given an input `123456 Hello 789`,

1. `assign_a(i)` will extract the number `123456` and assign it to `i`,
2. `assign_a(s)` will extract the string `"Hello"` and assign it to `s`,
3. `assign_a(j,i)` will assign i to j, j=i, without using the parse result.

Technically, the expression `assign_a(v)` is a template function that generates a semantic action. In fact, actor instances are not created directly since they usually involve a number of template parameters. Instead generator functions ("helper functions") are provided to generate actors from their arguments. All helper functions have the "_a" suffix. For example, `append_actor` is created using the `append_a` function.

The semantic action generated is polymorphic and should work with any type as long as it is compatible with the arguments received from the parser. It might not be obvious, but a string can accept the iterator first and last arguments that are passed into a generic semantic action (see above). In fact, any STL container that has an `assign(first, last)` member function can be used.

Actors summary

Below are tables summarizing the "built-in" actors with the conventions given below.

- `ref` is a **reference** to an object stored in a policy holder actor
- `value_ref` and `key_ref` are **const reference**s stored in a policy holder actor
- `value` is the **parse result**. This could be the result for the single argument () operator or the two argument () operator
- `vt` stands for the `value_type` type: `type& ref; // vt is type::value_type`.

Note that examples are provided after the tables.

Unary operator actors

++ref	**increment_a**(ref)
--ref	**decrement_a**(ref)

Assign actors

ref = value	**assign_a**(ref)
ref = value_ref	**assign_a**(ref, value_ref)

Container actors

ref.push_back(value)	**push_back_a**(ref)
ref.push_back(value_ref)	**push_back_a**(ref, value_ref)
ref.push_front(value)	**push_front_a**(ref)
ref.push_front(value_ref)	**push_front_a**(ref, value_ref)
ref.clear()	**clear_a**(ref)

Associative container actors

ref.insert(vt(value, value_ref))	**insert_key_a**(ref, value_ref)
ref.insert(vt(key_ref,value_ref))	**insert_at_a**(ref, key_ref_, value_ref)
ref.insert(vt(key_ref,value))	**insert_at_a**(ref, key_ref)
ref[value] = value_ref	**assign_key_a**(ref, value_ref)
ref.erase(ref,value)	**erase_a**(ref)
ref.erase(ref,key_ref)	**erase_a**(ref, key_ref)

Miscellanous actors

swaps aref and bref	**swap_a**(aref, bref)

Include Files

The header files for the predefined actors are located in `boost/spirit/actor`. The file `actors.hpp` contains all the includes for all the actors. You may include just the specific header files that you need. The list below enumerates the header files.

```
    #include <boost/spirit/actor/assign_actor.hpp>
    #include <boost/spirit/actor/assign_key.hpp>
    #include <boost/spirit/actor/clear_actor.hpp>
    #include <boost/spirit/actor/decrement_actor.hpp>
    #include <boost/spirit/actor/erase_actor.hpp>
    #include <boost/spirit/actor/increment_actor.hpp>
    #include <boost/spirit/actor/insert_key_actor.hpp>
    #include <boost/spirit/actor/insert_at_actor.hpp>
    #include <boost/spirit/actor/push_back_actor.hpp>
    #include <boost/spirit/actor/push_front_actor.hpp>
    #include <boost/spirit/actor/swap_actor.hpp>
```

Examples

Increment a value

Suppose that your input string is

```
    1,2,-3,4,...
```

and we want to count the number of ints. The actor `increment_a` applies `++` to its reference:

```
    int count = 0;
    rule<> r = list_p.direct(int_p[increment_a(count)], ch_p(','));
```

Append values to a vector (or other container)

Here, you want to fill a `vector<int>` with the numbers. The actor `push_back_a` can be used to insert the integers at the back of the vector:

```
    vector<int> v;
    rule<> r = list_p.direct(int_p[push_back_a(v)], ch_p(','));
```

insert key-value pairs into a map

Suppose that your input string is

```
    (1,2) (3,4) ...
```

and you want to parse the pair into a `map<int,int>`. `assign_a` can be used to store key and values in a temporary key variable, while `insert_a` is used to insert it into the map:

```
    map<int, int>::value_type k;
    map<int, int> m;

    rule<> pair =
        confix_p(
            '('
          , int_p[assign_a(k.first)] >> ',' >> int_p[assign_a(k.second)]
          , ')'
        )
        [insert_at_a(m, k)]
        ;
```

Policy holder actors and policy actions

The action takes place through a call to the `()` operator: single argument `()` operator call for character parsers and two argument (first, last) call for phrase parsers. Actors should implement at least one of the two `()` operator.

A lot of actors need to store reference to one or more objects. For example, actions on container need to store a reference to the container.

Therefore, this kind of actor have been broken down into **a)** an action policy that does the action (act member function), **b)** policy holder actor that stores the references and feeds the act member function.

Policy holder actors

The available policy holders are enumerated below.

Policy holders

Name	Stored variables	Act signature
ref_actor	1 reference	`act(ref)`
ref_value_actor	1 ref	`act(ref, value)` or `act(ref, first, last)`

ref_const_ref_actor	1 ref and 1 const ref	`act(ref, const_ref)`
ref_const_ref_value_actor	1 ref	`act(ref, value)` or `act(ref, first, last)`
ref_const_ref_const_ref_actor	1 ref, 2 const ref	`act(ref, const_ref1, const_ref2)`

Include Files

The predefined policy header files are located in `boost/spirit/actor`:

```
    #include <boost/spirit/actor/ref_actor.hpp>
    #include <boost/spirit/actor/ref_value_actor.hpp>
    #include <boost/spirit/actor/ref_const_ref.hpp>
    #include <boost/spirit/actor/ref_const_ref_value.hpp>
    #include <boost/spirit/actor/ref_const_ref_value.hpp>
    #include <boost/spirit/actor/ref_const_ref_const_ref.hpp>
```

Holder naming convention

Policy holder have the following naming convention:

```
    <member>_ >> *<member> >> !value >> actor
```

where `member` is the action policy member which can be of type:

- ref, a reference
- const_ref, a const reference
- value, by value
- empty, no stored members

and `value` states if the policy uses the parse result or not.

Holder example: `ref_actor` class

```
    // this is the building block for action that
    // take a reference and the parse result

    template<
        typename T, // reference type
        typename ActionT // action policy
    >
    class ref_value_actor : public ActionT
    {
    public:

        explicit ref_value_actor(T& ref_)
        : ref(ref_){}

        template<typename T2>
        void operator()(T2 const& val) const
        {
            act(ref, val); // defined in ActionT
        }

        template<typename IteratorT>
        void operator()(
            IteratorT const& first,
            IteratorT const& last) const
        {
            act(ref,first,last); // defined in ActionT
        }

    private:

       T& ref;
    };
```

Actor example: `assign_actor`

```
    // assign_action assigns the parse result to the reference

    struct assign_action
    {
        template<
            typename T,
            typename ValueT
        >
        void act(T& ref, ValueT const& value) const
        {
            ref = value;
        }

        template<
            typename T,
            typename IteratorT
        >
        void act(
            T& ref,
            IteratorT const& first,
            IteratorT const& last) const
        {
            typedef typename T::value_type value_type;
            value_type vt(first, last);
            ref = vt;
        }
    };
```

Helper function example: `assign_a` function

```
    // assign_a is a polymorphic helper function that generators an
    // assign_actor based on ref_value_actor, assign_action and the
    // type of its argument.

    template<typename T>
    inline ref_value_actor<T, assign_action>
    assign_a(T& ref)
    {
        return ref_value_actor<T, assign_action>(ref);
    }
```

Parametric Parsers

We already have a hint of the dynamic nature of the Spirit framework. This capability is fundamental to Spirit. Dynamic parsing is a very powerful concept. We shall take this concept further through run-time parametric parsers. We are able to handle parsing tasks that are impossible to do with any EBNF syntax alone.

A Little Secret

A little critter called `boost::ref` lurking in the boost distribution is quite powerful beast when used with Spirit's primitive parsers. We are used to seeing the Spirit primitive parsers created with string or character literals such as:

```
    ch_p('A')
    range_p('A', 'Z')
    str_p("Hello World")
```

str_p has a second form that accepts two iterators over the string:

```
    char const* first = "My oh my";
    char const* last = first + std::strlen(first);

    str_p(first, last)
```

What is not obvious is that we can use `boost::ref` as well:

```
    char ch = 'A';
    char from = 'A';
    char to = 'Z';

    ch_p(boost::ref(ch))
    range_p(boost::ref(from), boost::ref(to))
```

When `boost::ref` is used, the actual parameters to `ch_p` and `range_p` are held by reference. This means that we can change the values of `ch`, `from` and `to` anytime and the corresponding `ch_p` and `range_p` parser will follow their dynamic values. Of course, since they are held by reference, you must make sure that the referenced object is not destructed while parsing.

What about `str_p`?

While the first form of `str_p` (the single argument form) is reserved for null terminated string constants, the second form (the two argument first/last iterator form) may be used:

```
    char const* first = "My oh my";
    char const* last = first + std::strlen(first);

    str_p(boost::ref(first), boost::ref(last))
```

Hey, don't forget chseq_p. All these apply to this seldom used primitive as well.

Functional Parametric Primitives

```
    #include <boost/spirit/attribute/parametric.hpp>
```

Taking this further, Spirit includes functional versions of the primitives. Rather than taking in characters, strings or references to characters and strings (using boost::ref), the functional versions take in functions or functors.

f_chlit and f_ch_p

The functional version of chlit. This parser takes in a function or functor (function object). The function is expected to have an interface compatible with:

```
    CharT func()
```

where CharT is the character type (e.g. char, int, wchar_t).

The functor is expected to have an interface compatible with:

```
    struct functor
    {
        CharT operator()() const;
    };
```

where CharT is the character type (e.g. char, int, wchar_t).

Here's a contrived example:

```
    struct X
    {
        char operator()() const
        {
            return 'X';
        }
    };
```

Now we can use X to create our f_chlit parser:

```
    f_ch_p(X())
```

f_range and f_range_p

The functional version of range. This parser takes in a function or functor compatible with the interfaces above. The difference is that f_range (and f_range_p) expects two functors. One for the start and one for the end of the range.

f_chseq and f_chseq_p

The functional version of `chseq`. This parser takes in two functions or functors. One for the begin iterator and one for the end iterator. The function is expected to have an interface compatible with:

```
    IteratorT func()
```

where `IteratorT` is the iterator type (e.g. `char const*`, `wchar_t const*`).

The functor is expected to have an interface compatible with:

```
    struct functor
    {
        IteratorT operator()() const;
    };
```

where `IteratorT` is the iterator type (e.g. `char const*`, `wchar_t const*`).

f_strlit and f_str_p

The functional version of `strlit`. This parser takes in two functions or functors compatible with the interfaces that `f_chseq` expects.

Functional

If you look more closely, you'll notice that Spirit is all about composition of *parser functions*. A parser is just a function that accepts a scanner and returns a match. Parser *functions* are composed to form increasingly complex *higher order forms*. Notice too that the parser, albeit an object, is immutable and constant. All primitive and composite parser objects are `const`. The parse member function is even declared as `const`:

```
    template <typename ScannerT>
    typename parser_result<self_t, ScannerT>::type
    parse(ScannerT const& scan) const;
```

In all accounts, this looks and feels a lot like **Functional Programming**. And indeed it is. Spirit is by all means an application of Functional programming in the imperative C++ domain. In Haskell, for example, there is what are called **parser combinators** which are strikingly similar to the approach taken by Spirit- parser functions which are composed using various operators to create higher order parser functions that model a top-down recursive descent parser. Those smart Haskell folks have been doing this way before Spirit.

Functional style programming (or FP) libraries are gaining momentum in the C++ community. Certainly, we'll see more of FP in Spirit now and in the future. Actually, if one looks more closely, even the C++ standard library has an FP flavor. Stealthily beneath the core of the standard C++ library, a closer look into STL gives us a glimpse of a truly FP paradigm already in place. It is obvious that the authors of STL know and practice FP.

Semantic Actions in the FP Perspective

STL style FP

A more obvious application of STL-style FP in Spirit is the semantic action. What is STL-style FP? It is primarily the use of functors that can be composed to form higher order functors.

Functors

A Function Object, or Functor is simply any object that can be called as if it is a function. An ordinary function is a function object, and so is a function pointer; more generally, so is an object of a class that defines operator().

This STL-style FP can be seen everywhere these days. The following example is taken from **SGI's Standard Template Library Programmer's Guide**:

```
    //  Computes sin(x)/(x + DBL_MIN) for each element of a range.
    transform(first, last, first,
              compose2(divides<double>(),
                       ptr_fun(sin),
                       bind2nd(plus<double>(), DBL_MIN)));
```

Really, this is just *currying* in FP terminology.

Currying

What is "currying", and where does it come from?

Currying has its origins in the mathematical study of functions. It was observed by Frege in 1893 that it suffices to restrict attention to functions of a single argument. For example, for any two parameter function `f(x,y)`, there is a one parameter function `f'` such that `f'(x)` is a function that can be applied to y to give `(f'(x))(y) = f (x,y)`. This corresponds to the well known fact that the sets `(AxB -> C)` and `(A -> (B -> C))` are isomorphic, where `"x"` is cartesian product and `"->"` is function space. In functional programming, function application is denoted by juxtaposition, and assumed to associate to the left, so that the equation above becomes `f' x y = f(x,y)`.

In the context of Spirit, the same FP style functor composition may be applied to semantic actions. **full_calc.cpp** is a good example. Here's a snippet from that sample:

```
    expression =
        term
        >> *(   ('+' >> term)[make_op(plus<long>(), self.eval)]
            |   ('-' >> term)[make_op(minus<long>(), self.eval)]
            )
            ;
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

Boost style FP

Boost takes the FP paradigm further. There are libraries in boost that focus specifically on Function objects and higher-order programming.

Boost FP libraries

bind and **mem_fn**	Generalized binders for function/object/pointers and member functions, from Peter Dimov
compose	Functional composition adapters for the STL, from Nicolai Josuttis
function	Function object wrappers for deferred calls or callbacks, from Doug Gregor
functional	Enhanced function object adaptors, from Mark Rodgers

lambda	Define small unnamed function objects at the actual call site, and more, from Jaakko Järvi and Gary Powell
ref	A utility library for passing references to generic functions, from Jaako Järvi, Peter Dimov, Doug Gregor, and Dave Abrahams

The following is an example that uses boost **Bind** to use a member function as a Spirit semantic action. You can see this example in full in the file **bind.cpp**.

```
    class list_parser
    {
    public:

        typedef list_parser self_t;

        bool
        parse(char const* str)
        {
            return spirit::parse(str,

                //  Begin grammar
                (
                    real_p
                    [
                        bind(&self_t::add, this, _1)
                    ]

                    >> *(   ','
                        >>  real_p
                            [
                                bind(&self_t::add, this, _1)
                            ]
                        )
                )
                ,
                //  End grammar

                space_p).full;
        }

        void
        add(double n)
        {
            v.push_back(n);
        }

        vector<double> v;
    };
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

This parser parses a comma separated list of real numbers and stores them in a vector<double>. Boost.bind creates a Spirit conforming semantic action from the `list_parser`'s member function `add`.

Lambda and Phoenix

There's a library, authored by yours truly, named **Phoenix**. While this is not officially part of the Spirit distribution, this library has been used extensively to experiment on advanced FP techniques in C++. This library is highly influenced by

FC++ and boost Lambda (**BLL**).

BLL

In as much as Phoenix is influenced by boost Lambda (**BLL**), Phoenix innovations such as local variables, local functions and adaptable closures, in turn influenced BLL. Currently, BLL is very similar to Phoenix. Most importantly, BLL incorporated Phoenix's adaptable closures. In the future, Spirit will fully support BLL.

Phoenix allows one to write semantic actions inline in C++ through lambda (an unnamed function) expressions. Here's a snippet from the **phoenix_calc.cpp** example:

```
    expression
        =   term[expression.val = arg1]
            >> *(   ('+' >> term[expression.val += arg1])
                |   ('-' >> term[expression.val -= arg1])
                )
        ;

    term
        =   factor[term.val = arg1]
            >> *(   ('*' >> factor[term.val *= arg1])
                |   ('/' >> factor[term.val /= arg1])
                )
        ;

    factor
        =   ureal_p[factor.val = arg1]
        |   '(' >> expression[factor.val = arg1] >> ')'
        |   ('-' >> factor[factor.val = -arg1])
        |   ('+' >> factor[factor.val = arg1])
        ;
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

You do not have to worry about the details for now. There is a lot going on here that needs to be explained. The succeeding chapters will be enlightening.

Notice the use of lambda expressions such as:

```
    expression.val += arg1
```

Lambda Expressions?

Lambda expressions are actually unnamed partially applied functions where placeholders (e.g. arg1, arg2) are provided in place of some of the arguments. The reason this is called a lambda expression is that traditionally, such placeholders are written using the Greek letter lambda λ.

where `expression.val` is a closure variable of the expression rule (see **Closures**).

`arg1` is a placeholder for the first argument that the semantic action will receive (see **Phoenix Place-holders**). In Boost.Lambda (BLL), this corresponds to `_1`.

Phoenix

The preceding chapter introduced Phoenix as a means to implementing your semantic actions. We shall look a little bit more into this important library with focus on how you can use it handily with Spirit. This chapter is by no means a thorough discourse of the library. For more information on Phoenix, please take some time to read the **Phoenix User's Guide**. If you just want to use it quickly, this chapter will probably suffice. Rather than taking you to the theories and details of the library, we shall try to provide you with annotated exemplars instead. Hopefully, this will get you into high gear quickly.

Semantic actions in Spirit can be just about any function or function object (functor) as long as it can satisfy the required signature. For example, `uint_p` requires a signature of `void F(T)`, where `T` is the type of the integer (typically `unsigned int`). Plain vanilla actions are of the `void F(IterT, IterT)` variety. You can code your actions in plain C++. Calls to C++ functions or functors will thus be of the form `P[&F]` or `P[F()]` etc. (see **Semantic Actions**). Phoenix on the other hand, attempts to mimic C++ such that you can define the function body inlined in the code.

C++ in C++?

In as much as Spirit attempts to mimic EBNF in C++, Phoenix attempts to mimic C++ in C++!!!

var

Remember the `boost::ref`? We discussed that in the **Parametric Parsers chapter**. Phoenix has a similar, but more flexible, counterpart. It's called `var`. The usage is similar to `boost::ref` and you can use it as a direct replacement. However, unlike `boost::ref`, you can use it to form more complex expressions. Here are some examples:

```
    var(x) += 3
    var(x) = var(y) + var(z)
    var(x) = var(y) + (3 * var(z))
    var(x) = var(y)[var(i)] // assuming y is indexable and i is an index
```

Let's start with a simple example. We'll want to parse a comma separated list of numbers and report the sum of all the numbers. Using phoenix's var, we do not have to write external semantic actions. We simply inline the code inside the semantic action slots. Here's the complete grammar with our phoenix actions (see **sum.cpp** in the examples):

```
    real_p[var(n) = arg1] >> *(',' >> real_p[var(n) += arg1])
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

argN

Notice the expression: `var(n) = arg1` . What is `arg1` and what is it doing there? `arg1` is an argument placeholder. Remember that `real_p` (see **Numerics**) reports the parsed number to its attached semantic action. `arg1` is a placeholder for the first argument passed to the semantic action by the parser. If there are more than one arguments passed in, these arguments can be referred to using `arg1`..`argN`. For instance, generic semantic actions (transduction interface; see **Semantic Actions**) are passed 2 arguments: the iterators (`first`/`last`) to the matching portion of the input stream. You can refer to `first` and `last` through `arg1` and `arg2`, respectively.

Like var, argN is also composable. Here are some examples:

```
    var(x) += arg1
    var(x) = arg1 + var(z)
    var(x) = arg1 + (3 * arg2)
    var(x) = arg1[arg2] // assuming arg1 is indexable and arg2 is an index
```

val

Note the expression: `3 * arg2`. This expression is actually a short-hand equivalent to: `val(3) * arg2`. We shall see later why, in some cases, we need to explicitly wrap constants and literals inside the val. Again, like var and argN, val is also composable.

Functions

Remember our very first example? In the **Quick Start** chapter, we presented a parser that parses a comma separated list and stuffs the parsed numbers in a vector (see **number_list.cpp**) . For simplicity, we used Spirit's pre-defined actors (see **Predefined Actors**). In the example, we used `push_back_a`:

```
    real_p[push_back_a(v)] >> *(',' >> real_p[push_back_a(v)])
```

Phoenix allows you to write more powerful polymorphic functions, similar to `push_back_a`, easily. See **stuff_vector.cpp**. The example is similar to **number_list.cpp** in functionality, but this time, using phoenix a function to actually implement the `push_back` function:

```
    struct push_back_impl
    {
        template <typename Container, typename Item>
        struct result
        {
            typedef void type;
        };

        template <typename Container, typename Item>
        void operator()(Container& c, Item const& item) const
        {
            c.push_back(item);
        }
    };

    function<push_back_impl> const push_back = push_back_impl();
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

Predefined Phoenix Functions

A future version of Phoenix will include an extensive set of predefined functions covering the whole of STL containers, iterators and algorithms. push_back, will be part of this suite.

`push_back_impl` is a simple wrapper over the `push_back` member function of STL containers. The extra scaffolding is there to provide phoenix with additional information that otherwise cannot be directly deduced. `result` relays to phoenix the return type of the functor (`operator()`) given its argument types (`Container` and `Item`) . In this case, the return type is always, simply `void`.

`push_back` is a phoenix function object. This is the actual function object that we shall use. The beauty behind phoenix function objects is that the actual use is strikingly similar to a normal C++ function call. Here's the number list parser rewritten using our phoenix function object:

```
    real_p[push_back(var(v), arg1)] >> *(',' >> real_p[push_back(var(v), arg1)])
```

And, unlike predefined actors, they can be composed. See the pattern? Here are some examples:

```
    push_back(var(v), arg1 + 2)
    push_back(var(v), var(x) + arg1)
    push_back(var(v)[arg1], arg2) // assuming v is a vector of vectors and arg1 is an index
```

push_back does not have a return type. Say, for example, we wrote another phoenix function `sin`, we can use it in expressions as well:

```
    push_back(var(v), sin(arg1) * 2)
```

Construct

Sometimes, we wish to construct an object. For instance, we might want to create a `std::string` given the first/last iterators. For instance, say we want to parse a list of identifiers instead. Our grammar, without the actions, is:

```
    (+alpha_p) >> *(',' >> (+alpha_p))
```

`construct_` is a predefined phoenix function that, you guessed it, constructs an object, from the arguments passed in. The usage is:

```
    construct_<T>(arg1, arg2,... argN)
```

where T is the desired type and arg1..argN are the constructor arguments. For example, we can construct a `std::string` from the first/last iterator pair this way:

```
    construct_<std::string>(arg1, arg2)
```

Now, we attach the actions to our grammar:

```
    (+alpha_p)
    [
        push_back(var(v), construct_<std::string>(arg1, arg2))
    ]
    >>
    *(',' >>
        (+alpha_p)
        [
            push_back(var(v), construct_<std::string>(arg1, arg2))
        ]
    )
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

Lambda expressions

All these phoenix expressions we see above are lambda expressions. The important thing to note is that these expressions are not evaluated immediately. At grammar construction time, when the actions are attached to the productions, a lambda expression actually generates an unnamed function object that is evaluated later, at parse time. In other words, lambda expressions are **lazily evaluated**.

Lambda Expressions?

Lambda expressions are actually unnamed partially applied functions where placeholders (e.g. arg1, arg2) are provided in place of some of the arguments. The reason this is called a lambda expression is that traditionally, such placeholders are written using the Greek letter lambda λ.

Phoenix uses tricks not unlike those used by Spirit to mimic C++ such that you can define the function body inlined in the code. It's weird, but as mentioned, Phoenix actually mimicks C++ in C++ using expression templates. Surely, there are limitations...

All components in a Phoenix expression must be an **actor** (in phoenix parlance) in the same way that components in Spirit should be a `parser`. In Spirit, you can write:

```
    r = ch_p('x') >> 'y';
```

But not:

```
    r = 'x' >> 'y';
```

In essence, `parser >> char` is a parser, but `char >> char` is a char (the char shift-right by another char).

The same restrictions apply to Phoenix. For instance:

```
    int x = 1;
    cout << var(x) << "pizza"
```

is a well formed Phoenix expression that's lazily evaluated. But:

```
    cout << x << "pizza"
```

is not. Such expressions are immediately executed. C++ syntax dictates that at least **one** of the operands must be a Phoenix actor type. This also applies to compound expressions. For example:

```
    cout << var(x) << "pizza" << "man"
```

This is evaluated as:

```
    (((cout << var(x)) << "pizza") << "man")
```

Since `(cout << var(x))` is an actor, at least **one** of the operands is a phoenix actor, `((cout << var(x)) << "pizza")` is also a Phoenix actor, and the whole expression is thus also an actor.

Sometimes, it is safe to write:

```
    cout << var(x) << val("pizza") << val("man")
```

just to make it explicitly clear what we are dealing with, especially with complex expressions, in the same way as we explicitly wrap literal strings in `str_p("lit")` in Spirit.

Phoenix (and Spirit) also deals with unary operators. In such cases, we have no choice. The operand must be a Phoenix actor (or Spirit parser). Examples:

Spirit:

```
    *ch_p('z')  // good
    *('z') // bad
```

Phoenix:

```
    *var(x) // good (lazy)
    *x // bad (immediate)
```

Also, in Phoenix, for assignments and indexing to be lazily evaluated, the object acted upon should be a Phoenix actor. Examples:

```
    var(x) = 123 // good (lazy)
    x = 123 // bad (immediate)
    var(x)[0] // good (lazy)
    x[0] // bad, immediate
    var(x)[var(i)] // good (lazy)
    x[var(i)] // bad and illegal (x is not an actor)
    var(x[var(i)]) // bad and illegal (x is not an actor)
```

Wrapping up

Well, there you have it. I hope with this jump-start chapter, you may be able to harness the power of lambda expressions. By all means, please read the **phoenix manual** to learn more about the nitty gritty details. Surely, you'll get to know a lot more than just by reading this chapter. There are a lot of things still to be touched. There won't be enough space here to cover all the features of Phoenix even in brief.

The next chapter, **Closures**, we'll see more of phoenix. Stay tuned.

Closures

Overview

Using phoenix, in the previous chapter, we've seen how we can get data from our parsers using `var`:

```
    int i;
    integer = int_p[var(i) = arg1];
```

Nifty! Our rule `integer`, if successful, passes the parsed integer to the variable `i`. Everytime we need to parse an integer, we can call our rule `integer` and simply extract the parsed number from the variable `i`. There's something you should be aware of though. In the viewpoint of the grammar, the variable `i` is global. When the grammar gets more complex, it's hard to keep track of the current state of `i`. And, with recursive rules, global variables simply won't be adequate.

Closures are needed if you need your rules (or grammars) to be reentrant. For example, a rule (or grammar) might be called recursively indirectly or directly by itself. The calculator is a good example. The expression rule recursively calls itself indirectly when it invokes the factor rule.

Closures provide named (lazy) variables associated with each parse rule invocation. A closure variable is addressed using member syntax:

```
    rulename.varname
```

A closure variable `R.x` may be addressed in the semantic action of any other rule invoked by `R`; it refers to the innermost enclosing invocation of `R`. If no such invocation exists, an assertion occurs at runtime.

Closures provide an environment, a stack frame, for local variables. Most importantly, the closure variables are accessible from the EBNF grammar specification and can be used to pass parser information upstream or downstream from the topmost rule down to the terminals in a top-down recursive descent. Closures facilitate dynamic scoping in C++. Spirit's closure implementation is based on *Todd Veldhuizen*'s **Dynamic scoping in C++** technique that he presented in his paper **Techniques for Scientic C++**.

When a rule is given a closure, the closure's local variables are created prior to entering the parse function and destructed after exiting the parse function. These local variables are true local variables that exist on the hardware stack.

Closures and Phoenix

Spirit v1.8 closure support requires **Phoenix**. In the future, Spirit will fully support **BLL**. Currently, work is underway to merge the features of both libraries.

Example

Let's go back to the calculator grammar introduced in the **Functional** chapter. Here's the full grammar again, plus the closure declarations:

```
    struct calc_closure : boost::spirit::closure<calc_closure, double>
    {
        member1 val;
    };

    struct calculator : public grammar<calculator, calc_closure::context_t>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(calculator const& self)
            {
                top = expression[self.val = arg1];

                expression
                    =   term[expression.val = arg1]
                        >> *(   ('+' >> term[expression.val += arg1])
                            |   ('-' >> term[expression.val -= arg1])
                            )
                    ;

                term
                    =   factor[term.val = arg1]
                        >> *(   ('*' >> factor[term.val *= arg1])
                            |   ('/' >> factor[term.val /= arg1])
                            )
                    ;

                factor
                    =   ureal_p[factor.val = arg1]
                    |   '(' >> expression[factor.val = arg1] >> ')'
                    |   ('-' >> factor[factor.val = -arg1])
                    |   ('+' >> factor[factor.val = arg1])
                    ;
            }

            typedef rule<ScannerT, calc_closure::context_t> rule_t;
            rule_t expression, term, factor;
            rule<ScannerT> top;

            rule<ScannerT> const&
            start() const { return top; }
        };
    };
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

Surely, we've come a long way from the original version of this calculator. With inline **lambda expressions**, we were able to write self contained grammars complete with semantic actions.

The first thing to notice is the declaration of `calc_closure`.

Declaring closures

The general closure declaration syntax is:

```
    struct name : spirit::closure<name, type1, type2, type3,... typeN>
    {
        member1 m_name1;
        member2 m_name2;
        member3 m_name3;
        ...
        memberN m_nameN;
    };
```

member1... memberN are indirect links to the actual closure variables. Their indirect types correspond to type1... typeN. In our example, we declared calc_closure:

```
    struct calc_closure : boost::spirit::closure<calc_closure, double>
    {
        member1 val;
    };
```

calc_closure has a single variable val of type double.

BOOST_SPIRIT_CLOSURE_LIMIT

Spirit predefined maximum closure limit. This limit defines the maximum number of elements a closure can hold. This number defaults to 3. The actual maximum is rounded up in multiples of 3. Thus, if this value is 4, the actual limit is 6. The ultimate maximum limit in this implementation is 15. It should **NOT** be greater than PHOENIX_LIMIT (see **phoenix**). Example:

```
// Define these before including anything else
#define PHOENIX_LIMIT 10
#define BOOST_SPIRIT_CLOSURE_LIMIT 10
```

Attaching closures

Closures can be applied to rules, subrules and grammars (non-terminals). The closure has a special **parser context** that can be used with these non-terminals. The closure's context is its means to hook into the non-terminal. The context of the closure C is C::context_t.

We can see in the example that we attached calc_closure to the expression, term and factor rules in our grammar:

```
    typedef rule<ScannerT, calc_closure::context_t> rule_t;
    rule_t expression, term, factor;
```

as well as the grammar itself:

```
    struct calculator : public grammar<calculator, calc_closure::context_t>
```

Closure return value

The closure member1 is the closure's return value. This return value, like the one returned by anychar_p, for example, can be used to propagate data up the parser hierarchy or passed to semantic actions. Thus, expression, term and factor, as well as the calculator grammar itself, all return a double.

Accessing closure variables

Closure variables can be accessed from within semantic actions just like you would struct members: by qualifying the member name with its owner rule, subrule or grammar. In our example above, notice how we referred to the closure member val. Example:

```
    expression.val // refer to expression's closure member val
```

Initializing closure variables

We didn't use this feature in the example, yet, for completeness...

Sometimes, we need to initialize our closure variables upon entering a non-terminal (rule, subrule or grammar). Closure enabled non-terminals, by default, default-construct variables upon entering the parse member function. If this is not desirable, we can pass constructor arguments to the non-terminal. The syntax mimics a function call.

For (*a contrived*) example, if you wish to construct `calc_closure`'s variables to `3.6`, when we invoke the rule `expression`, we write:

```
    expression(3.6) // invoke rule expression and set its closure variable to 3
```

The constructor arguments are actually Phoenix lambda expressions, so you can use arbitrarily complex expressions. Here's another *contrived example:*

```
    // call rule factor and set its closure variable to (expression.x / 8) * fa
    factor((expression.x / 8) * term.y)
```

We can pass less arguments than the actual number of variables in the closure. The variables at the right with no corresponding constructor arguments are default constructed. Passing more arguments than there are closure variables is an error.

See **parameters.cpp** for a compilable example. This is part of the Spirit distribution.

Closures and Dynamic parsing

Let's write a very simple parser for an XML/HTML like language with arbitrarily nested tags. The typical approach to this type of nested tag parsing is to delegate the actual tag matching to semantic actions, perhaps using a symbol table. For example, the semantic actions are responsible for ensuring that the tags are nested (e.g. this code: `<p><table></p></table>` is erroneous).

Spirit allows us to dynamically modify the parser at runtime. The ability to guide parser behavior through semantic actions makes it possible to ensure the nesting of tags directly in the parser. We shall see how this is possible. here's the grammar in its simplest form:

```
    element = start_tag >> *element >> end_tag;
```

An element is a `start_tag` (e.g. `<font>`) folowed by zero or more elements, and ended by an `end_tag` (e.g. `</font>`). Now, here's a first shot at our `start_tag`:

```
    start_tag = '<' >> lexeme_d[(+alpha_p)] >> '>';
```

Notice that the `end_tag` is just the same as `start_tag` with the addition of a slash:

```
    end_tag = "</" >> what_we_got_in_the_start_tag >> '>';
```

What we need to do is to temporarily store what we got in our `start_tag` and use that later to parse our `end_tag`. Nifty, we can use the **parametric parser** primitives to parse our `end_tag`:

```
    end_tag = "</" >> f_str_p(tag) >> '>';
```

where we parameterize `f_str_p` with what we stored (tag).

Be reminded though that our grammar is recursive. The element rule calls itself. Hence, we can't just use a variable and use `phoenix::var` or `boost::ref`. Nested recursion will simply gobble up the variable. Each invocation of element must have a closure variable `tag`. Here now is the complete grammar:

```
    struct tags_closure : boost::spirit::closure<tags_closure, string>
    {
        member1 tag;
    };

    struct tags : public grammar<tags>
    {
        template <typename ScannerT>
        struct definition {

            definition(tags const& /*self*/)
            {
                element = start_tag >> *element >> end_tag;

                start_tag =
                        '<'
                    >>  lexeme_d
                        [
                            (+alpha_p)
                            [
                                //  construct string from arg1 and arg2 lazily
                                //  and assign to element.tag

                                element.tag = construct_<string>(arg1, arg2)
                            ]
                        ]
                    >> '>';

                end_tag = "</" >> f_str_p(element.tag) >> '>';
            }

            rule<ScannerT, tags_closure::context_t> element;
            rule<ScannerT> start_tag, end_tag;

            rule<ScannerT, tags_closure::context_t> const&
            start() const { return element; }
        };
    };
```

We attached a semantic action to the `(+alpha_p)` part of the start_tag. There, we stored the parsed tag in the `element`'s closure variable `tag`. Later, in the `end_tag`, we simply used the `element`'s closure variable `tag` to parameterize our `f_str_p` parser. Simple and elegant. If some of the details begin to look like greek (e.g. what is `construct_`?), please consult the **Phoenix** chapter.

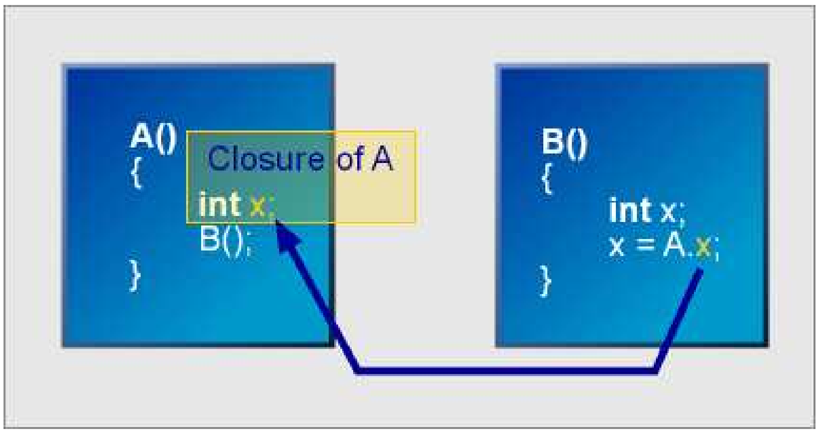
The full source code can be **viewed here**. This is part of the Spirit distribution.

Closures in-depth

What are Closures?

The closure is an object that ***"closes"*** over the local variables of a function making them visible and accessible outside the function. What is more interesting is that the closure actually packages a local context (stack frame where some variables reside) and makes it available outside the scope in which they actually exist. The information is essentially ***"captured"*** by the closure allowing it to be referred to anywhere and anytime, even prior to the actual creation of the variables.

The following diagram depicts the situation where a function A (or rule) exposes its closure and another function B references A's variables through its closure.



The closure as an object that *"closes"* over the local variables of a function making them visible and accessible outside the function

Of course, function A should be active when A.x is referenced. What this means is that function B is reliant on function A (If B is a nested function of A, this will always be the case). The free form nature of Spirit rules allows access to a closure variable anytime, anywhere. Accessing A.x is equivalent to referring to the topmost stack variable x of function A. If function A is not active when A.x is referenced, a runtime exception will be thrown.

Nested Functions

To fully understand the importance of closures, it is best to look at a language such as Pascal which allows nested functions. Since we are dealing with C++, lets us assume for the moment that C++ allows nested functions. Consider the following ***pseudo*** C++ code:

```
    void a()
    {
        int va;
        void b()
        {
            int vb;
            void c()
            {
                int vc;
            }

            c();
        }

        b();
    }
```

We have three functions a, b and c where c is nested in b and b is nested in a. We also have three variables va, vb and vc. The lifetime of each of these local variables starts when the function where it is declared is entered and ends when the function exits. The scope of a local variable spans all nested functions inside the enclosing function where the variable is declared.

Going downstream from function a to function c, when function a is entered, the variable va will be created in the stack. When function b is entered (called by a), va is very well in scope and is visble in b. At which point a fresh variable, vb, is created on the stack. When function c is entered, both va and vb are visibly in scope, and a fresh local variable vc is created.

Going upstream, vc is not and cannot be visible outside the function c. vc's life has already expired once c exits. The same is true with vb; vb is accessible in function c but not in function a.

Nested Mutually Recursive Rules

Now consider that a, b and c are rules:

```
    a = b >> *(('+' >> b) | ('-' >> b));
    b = c >> *(('*' >> c) | ('/' >> c));
    c = int_p | '(' >> a >> ')' | ('-' >> c) | ('+' >> c);
```

We can visualize a, b and c as mutually recursive functions where a calls b, b calls c and c recursively calls a. Now, imagine if a, b and c each has a local variable named value that can be referred to in our grammar by explicit qualification:

```
    a.value // refer to a's value local variable
    b.value // refer to b's value local variable
    c.value // refer to c's value local variable
```

Like above, when a is entered, a local variable value is created on the stack. This variable can be referred to by both b and c. Again, when b is called by a, b creates a local variable value. This variable is accessible by c but not by a.

Here now is where the analogy with nested functions end: when c is called, a fresh variable value is created which, as usual, lasts the whole lifetime of c. Pay close attention however that c may call a recursively. When this happens, a may now refer to the local variable of c.

Dynamic Parsers

We see dynamic parsing everywhere in Spirit. A special group of parsers, aptly named dynamic parsers, form the most basic building blocks to dynamic parsing. This chapter focuses on these critters. You'll notice the similarity of these parsers with C++'s control structures. The similarity is not a coincidence. These parsers give an imperative flavor to parsing, and, since imperative constructs are not native to declarative EBNF, mimicking the host language, C++, should make their use immediately familiar.

Dynamic parsers modify the parsing behavior according to conditions. Constructing dynamic parsers requires a condition argument and a body parser argument. Additional arguments are required by some parsers.

Conditions

Functions or functors returning values convertable to bool can be used as conditions. When the evaluation of the function/functor yields true it will be considered as meeting the condition.

Parsers can be used as conditions, as well. When the parser matches the condition is met. Parsers used as conditions work in an all-or-nothing manner: the scanner will not be advanced when they don't match.

A failure to meet the condition will not result in a parse error.

if_p

`if_p` can be used with or without an else-part. The syntax is:

```
    if_p(condition)[then-parser]
```

or

```
    if_p(condition)[then-parser].else_p[else-parser]
```

When the condition is met the then-parser is used next in the parsing process. When the condition is not met and an else-parser is available the else-parser is used next. When the condition isn't met and no else-parser is available then the whole parser matches the empty sequence. (⚠ Note: older versions of `if_p` report a failure when the condition isn't met and no else-parser is available.)

Example:

```
    if_p("0x")[hex_p].else_p[uint_p]
```

while_p, do_p

while_p/do_p syntax is:

```
    while_p(condition)[body-parser]
    do_p[body-parser].while_p(condition)
```

As long as the condition is met the dynamic parser constructed by while_p will try to match the body-parser. do_p returns a parser that tries to match the body-parser and then behaves just like the parser returned by while_p. A failure to match the body-parser will cause a failure to be reported by the while/do-parser.

Example:

```
    uint_p[assign_a(sum)] >> while_p('+')[uint_p(add(sum)]
    '"' >> while_p(~eps_p('"'))[c_escape_ch_p[push_back_a(result)]] >> '"'
```

for_p

for_p requires four arguments. The syntax is:

```
    for_p(init, condition, step)[body-parser]
```

init and step have to be 0-ary functions/functors. for_p returns a parser that will:

1. call init
2. check the condition, if the condition isn't met then a match is returned. The match will cover everything that has been matched successfully up to this point.
3. tries to match the body-parser. A failure to match the body-parser will cause a failure to be reported by the for-parser
4. calls step
5. goes to 2.

Storable Rules

The rule is a weird C++ citizen, unlike any other C++ object. It does not have the proper copy and assignment semantics and cannot be stored and passed around by value. You cannot store rules in STL containers (vector, stack, etc) for later use and you cannot pass and return rules to and from functions by value.

EBNF is primarily declarative. Like in functional programming, an EBNF grammar is a static recipe and there's no notion of do this then that. However, in Spirit, we managed to coax imperative C++ to take in declarative EBNF. Hah! Fun!... We did that by masquerading the C++ assignment operator to mimic EBNF's `::=`. To do that, we gave the rule class' assignment operator and copy constructor a different meaning and semantics. The downside is that doing so made the rule unlike any other C++ object. You can't copy it. You can't assign it.

We want to have the dynamic nature of C++ to our advantage. We've seen dynamic Spirit in action here and there. There are indeed some interesting applications of dynamic parsers using Spirit. Yet, we will not fully utilize the power of dynamic parsing, unless we have a rule that behaves like any other good C++ object. With such a beast, we can write full parsers that's defined at run time, as opposed to compile time.

We now have dynamic rules: `stored_rules`. Basically they are rules with perfect C++ assignment/copy-constructor semantics. This means that `stored_rules` can be stored in containers and/or dynamically created at run-time.

```
    template<
        typename ScannerT = scanner<>,
        typename ContextT = parser_context<>,
        typename TagT = parser_address_tag>
    class stored_rule;
```

The interface is exactly the same as with the rule class (see the **section on rules** for more information regarding the API). The only difference is with the copy and assignment semantics. Now, with `stored_rules`, we can dynamically and algorithmically define our rules. Here are some samples...

Say I want to dynamically create a rule for:

```
    start = *(a | b | c);
```

I can write it dynamically step-by-step:

```
    stored_rule<>  start;

    start = a;
    start = start.copy() | b;
    start = start.copy() | c;
    start = *(start.copy());
```

Later, I changed my mind and want to redefine it (again dynamically) as:

```
    start = (a | b) >> (start | b);
```

I write:

```
    start = b;
    start = a | start.copy();
    start = start.copy() >> (start | b);
```

Notice the statement:

```
    start = start.copy() | b;
```

Why is start.copy() required? Well, because like rules, stored rules are still embedded by reference when found in the RHS (one reason is to avoid cyclic-shared-pointers). If we write:

```
    start = start | b;
```

We have **left-recursion**! Copying copy of start avoids self referencing. What we are doing is making a copy of start, ORing it with b, then destructively assigning the result back to start.

The Lazy Parser

Closures are cool. It allows us to inject stack based local variables anywhere in our parse descent hierarchy. Typically, we store temporary variables, generated by our semantic actions, in our closure variables, as a means to pass information up and down the recursive descent.

Now imagine this... Having in mind that closure variables can be just about any type, we can store a parser, a rule, or a pointer to a parser or rule, in a closure variable. *Yeah, right, so what?...* Ok, hold on... What if we can use this closure variable to initiate a parse? Think about it for a second. Suddenly we'll have some powerful dynamic parsers! Suddenly we'll have a full round trip from to **Phoenix** and Spirit and back! **Phoenix** semantic actions choose the right Spirit parser and Spirit parsers choose the right **Phoenix** semantic action. Oh MAN, what a honky cool idea, I might say!!

lazy_p

This is the idea behind the `lazy_p` parser. The `lazy_p` syntax is:

```
    lazy_p(actor)
```

where actor is a **Phoenix** expression that returns a Spirit parser. This returned parser is used in the parsing process.

Example:

```
    lazy_p(phoenix::val(int_p))[assign_a(result)]
```

Semantic actions attached to the `lazy_p` parser expects the same signature as that of the returned parser (`int_p`, in our example above).

lazy_p example

To give you a better glimpse (see the **`lazy_parser.cpp`**), say you want to parse inputs such as:

```
    dec
    {
        1 2 3
        bin
        {
            1 10 11
        }
        4 5 6
    }
```

where `bin {...}` and `dec {...}` specifies the numeric format (binary or decimal) that we are expecting to read. If we analyze the input, we want a grammar like:

```
    base = "bin" | "dec";
    block = base >> '{' >> *block_line >> '}';
    block_line = number | block;
```

We intentionally left out the `number` rule. The tricky part is that the way `number` rule behaves depends on the result of the `base` rule. If `base` got a *"bin"*, then number should parse binary numbers. If `base` got a *"dec"*, then number should parse decimal numbers. Typically we'll have to rewrite our grammar to accomodate the different parsing behavior:

```
    block =
            "bin" >> '{' >> *bin_line >> '}'
        |   "dec" >> '{' >> *dec_line >> '}'
        ;
    bin_line = bin_p | block;
    dec_line = int_p | block;
```

while this is fine, the redundancy makes us want to find a better solution; after all, we'd want to make full use of Spirit's dynamic parsing capabilities. Apart from that, there will be cases where the set of parsing behaviors for our `number` rule is not known when the grammar is written. We'll only be given a map of string descriptors and corresponding rules [e.g. (("dec", int_p), ("bin", bin_p) ... etc...)].

The basic idea is to have a rule for binary and decimal numbers. That's easy enough to do (see **numerics**). When `base` is being parsed, in your semantic action, store a pointer to the selected base in a closure variable (e.g. `block.int_rule`). Here's an example:

```
    base
        = str_p("bin")[block.int_rule = &var(bin_rule)]
        | str_p("dec")[block.int_rule = &var(dec_rule)]
        ;
```

With this setup, your number rule will now look something like:

```
    number = lazy_p(*block.int_rule);
```

The **`lazy_parser.cpp`** does it a bit differently, ingeniously using the **symbol table** to dispatch the correct rule, but in essence, both strategies are similar. This technique, using the symbol table, is detailed in the Techiques section: **nabialek_trick**. Admitedly, when you add up all the rules, the resulting grammar is more complex than the hard-coded grammar above. Yet, for more complex grammar patterns with a lot more rules to choose from, the additional setup is well worth it.

The Select Parser

Select parsers may be used to identify a single parser from a given list of parsers, which successfully recognizes the current input sequence. Example:

```
    rule<> rule_select =
        select_p
        (
            parser_a
          , parser_b
            /* ... */
          , parser_n
        );
```

The parsers (parser_a, parser_b etc.) are tried sequentially from left to right until a parser matches the current input sequence. If there is a matching parser found, the `select_p` parser returns the parser's position (zero based index). For instance, in the example above, `1` is returned if parser_b matches.

There are two predefined parsers of the select parser family: `select_p` and `select_fail_p`. These parsers differ in the way the no match case is handled (when none of the parsers match the current input sequence). While the `select_p` parser will return `-1` if no matching parser is found, the `select_fail_p` parser will not match at all.

The following sample shows how the select parser may be used very conveniently in conjunction with a **switch parser**:

```
    int choice = -1;
    rule<> rule_select =
            select_fail_p('a', 'b', 'c', 'd')[assign_a(choice)]
        >>  switch_p(var(choice))
            [
                case_p<0>(int_p),
                case_p<1>(ch_p(',')),
                case_p<2>(str_p("bcd")),
                default_p
            ]
        ;
```

This example shows a rule, which matches:

- *'a'* followed by an integer
- *'b'* followed by a *','*
- *'c'* followed by *"bcd"*
- a single *'d'*.

For other input sequences the give rule does not match at all.

⚠ BOOST_SPIRIT_SELECT_LIMIT

The number of possible entries inside the select_p parser is limited by the Spirit compile time constant BOOST_SPIRIT_SELECT_LIMIT, which defaults to 3. This value should not be greater than the compile time constant given by PHOENIX_LIMIT (see **phoenix**). Example:

```
// Define these before including anything else
#define PHOENIX_LIMIT 10
#define BOOST_SPIRIT_SELECT_LIMIT 10
```

The Switch Parser

Switch parsers may be used to simplify certain alternation constructs. Consider the following code:

```
    rule<> rule_overall =
            ch_p('a') >> parser_a
        |   ch_p('b') >> parser_b
            // ...
        |   ch_p('n') >> parser_n
        ;
```

Each of the alternatives are evaluated normally in a sequential manner. This tend to be inefficient, especially for a large number of alternatives. To avoid this inefficiency and to make it possible to write such constructs in a more readable form, Spirit contains the `switch_p` family of parsers. The switch_p parser allows us to rewrite the previous construct as:

```
    rule<> rule_overall =
            switch_p
            [
                case_p<'a'>(parser_a),
                case_p<'b'>(parser_b),
                // ...
                case_p<'n'>(parser_n)
            ]
        ;
```

This `switch_p` parser takes the next character (or token) from the input stream and tries to match it against the given integral compile time constants supplied as the template parameters to the `case_p` parsers. If this character matches one of the `case_p` branches, the associated parser is executed (i.e. if 'a' is matched, `parser_a` is executed, if 'b' is matched, `parser_b` is executed and so on) . If no `case_p` branch matches the next input character, the overall construct does not match at all.

Nabialek trick

The ***"Nabialek trick"*** (from the name of its inventor, Sam Nabialek), can also improve the rule dispatch from linear non-deterministic to deterministic. This is similar to the `switch_p` parser, yet, can handle grammars where a keyword (operator, etc), instead of a single character or token, precedes a production.

Sometimes it is desireable to add handling of the default case (none of the `case_p` branches matched). This may be achieved with the help of a `default_p` branch:

```
    rule<> rule_overall =
            switch_p
            [
                case_p<'a'>(parser_a),
                case_p<'b'>(parser_b),
                // ...
                case_p<'n'>(parser_n),
                default_p(parser_default)
            ]
        ;
```

This form chooses the `parser_default` parser if none of the cases matches the next character from the input stream. Please note that, obviously, only one `default_p` branch may be added to the `switch_p` parser construct.

Moreover, it is possible to omit the parentheses and body from the `default_p` construct, in which case, no additional parser is executed and the overall `switch_p` construct simply returns a match on any character of the input stream, which does not match any of the `case_p` branches:

```
    rule<> rule_overall =
            switch_p
            [
                case_p<'a'>(parser_a),
                case_p<'b'>(parser_b),
                // ...
                case_p<'n'>(parser_n),
                default_p
            ]
        ;
```

There is another form of the switch_p construct. This form allows us to explicitly specify the value to be used for matching against the `case_p` branches:

```
    rule<> rule_overall =
            switch_p(cond)
            [
                case_p<'a'>(parser_a),
                case_p<'b'>(parser_b),
                // ...
                case_p<'n'>(parser_n)
            ]
        ;
```

where `cond` is a parser or a nullary function or function object (functor). If it is a parser, then it is tried and its return value is used to match against the `case_p` branches. If it is a nullary function or functor, then its return value will be used.

Please note that during its compilation, the `switch_p` construct is transformed into a real C++ `switch` statement. This makes the runtime execution very efficient.

⚠ `BOOST_SPIRIT_SWITCH_CASE_LIMIT`

The number of possible `case_p`/`default_p` branches is limited by the Spirit compile time constant `BOOST_SPIRIT_SWITCH_CASE_LIMIT`, which defaults to 3. There is no theoretical upper limit for this constant, but most compilers won't allow you to specify a very large number.

Example:

```
// Define these before including switch.hpp
#define BOOST_SPIRIT_SWITCH_CASE_LIMIT 10
```

Escape Character Parser

The Escape Character Parser is a utility parser, which parses escaped character sequences used in C/C++, LEX or Perl regular expressions. Combined with the confix_p utility parser, it is useful for parsing C/C++ strings containing double quotes and other escaped characters:

```
    confix_p('"', *c_escape_ch_p, '"')
```

There are two different types of the Escape Character Parser: `c_escape_ch_p`, which parses C/C++ escaped character sequences and `lex_escape_ch_p`, which parses LEX style escaped character sequences. The following table shows the valid character sequences understood by these utility parsers.

Summary of valid escaped character sequences

c_escape_ch_p	`\b, \t, \n, \f, \r, \\, \", \', \xHH, \OOO` where: H is some hexadecimal digit (0..9, a..f, A..F) and O is some octal digit (0..7)
lex_escape_ch_p	all C/C++ escaped character sequences as described above and additionally any other character, which follows a backslash

If there is a semantic action attached directly to the Escape Character Parser, all valid escaped characters are converted to their character equivalent (i.e. a backslash followed by a 'r' is converted to '\r'), which is fed to the attached actor. The number of hexadecimal or octal digits parsed depends on the size of one input character. An overflow will be detected and will generate a non-match. lex_escape_ch_p will strip the leading backslash for all character sequences which are not listed as valid C/C++ escape sequences when passing the unescaped character to an attached action.

Please note though, that if there is a semantic action attached to an outermost parser (for instance as in `(*c_escape_ch_p)[some_actor]`, where the action is attached to the kleene star generated parser) no conversion takes place at the moment, but nevertheless the escaped characters are parsed correctly. This limitation will be removed in a future version of the library.

Loops

So far we have introduced a couple of EBNF operators that deal with looping. We have the + positive operator, which matches the preceding symbol one (1) or more times, as well as the Kleene star * which matches the preceding symbol zero (0) or more times.

Taking this further, we may want to have a generalized loop operator. To some this may seem to be a case of overkill. Yet there are grammars that are impractical and cumbersome, if not impossible, for the basic EBNF iteration syntax to specify. Examples:

- A file name may have a maximum of 255 characters only.
- A specific bitmap file format has exactly 4096 RGB color information.
- A 32 bit binary string (1..32 1s or 0s).

Other than the Kleene star *, the Positive closure +, and the optional !, a more flexible mechanism for looping is provided for by the framework.

Loop Constructs

repeat_p (n) [p]	Repeat **p** exactly **n** times
repeat_p (n1, n2) [p]	Repeat **p** at least **n1** times and at most **n2** times
repeat_p (n, more) [p]	Repeat **p** at least **n** times, continuing until **p** fails or the input is consumed

Using the `repeat_p` parser, we can now write our examples above:

A file name with a maximum of 255 characters:

```
    valid_fname_chars = /*..*/;
    filename = repeat_p(1, 255)[valid_fname_chars];
```

A specific bitmap file format which has exactly 4096 RGB color information:

```
    uint_parser<unsigned, 16, 6, 6> rgb_p;
    bitmap = repeat_p(4096)[rgb_p];
```

As for the 32 bit binary string (1..32 1s or 0s), of course we could have easily used the `bin_p` numeric parser instead. For the sake of demonstration however:

```
    bin32 = lexeme_d[repeat_p(1, 32)[ch_p('1') | '0']];
```

Loop parsers are run-time **parametric**.

The Loop parsers can be dynamic. Consider the parsing of a binary file of Pascal-style length prefixed string, where the first byte determines the length of the incoming string. Here's a sample input:

```
11  h  e  l  l  o  _  w  o  r  l  d
```

This trivial example cannot be practically defined in traditional EBNF. Although some EBNF syntax allow more powerful repetition constructs other than the Kleene star, we are still limited to parsing fixed strings. The nature of EBNF forces the repetition factor to be a constant. On the other hand, Spirit allows the repetition factor to be variable at run time. We could write a grammar that accepts the input string above:

```
    int c;
    r = anychar_p[assign_a(c)] >> repeat_p(boost::ref(c))[anychar_p];
```

The expression

```
    anychar_p[assign_a(c)]
```

extracts the first character from the input and puts it in `c`. What is interesting is that in addition to constants, we can also use variables as parameters to `repeat_p`, as demonstrated in

```
    repeat_p(boost::ref(c))[anychar_p]
```

Notice that `boost::ref` is used to reference the integer `c`. This usage of `repeat_p` makes the parser defer the evaluation of the repetition factor until it is actually needed. Continuing our example, since the value 11 is already extracted from the input, `repeat_p` is is now expected to loop exactly 11 times.

Character Sets

The character set `chset` matches a set of characters over a finite range bounded by the limits of its template parameter `CharT`. This class is an optimization of a parser that acts on a set of single characters. The template class is parameterized by the character type `CharT` and can work efficiently with 8, 16 and 32 and even 64 bit characters.

```
    template <typename CharT = char>
    class chset;
```

The `chset` is constructed from literals (e.g. `'x'`), `ch_p` or `chlit<>`, `range_p` or `range<>`, `anychar_p` and `nothing_p` (see **primitives**) or copy-constructed from another `chset`. The `chset` class uses a copy-on-write scheme that enables instances to be passed along easily by value.

Sparse bit vectors

To accomodate 16/32 and 64 bit characters, the `chset` class statically switches from a `std::bitset` implementation when the character type is not greater than 8 bits, to a sparse bit/boolean set which uses a sorted vector of disjoint ranges (`range_run`). The set is constructed from ranges such that adjacent or overlapping ranges are coalesced.

range_runs are very space-economical in situations where there are lots of ranges and a few individual disjoint values. Searching is O(log n) where n is the number of ranges.

Examples:

```
    chset<> s1('x');
    chset<> s2(anychar_p - s1);
```

Optionally, character sets may also be constructed using a definition string following a syntax that resembles posix style regular expression character sets, except that double quotes delimit the set elements instead of square brackets and there is no special negation `^` character.

```
    range = anychar_p >> '-' >> anychar_p;
    set = *(range_p | anychar_p);
```

Since we are defining the set using a C string, the usual C/C++ literal string syntax rules apply. Examples:

```
    chset<> s1("a-zA-Z");        // alphabetic characters
    chset<> s2("0-9a-fA-F");     // hexadecimal characters
    chset<> s3("actgACTG");      // DNA identifiers
    chset<> s4("\x7f\x7e");      // Hexadecimal 0x7F and 0x7E
```

The standard Spirit set operators apply (see **operators**) plus an additional character-set-specific inverse (negation ~) operator:

Character set operators

~a	Set inverse
a | b	Set union
a &	Set intersection
a - b	Set difference
a ^ b	Set xor

where operands a and b are both `chsets` or one of the operand is either a literal character, `ch_p` or `chlit`, `range_p` or `range`, `anychar_p` or `nothing_p`. Special optimized overloads are provided for `anychar_p` and `nothing_p` operands. A `nothing_p` operand is converted to an empty set, while an `anychar_p` operand is converted to a set having elements of the full range of the character type used (e.g. 0-255 for unsigned 8 bit chars).

A special case is `~anychar_p` which yields `nothing_p`, but `~nothing_p` is illegal. Inversion of `anychar_p` is asymmetrical, a one-way trip comparable to converting `T*` to a `void*.`

Special conversions

chset<CharT>(nothing_p)	empty set
chset<CharT>(anychar_p)	full range of CharT (e.g. 0-255 for unsigned 8 bit chars)
~anychar_p	nothing_p
~nothing_p	illegal

Confix Parsers

Confix Parsers

Confix Parsers recognize a sequence out of three independent elements: an opening, an expression and a closing. A simple example is a C comment:

```
    /* This is a C comment */
```

which could be parsed through the following rule definition:

```
    rule<> c_comment_rule
        =   confix_p("/*", *anychar_p, "*/")
        ;
```

The `confix_p` parser generator should be used for generating the required Confix Parser. The three parameters to `confix_p` can be single characters (as above), strings or, if more complex parsing logic is required, auxiliary parsers, each of which is automatically converted to the corresponding parser type needed for successful parsing.

The generated parser is equivalent to the following rule:

```
    open >> (expr - close) >> close
```

If the expr parser is an `action_parser_category` type parser (a parser with an attached semantic action) we have to do something special. This happens, if the user wrote something like:

```
    confix_p(open, expr[func], close)
```

where `expr` is the parser matching the expr of the confix sequence and `func` is a functor to be called after matching the `expr`. If we would do nothing, the resulting code would parse the sequence as follows:

```
    open >> (expr[func] - close) >> close
```

which in most cases is not what the user expects. (If this is what you've expected, then please use the `confix_p` generator function `direct()`, which will inhibit the parser refactoring). To make the confix parser behave as expected:

```
    open >> (expr - close)[func] >> close
```

the actor attached to the `expr` parser has to be re-attached to the `(expr - close)` parser construct, which will make the resulting confix parser 'do the right thing'. This refactoring is done by the help of the **Refactoring Parsers**. Additionally special care must be taken, if the expr parser is a `unary_parser_category` type parser as

```
    confix_p(open, *anychar_p, close)
```

which without any refactoring would result in

```
    open >> (*anychar_p - close) >> close
```

and will not give the expected result (*anychar_p will eat up all the input up to the end of the input stream). So we have to refactor this into:

```
    open >> *(anychar_p - close) >> close
```

what will give the correct result.

The case, where the expr parser is a combination of the two mentioned problems (i.e. the expr parser is a unary parser with an attached action), is handled accordingly too, so:

```
    confix_p(open, (*anychar_p)[func], close)
```

will be parsed as expected:

```
    open >> (*(anychar_p - end))[func] >> close
```

The required refactoring is implemented here with the help of the **Refactoring Parsers** too.

Summary of Confix Parser refactorings

You write it as:	It is refactored to:
`confix_p(open, expr, close)`	`open >> (expr - close) >> close`
`confix_p(open, expr[func], close)`	`open >> (expr - close)[func] >> close`
`confix_p(open, *expr, close)`	`open >> *(expr - close) >> close`
`confix_p(open, (*expr)[func], close)`	`open >> (*(expr - close))[func] >> close`

Comment Parsers

The Comment Parser generator template `comment_p` is helper for generating a correct **Confix Parser** from auxiliary parameters, which is able to parse comment constructs as follows:

```
    StartCommentToken >> Comment text >> EndCommentToken
```

There are the following types supported as parameters: parsers, single characters and strings (see as_parser). If it is used with one parameter, a comment starting with the given first parser parameter up to the end of the line is matched. So for instance the following parser matches C++ style comments:

```
    comment_p("//")
```

If it is used with two parameters, a comment starting with the first parser parameter up to the second parser parameter is matched. For instance a C style comment parser could be constrcuted as:

```
    comment_p("/*", "*/")
```

The `comment_p` parser generator allows to generate parsers for matching non-nested comments (as for C/C++ comments). Sometimes it is necessary to parse nested comments as for instance allowed in Pascal.

```
    { This is a { nested } PASCAL-comment }
```

Such nested comments are parseable through parsers generated by the `comment_nest_p` generator template functor. The following example shows a parser, which can be used for parsing the two different (nestable) Pascal comment styles:

```
    rule<> pascal_comment
        =   comment_nest_p("(*", "*)")
        |   comment_nest_p('{', '}')
        ;
```

Please note, that a comment is parsed implicitly as if the whole `comment_p(...)` statement were embedded into a `lexeme_d[]` directive, i.e. during parsing of a comment no token skipping will occur, even if you've defined a skip parser for your whole parsing process.

comments.cpp demonstrates various comment parsing schemes:

1. Parsing of different comment styles
   - parsing C/C++-style comment
   - parsing C++-style comment
   - parsing PASCAL-style comment
2. Parsing tagged data with the help of the confix_parser
3. Parsing tagged data with the help of the confix_parser but the semantic action is directly attached to the body sequence parser

This is part of the Spirit distribution.

List Parsers

List Parsers are generated by the special predefined parser generator object list_p, which generates parsers recognizing list structures of the type

```
    item >> *(delimiter >> item) >> !end
```

where item is an expression, delimiter is a delimiter and end is an optional closing expression. As you can see, the list_p generated parser does not recognize empty lists, i.e. the parser must find at least one item in the input stream to return a successful match. If you wish to also match an empty list, you can make your list_p optional with operator! An example where this utility parser is helpful is parsing comma separated C/C++ strings, which can be easily formulated as:

```
    rule<> list_of_c_strings_rule
        =   list_p(confix_p('\"', *c_escape_char_p, '\"'), ',')
        ;
```

The confix_p and c_escape_char_p parser generators are described **here** and **here**.

The list_p parser generator object can be used to generate the following different types of List Parsers:

List Parsers

list_p	list_p used by itself parses comma separated lists without special item formatting, i.e. everything in between two commas is matched as an item, no end of list token is matched
list_p(delimiter)	generates a list parser, which recognizes lists with the given delimiter and matches everything in between them as an item, no end of list token is matched
list_p(item, delimiter)	generates a list parser, which recognizes lists with the given delimiter and matches items based on the given item parser, no end of list token is matched
list_p(item, delimiter, end)	generates a list parser, which recognizes lists with the given delimiter and matches items based on the given item parser and additionally recognizes an optional end expression

All of the parameters to list_p can be single characters, strings or, if more complex parsing logic is required, auxiliary parsers, each of which is automatically converted to the corresponding parser type needed for successful parsing.

If the item parser is an action_parser_category type (parser with an attached semantic action) we have to do something special. This happens, if the user wrote something like:

```
    list_p(item[func], delim)
```

where `item` is the parser matching one item of the list sequence and `func` is a functor to be called after matching one item. If we would do nothing, the resulting code would parse the sequence as follows:

```
    (item[func] - delim) >> *(delim >> (item[func] - delim))
```

what in most cases is not what the user expects. (If this is what you've expected, then please use one of the `list_p` generator functions `direct()`, which will inhibit refactoring of the `item` parser). To make the list parser behave as expected:

```
    (item - delim)[func] >> *(delim >> (item - delim)[func])
```

the actor attached to the item parser has to be re-attached to the `(item - delim)` parser construct, which will make the resulting list parser 'do the right thing'. This refactoring is done by the help of the **Refactoring Parsers**. Additionally special care must be taken, if the item parser is a `unary_parser_category` type parser as for instance:

```
    list_p(*anychar_p, ',')
```

which without any refactoring would result in

```
        (*anychar_p - ch_p(','))
    >> *( ch_p(',') >> (*anychar_p - ch_p(',')) )
```

and will not give the expected result (the first `*anychar_p` will eat up all the input up to the end of the input stream). So we have to refactor this into:

```
       *(anychar_p - ch_p(','))
    >> *( ch_p(',') >> *(anychar_p - ch_p(',')) )
```

what will give the correct result.

The case, where the item parser is a combination of the two mentioned problems (i.e. the item parser is a unary parser with an attached action), is handled accordingly too:

```
    list_p((*anychar_p)[func], ',')
```

will be parsed as expected:

```
        (*(anychar_p - ch_p(',')))[func]
    >> *( ch_p(',') >> (*(anychar_p - ch_p(',')))[func] )
```

The required refactoring is implemented with the help of the **Refactoring Parsers**.

Summary of List Parser refactorings

You write it as:	It is refactored to:
`list_p(item, delimiter)`	`(item - delimiter) >> *(delimiter >> (item - delimiter))`

`list_p(item[func], delimiter)`	`(item - delimiter)[func] >> *(delimiter >> (item - delimiter)[func])`
`list_p(*item, delimiter)`	`*(item - delimiter) >> *(delimiter >> *(item - delimiter))`
`list_p((*item)[func], delimiter)`	`(*(item - delimiter))[func] >> *(delimiter >> (*(item - delimiter))[func])`

list_parser.cpp sample shows the usage of the list_p utility parser:

1. parsing a simple ',' delimited list w/o item formatting
2. parsing a CSV list (comma separated values - strings, integers or reals)
3. parsing a token list (token separated values - strings, integers or reals) with an action parser directly attached to the item part of the list_p generated parser

This is part of the Spirit distribution.

Functor Parser

The simplest way to write your hand coded parser that works well with the rest of the Spirit library is to simply write a functor parser.

A functor parser is expected to have the interface:

```
    struct functor
    {
        typedef T result_t;

        template <typename ScannerT>
        std::ptrdiff_t
        operator()(ScannerT const& scan, result_t& result) const;
    };
```

where typedef T result_t; is the attribute type of the parser that will be passed back to the match result (see **In-depth: The Parser**). If the parser does not need to return an attribute, this can simply be nil_t. The `std::ptrdiff_t` result is the number of matching characters matched by your parser. A negative value flags an unsucessful match.

A conforming functor parser can transformed into a well formed Spirit parser by wrapping it in the functor_parser template:

```
    functor_parser<functor> functor_p;
```

Example

The following example puts the functor_parser into action:

```
    struct number_parser
    {
        typedef int result_t;
        template <typename ScannerT>
        std::ptrdiff_t
        operator()(ScannerT const& scan, result_t& result) const
        {
            if (scan.at_end())
                return -1;

            char ch = *scan;
            if (ch < '0' || ch > '9')
                return -1;

            result = 0;
            std::ptrdiff_t len = 0;

            do
            {
                result = result*10 + int(ch - '0');
                ++len;
                ++scan;
            } while (!scan.at_end() && (ch = *scan, ch >= '0' && ch <= '9'));

            return len;
        }
    };
    functor_parser<number_parser> number_parser_p;
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

To further understand the implementation, see **In-depth: The Scanner** for the scanner API details. We now have a parser `number_parser_p` that we can use just like any other Spirit parser. Example:

```
    r = number_parser_p >> *(',' >> number_parser_p);
```

Refactoring Parsers

There are three types of Refactoring Parsers implemented right now, which help to abstract common parser refactoring tasks. Parser refactoring means, that a concrete parser construct is replaced (refactored) by another very similar parser construct. Two of the Refactoring Parsers described here (`refactor_unary_parser` and `refactor_action_parser`) are introduced to allow a simple and more expressive notation while using **Confix Parsers** and **List Parsers**. The third Refactoring Parser (`attach_action_parser`) is implemented to abstract some functionality required for the Grouping Parser. Nevertheless these Refactoring Parsers may help in solving other complex parsing tasks too.

Refactoring unary parsers

The `refactor_unary_d` parser generator, which should be used to generate a unary refactoring parser, transforms a construct of the following type

```
    refactor_unary_d[*some_parser - another_parser]
```

to

```
    *(some_parser - another_parser)
```

where `refactor_unary_d` is a predefined object of the parser generator struct `refactor_unary_gen<>`

The `refactor_unary_d` parser generator generates a new parser as shown above, only if the original construct is an auxilliary binary parser (here the difference parser) and the left operand of this binary parser is an auxilliary unary parser (here the kleene star operator). If the original parser isn't a binary parser the compilation will fail. If the left operand isn't an unary parser, no refactoring will take place.

Refactoring action parsers

The `refactor_action_d` parser generator, which should be used to generate an action refactoring parser, transforms a construct of the following type

```
    refactor_action_d[some_parser[some_actor] - another_parser]
```

to

```
    (some_parser - another_parser)[some_actor]
```

where `refactor_action_d` is a predefined object of the parser generator struct `refactor_action_gen<>`

The `refactor_action_d` parser generator generates a new parser as shown above, only if the original construct is an auxilliary binary parser (here the difference parser) and the left operand of this binary parser is an auxilliary parser generated by an attached semantic action. If the original parser isn't a binary parser the compilation will fail. If the left operand isn't an action parser, no refactoring will take place.

Attach action refactoring

The `attach_action_d` parser generator, which should be used to generate an attach action refactoring parser, transforms a construct of the following type

```
    attach_action_d[(some_parser >> another_parser)[some_actor]]
```

to

```
    some_parser[some_actor] >> another_parser[some_actor]
```

> where `attach_action_d` is a predefined object of the parser generator struct `attach_action_gen<>`

The `attach_action_d` parser generator generates a new parser as shown above, only if the original construct is an auxilliary action parser and the parser to it this action is attached is an auxilliary binary parser (here the sequence parser). If the original parser isn't a action parser the compilation will fail. If the parser to which the action is attached isn't an binary parser, no refactoring will take place.

Nested refactoring

Sometimes it is required to nest different types of refactoring, i.e. to transform constructs like

```
    (*some_parser)[some_actor] - another_parser
```

to

```
    (*(some_parser - another_parser))[some_actor]
```

To simplify the construction of such nested refactoring parsers the `refactor_unary_gen<>` and `refactor_action_gen<>` both can take another refactoring parser generator type as their respective template parameter. For instance, to construct a refactoring parser generator for the mentioned nested transformation we should write:

```
    typedef refactor_action_gen<refactor_unary_gen<> > refactor_t;
    const refactor_t refactor_nested_d = refactor_t(refactor_unary_d);
```

Now we could use it as follows to get the required result:

```
    refactor_nested_d[(*some_parser)[some_actor] - another_parser]
```

An empty template parameter means not to nest this particular refactoring parser. The default template parameter is `non_nesting_refactoring`, a predefined helper structure for inhibiting nesting. Sometimes it is required to nest a particular refactoring parser with itself. This is achieved by providing the predefined helper structure `self_nested_refactoring` as the template parameter to the corresponding refactoring parser generator template.

See **refactoring.cpp** for a compilable example. This is part of the Spirit distribution.

Regular Expression Parser

Regular expressions are a form of pattern-matching that are often used in text processing. Many users will be familiar with the usage of regular expressions. Initially there were the Unix utilities grep, sed and awk, and the programming language perl, each of which make extensive use of regular expressions. Today the usage of such regular expressions is integrated in many more available systems.

During parser construction it is often useful to have the power of regular expressions available. The Regular Expression Parser was introduced, to make the use of regular expressions accessible for Spirit parser construction.

The Regular Expression Parser `rxstrlit` has a single template type parameter: an iterator type. Internally, `rxstrlit` holds the Boost Regex object containing the provided regular expression. The `rxstrlit` attempts to match the current input stream with this regular expression. The template type parameter defaults to `char const*`. `rxstrlit` has two constructors. The first accepts a null-terminated character pointer. This constructor may be used to build `rxstrlit's` from quoted regular expression literals. The second constructor takes in a first/last iterator pair. The function generator version is `regex_p`.

Here are some examples:

```
    rxstrlit<>("Hello[[:space:]]+[W|w]orld")
    regex_p("Hello[[:space:]]+[W|w]orld")

    std::string msg("Hello[[:space:]]+[W|w]orld");
    rxstrlit<>(msg.begin(), msg.end());
```

The generated parser object acts at the character level, thus an eventually given skip parser is not used during the attempt to match the regular expression (see **The Scanner Business**).

The Regular Expression Parser is implemented by the help of the **Boost Regex++ library**, so you have to have some limitations in mind.

- Boost libraries have to be installed on your computer and the Boost root directory has to be added to your compiler `#include<...>` search path. You can download the actual version at the **Boost web site**.

- The Boost Regex library requires the usage of bi-directional iterators. So you have to ensure this during the usage of the Spirit parser, which contains a Regular Expression Parser.

- The Boost Regex library is not a header only library, as Spirit is, though it provides the possibility to include all of the sources, if you are using it in one compilation unit only. Define the preprocessor constant `BOOST_SPIRIT_NO_REGEX_LIB` before including the spirit Regular

Expression Parser header, if you want to include all the Boost Regex sources into this compilation unit. If you are using the Regular Expression Parser in more than one compilation unit, you should not define this constant and must link your application against the regex library as described in the related documentation.

See **regular_expression.cpp** for a compilable example. This is part of the Spirit distribution.

Scoped Lock

scoped_lock_d

The `scoped_lock_d` directive constructs a parser that locks a mutex during the attempt to match the contained parser.

Syntax:

```
    scoped_lock_d(mutex&)[body-parser]
```

Note, that nesting `scoped_lock_d` directives bears the risk of deadlocks since the order of locking depends on the grammar used and may even depend on the input being parsed. Locking order has to be consistent within an application to ensure deadlock free operation.

Distinct Parser

Distinct Parsers

The distinct parsers are utility parsers which ensure that matched input is not immediately followed by a forbidden pattern. Their typical usage is to distinguish keywords from identifiers.

distinct_parser

The basic usage of the `distinct_parser` is to replace the `str_p` parser. For example the `declaration_rule` in the following example:

```
    rule<ScannerT> declaration_rule = str_p("declare") >> lexeme_d[+alpha_p];
```

would correctly match an input "declare abc", but as well an input"declareabc" what is usually not intended. In order to avoid this, we can use `distinct_parser`:

```
    // keyword_p may be defined in the global scope
    distinct_parser<> keyword_p("a-zA-Z0-9_");

    rule<ScannerT> declaration_rule = keyword_p("declare") >> lexeme_d[+alpha_p];
```

The `keyword_p` works in the same way as the `str_p` parser but matches only when the matched input is not immediately followed by one of the characters from the set passed to the constructor of `keyword_p`. In the example the "declare" can't be immediately followed by any alphabetic character, any number or an underscore.

See the full **example here** .

distinct_directive

For more sophisticated cases, for example when keywords are stored in a symbol table, we can use `distinct_directive`.

```
    distinct_directive<> keyword_d("a-zA-Z0-9_");

    symbol<> keywords = "declare", "begin", "end";
    rule<ScannerT> keyword = keyword_d[keywords];
```

dynamic_distinct_parser and dynamic_distinct_directive

In some cases a set of forbidden follow-up characters is not sufficient. For example ASN.1 naming conventions allows identifiers to contain dashes, but not double dashes (which marks the beginning of a comment). Furthermore, identifiers can't end with a dash. So, a matched keyword can't be followed by any alphanumeric character or exactly one dash, but can be followed by two dashes.

This is when `dynamic_distinct_parser` and the `dynamic_distinct_directive` come into play. The constructor of the `dynamic_distinct_parser` accepts a parser which matches any input that **must NOT** follow the keyword.

```
    // Alphanumeric characters and a dash followed by a non-dash
    // may not follow an ASN.1 identifier.
    dynamic_distinct_parser<> keyword_p(alnum_p | ('-' >> ~ch_p('-')));

    rule<ScannerT> declaration_rule = keyword_p("declare") >> lexeme_d[+alpha_p];
```

Since the `dynamic_distinct_parser` internally uses a rule, its type is dependent on the scanner type. So, the `keyword_p` shouldn't be defined globally, but rather within the grammar.

See the full **example here**.

How it works

When the `keyword_p_1` and the `keyword_p_2` are defined as

```
    distinct_parser<> keyword_p(forbidden_chars);
    distinct_parser_dynamic<> keyword_p(forbidden_tail_parser);
```

the parsers

```
    keyword_p_1(str)
    keyword_p_2(str)
```

are equivalent to the rules

```
    lexeme_d[chseq_p(str) >> ~epsilon_p(chset_p(forbidden_chars))]
    lexeme_d[chseq_p(str) >> ~epsilon_p(forbidden_tail_parser)]
```

Symbols

This class symbols implements a symbol table. The symbol table holds a dictionary of symbols where each symbol is a sequence of CharTs (a `char`, `wchar_t`, `int`, enumeration etc.) . The template class, parameterized by the character type (CharT), can work efficiently with 8, 16, 32 and even 64 bit characters. Mutable data of type T is associated with each symbol.

Traditionally, symbol table management is maintained seperately outside the BNF grammar through semantic actions. Contrary to standard practice, the Spirit symbol table class `symbols` is-a parser. An instance of which may be used anywhere in the EBNF grammar specification. It is an example of a dynamic parser. A dynamic parser is characterized by its ability to modify its behavior at run time. Initially, an empty symbols object matches nothing. At any time, symbols may be added, thus, dynamically altering its behavior.

Each entry in a symbol table has an associated mutable data slot. In this regard, one can view the symbol table as an associative container (or map) of key-value pairs where the keys are strings.

The symbols class expects two template parameters (actually there is a third, see detail box). The first parameter `T` specifies the data type associated with each symbol (defaults to `int`) and the second parameter `CharT` specifies the character type of the symbols (defaults to `char`).

```
    template
    <
        typename T = int,
        typename CharT = char,
        typename SetT = impl::tst<T, CharT>
    >
    class symbols;
```

Ternary State Trees

The actual set implementation is supplied by the SetT template parameter (3rd template parameter of the symbols class) . By default, this uses the tst class which is an implementation of the Ternary Search Tree.

Ternary Search Trees are faster than hashing for many typical search problems especially when the search interface is iterator based. Searching for a string of length k in a ternary search tree with n strings will require at most O(log n+k) character comparisons. TSTs are many times faster than hash tables for unsuccessful searches since mismatches are discovered earlier after examining only a few characters. Hash tables always examine an entire key when searching.

For details see **http://www.cs.princeton.edu/~rs/strings/**.

Here are some sample declarations:

```
    symbols<> sym;
    symbols<short, wchar_t> sym2;

    struct my_info
    {
        int     id;
        double  value;
    };

    symbols<my_info> sym3;
```

After having declared our symbol tables, symbols may be added statically using the construct:

```
    sym = a, b, c, d ...;
```

where `sym` is a symbol table and `a..d` etc. are strings. ✏️Note that the comma operator is separating the items being added to the symbol table, through an assignment. Due to operator overloading this is possible and correct (though it may take a little getting used to) and is a concise way to initialize the symbol table with many symbols. Also, it is perfectly valid to make multiple assignments to a symbol table to iteratively add symbols (or groups of symbols) at different times.

Simple example:

```
    sym = "pineapple", "orange", "banana", "apple", "mango";
```

Note that it is invalid to add the same symbol multiple times to a symbol table, though you may modify the value associated with a symbol artibrarily many times.

Now, we may use sym in the grammar. Example:

```
    fruits = sym >> *(',' >> sym);
```

Alternatively, symbols may be added dynamically through the member functor `add` (see **`symbol_inserter`** below). The member functor `add` may be attached to a parser as a semantic action taking in a begin/end pair:

```
    p[sym.add]
```

where p is a parser (and sym is a symbol table). On success, the matching portion of the input is added to the symbol table.

`add` may also be used to directly initialize data. Examples:

```
    sym.add("hello", 1)("crazy", 2)("world", 3);
```

Assuming of course that the data slot associated with `sym` is an integer.

The data associated with each symbol may be modified any time. The most obvious way of course is through **semantic actions**. A function or functor, as usual, may be attached to the symbol table. The symbol table expects a function or functor compatible with the signature:

Signature for functions:

```
    void func(T& data);
```

Signature for functors:

```
    struct ftor
    {
        void operator()(T& data) const;
    };
```

Where `T` is the data type of the symbol table (the `T` in its template parameter list). When the symbol table successfully matches something from the input, the data associated with the matching entry in the symbol table is reported to the semantic action.

Symbol table utilities

Sometimes, one may wish to deal with the symbol table directly. Provided are some symbol table utilities.

add

```
    template <typename T, typename CharT, typename SetT>
    T*  add(symbols<T, CharT, SetT>& table, CharT const* sym, T const& data = T()),
```

adds a symbol `sym` (C string) to a symbol table `table` plus an optional data `data` associated with the symbol. Returns a pointer to the data associated with the symbol or `NULL` if add failed (e.g. when the symbol is already added before).

find

```
    template <typename T, typename CharT, typename SetT>
    T*  find(symbols<T, CharT, SetT> const& table, CharT const* sym);
```

finds a symbol `sym` (C string) from a symbol table `table`. Returns a pointer to the data associated with the symbol or `NULL` if not found

symbol_inserter

The symbols class holds an instance of this class named `add`. This can be called directly just like a member function, passing in a first/last iterator and optional data:

```
    sym.add(first, last, data);
```

Or, passing in a C string and optional data:

```
    sym.add(c_string, data);
```

where `sym` is a symbol table. The `data` argument is optional. The nice thing about this scheme is that it can be cascaded. We've seen this applied above. Here's a snippet from the roman numerals parser

```
    //  Parse roman numerals (1..9) using the symbol table.

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

    } ones_p;
```

Notice that a user defined struct `ones` is subclassed from `symbols`. Then at construction time, we added all the symbols using the `add` symbol_inserter.

The full source code can be **viewed here**. This is part of the Spirit distribution.

Again, `add` may also be used as a semantic action since it conforms to the action interface (see semantic actions):

```
    p[sym.add]
```

where p is a parser of course.

Trees

Why use parse trees

Parse trees are an in-memory representation of the input with a structure that conforms to the grammar.

The advantages of using parse trees instead of semantic actions:

- You can make multiple passes over the data without having to re-parse the input.
- You can perform transformations on the tree.
- You can evaluate things in any order you want, whereas with attribute schemes you have to process in a begin to end fashion.
- You do not have to worry about backtracking and action side effects that may occur with an ambiguous grammar.

Example

Now that you think you may want to use trees, I'll give an example of how to use them and you can see how easy they are to use. So, following with tradition (and the rest of the documentation) we'll do a calculator. Here's the grammar:

```
    integer
        =   lexeme_d[ token_node_d[ (!ch_p('-') >> +digit_p) ] ]
        ;

    factor
        =   integer
        |   '(' >> expression >> ')'
        |   ('-' >> factor)
        ;

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

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

Now, you'll notice the only thing different in this grammar is the `token_node_d` directive. This causes the integer rule to group all the input into one node. Without `token_node_d`, each character would get it's own node. As you'll soon see, it's easier to convert the input into an int when all the characters are in one node. Here is how the parse is done to create a tree:

```
    tree_parse_info<> info = pt_parse(first, expression);
```

`pt_parse()` is similar to `parse()`. There are four overloads: two for pairs of first and last iterators and two for character strings. Two of the functions take a skipper parser and the other two do not.

The `tree_parse_info` struct contains the same information as a `parse_info` struct as well as one extra data member called trees. When the parse finishes, trees will contain the parse tree.

Here is how you can use the tree to evaluate the input:

```
    if (info.full)
    {
        cout << "parsing succeeded\n";
        cout << "result = " << evaluate(info) << "\n\n";
    }
```

Now you ask, where did `evaluate()` come from? Is it part of spirit? Unfortunately, no, `evaluate()` is only a part of the sample. Here it is:

```
    long evaluate(const tree_parse_info<>& info)
    {
        return eval_expression(info.trees.begin());
    }
```

So here you see evaluate() simply calls `eval_expression()` passing the begin() iterator of info.trees. Now here's the rest of the example:

```
    // Here's some typedefs to simplify things
    typedef char const*                              iterator_t;
    typedef tree_match<iterator_t>                   parse_tree_match_t;
    typedef parse_tree_match_t::const_tree_iterator  iter_t;

    // Here's the function prototypes that we'll use.  One function for each
    // grammar rule.
    long evaluate(const tree_parse_info<>& info);
    long eval_expression(iter_t const& i);
    long eval_term(iter_t const& i);
    long eval_factor(iter_t const& i);
    long eval_integer(iter_t const& i);

    // i should be pointing to a node created by the expression rule
    long eval_expression(iter_t const& i)
    {
        // first child points to a term, so call eval_term on it
        iter_t chi = i->children.begin();
        long lhs = eval_term(chi);
        for (++chi; chi != i->children.end(); ++chi)
        {
            // next node points to the operator.  The text of the operator is
            // stored in value (a vector<char>)
            char op = *(chi->value.begin());
            ++chi;
            long rhs = eval_term(chi);
            if (op == '+')
                lhs += rhs;
            else if (op == '-')
                lhs -= rhs;
            else
                assert(0);
        }
        return lhs;
    }

    long eval_term(iter_t const& i)
    {
        //  ... see parse_tree_calc1.cpp for complete example
        //  (it's rather similar to eval_expression() ) ...
    }

    long eval_factor(iter_t const& i)
    {
        //  ... again, see parse_tree_calc1.cpp if you want all the details ...
    }

    long eval_integer(iter_t const& i)
    {
        // use the range constructor for a string
        string integer(i->value.begin(), i->value.end());
        // convert the string to an integer
        return strtol(integer.c_str(), 0, 10);
    }
```

The full source code can be **viewed here**. This is part of the Spirit distribution.

So, you like what you see, but maybe you think that the parse tree is too hard to process? With a few more directives you can generate an abstract syntax tree (ast) and cut the amount of evaluation code by at least **50%**. So without any delay, here's the ast calculator grammar:

```
    integer
        =   leaf_node_d[ lexeme_d[ (!ch_p('-') >> +digit_p) ] ]
        ;
    factor
        =   integer
        |   inner_node_d[ch_p('(') >> expression >> ch_p(')')]
        |   (root_node_d[ch_p('-')] >> factor)
        ;

    term
        =   factor
            >> *(  (root_node_d[ch_p('*')] >> factor)
                |  (root_node_d[ch_p('/')] >> factor)
                )
        ;

    expression
        =   term
            >> *(  (root_node_d[ch_p('+')] >> term)
                | (root_node_d[ch_p('-')] >> term)
                )
        ;
```

The differences from the parse tree grammar are hi-lighted in **bold-red**. The `inner_node_d` directive causes the first and last nodes generated by the enclosed parser to be discarded, since we don't really care about the parentheses when evaluating the expression. The `root_node_d` directive is the key to ast generation. A node that is generated by the parser inside of `root_node_d` is marked as a root node. When a root node is created, it becomes a root or parent node of the other nodes generated by the same rule.

To start the parse and generate the ast, you must use the ast_parse functions, which are similar to the pt_parse functions.

```
    tree_parse_info<> info = ast_parse(first, expression);
```

Here is the eval_expression function (note that to process the ast we only need one function instead of four):

```
    long eval_expression(iter_t const& i)
    {
        if (i->value.id() == parser_id(&integer))
        {
            // convert string to integer
            string integer(i->value.begin(), i->value.end());
            return strtol(integer.c_str(), 0, 10);
        }
        else if (i->value.id() == parser_id(&factor))
        {
            // factor can only be unary minus
            return - eval_expression(i->children.begin());
        }
        else if (i->value.id() == parser_id(&term))
        {
            if (*i->value.begin() == '*')
            {
                return eval_expression(i->children.begin()) *
                    eval_expression(i->children.begin()+1);
            }
            else if (*i->value.begin() == '/')
            {
                return eval_expression(i->children.begin()) /
                    eval_expression(i->children.begin()+1);
            }
        }
        else if (i->value.id() == parser_id(&expression))
        {
            if (*i->value.begin() == '+')
            {
                return eval_expression(i->children.begin()) +
                    eval_expression(i->children.begin()+1);
            }
            else if (*i->value.begin() == '-')
            {
                return eval_expression(i->children.begin()) -
                    eval_expression(i->children.begin()+1);
            }
        }

        return 0;
    }
```

An entire working example is **ast_calc.cpp**. Hopefully this example has been enough to whet your appetite for trees. For more nitty-gritty details, keep on reading the rest of this chapter.

Usage

pt_parse

To create a parse tree, you can call one of the five free functions:

```
    template <typename FactoryT, typename IteratorT, typename ParserT, typename SkipT>
    tree_parse_info<IteratorT, FactoryT>
    pt_parse(
        IteratorT const&        first_,
        IteratorT const&        last_,
        parser<ParserT> const&  parser,
        SkipT const&            skip_,
        FactoryT const &        factory_ = FactoryT());
    template <typename IteratorT, typename ParserT, typename SkipT>
    tree_parse_info<IteratorT>
    pt_parse(
        IteratorT const&        first_,
        IteratorT const&        last_,
        parser<ParserT> const&  parser,
        SkipT const&            skip_);
    template <typename IteratorT, typename ParserT>
    tree_parse_info<IteratorT>
    pt_parse(
        IteratorT const&        first_,
        IteratorT const&        last_,
        parser<ParserT> const&  parser);
    template <typename CharT, typename ParserT, typename SkipT>
    tree_parse_info<CharT const*>
    pt_parse(
        CharT const*            str,
        parser<ParserT> const&  parser,
        SkipT const&            skip);
    template <typename CharT, typename ParserT>
    tree_parse_info<CharT const*>
    pt_parse(
        CharT const*            str,
        parser<ParserT> const&  parser);
```

ast_parse

To create an abstract syntax tree (ast for short) you call one of the five free functions:

```
    template <typename FactoryT, typename IteratorT, typename ParserT, typename SkipT>
    tree_parse_info<IteratorT, FactoryT>
    ast_parse(
        IteratorT const&        first_,
        IteratorT const&        last_,
        parser<ParserT> const&  parser,
        SkipT const&            skip_,
        FactoryT const &        factory_ = FactoryT());
    template <typename IteratorT, typename ParserT, typename SkipT>
    tree_parse_info<IteratorT>
    ast_parse(
        IteratorT const&        first_,
        IteratorT const&        last_,
        parser<ParserT> const&  parser,
        SkipT const&            skip_);
    template <typename IteratorT, typename ParserT>
    tree_parse_info<IteratorT>
    ast_parse(
        IteratorT const&        first_,
        IteratorT const&        last,
        parser<ParserT> const&  parser);
    template <typename CharT, typename ParserT, typename SkipT>
    tree_parse_info<CharT const*>
    ast_parse(
        CharT const*            str,
        parser<ParserT> const&  parser,
        SkipT const&            skip);
    template <typename CharT, typename ParserT>
    tree_parse_info<CharT const*>
    ast_parse(
        CharT const*            str,
        parser<ParserT> const&  parser);
```

tree_parse_info

The `tree_parse_info` struct returned from pt_parse and ast_parse contains information about the parse:

```
    template <typename IteratorT = char const*>
    struct tree_parse_info
    {
        IteratorT   stop;
        bool        match;
        bool        full;
        std::size_t length;

        typename tree_match<IteratorT>::container_t trees;
    };
```

tree_parse_info

stop	points to the final parse position (i.e. parsing processed the input up to this point).
match	true if parsing is successful. This may be full (the parser consumed all the input), or partial (the parser consumed only a portion of the input.)
full	true when we have a full match (when the parser consumed all the input).
length	The number of characters consumed by the parser. This is valid only if we have a successful match (either partial or full).
trees	Contains the root node(s) of the tree.

tree_match

When Spirit is generating a tree, the parser's parse() member function will return a tree_match object, instead of a match object. tree_match has three template parameters. The first is the Iterator type which defaults to `char const*`. The second is the node factory, which defaults to **node_val_data_factory**. The third is the attribute type stored in the match. A tree_match has a member variable which is a container (a `std::vector`) of **tree_node** objects named trees. For efficiency reasons, when a tree_match is copied, the trees are **not** copied, they are moved to the new object, and the source object is left with an empty tree container. tree_match supports the same interface as the match class: it has an operator bool() so you can test it for a sucessful match: if (matched), and you can query the match length via the length() function. The class has this interface:

```
    template <typename IteratorT = char const*, typename NodeFactoryT = node_val_data_factory
    struct tree_match
    {
        typedef typename NodeFactoryT::template factory<IteratorT> node_factory_t;
        typedef typename node_factory_t::node_t                    parse_node_t;
        typedef          tree_node<parse_node_t>                   node_t;
        typedef typename node_t::children_t                        container_t;
        typedef typename container_t::iterator                     tree_iterator;
        typedef typename container_t::const_iterator               const_tree_iterator;

        tree_match();
        tree_match(std::size_t length, parse_node_t const& n);
        tree_match(tree_match const& x);
        explicit tree_match(match const& x);
        tree_match& operator=(tree_match const& x);
        void swap(tree_match& x);
        operator bool() const;
        int length() const;

        container_t trees;
    };
```

When a parse has sucessfully completed, the trees data member will contain the root node of the tree.

vector?

You may wonder, why is it a vector then? The answer is that it is partly for implementation purposes, and also if you do not use any rules in your grammar, then trees will contain a sequence of nodes that will not have any children.

Having spirit create a tree is similar to how a normal parse is done:

```
    tree_match<> hit = expression.parse(tree_scanner);

    if (hit)
        process_tree_root(hit.trees[0]); // do something with the tree
```

tree_node

Once you have created a tree by calling **pt_parse** or **ast_parse**, you have a **tree_parse_info** which contains the root node of the tree, and now you need to do something with the tree. The data member trees of **tree_parse_info** is a std::vector<tree_node>. tree_node provides the tree structure. The class has one template parameter named T. tree_node contains an instance of type T. It also contains a std::vector<tree_node<T> > which are the node's children. The class looks like this:

```
    template <typename T>
    struct tree_node
    {
        typedef T parse_node_t;
        typedef std::vector<tree_node<T> > children_t;
        typedef typename children_t::iterator tree_iterator;
        typedef typename children_t::const_iterator const_tree_iterator;

        T value;
        children_t children;

        tree_node();
        explicit tree_node(T const& v);
        tree_node(T const& v, children_t const& c);
        void swap(tree_node<T>& x);
    };
```

This class is simply used to separate the tree framework from the data stored in the tree. It is a generic node and any type can be stored inside it and acessed via the data member value. The default type for T is `node_val_data`.

node_val_data

The `node_val_data` class contains the actual information about each node. This includes the text or token sequence that was parsed, an `id` that indicates which rule created the node, a boolean flag that indicates whether the node was marked as a root node, and an optional user-specified value. This is the interface:

```
    template <typename IteratorT = char const*, typename ValueT = nil_t>
    struct node_val_data
    {
        typedef typename std::iterator_traits<IteratorT>::value_type value_type;
        typedef std::vector<value_type> container_t;
        typedef typename container_t::iterator iterator_t;
        typedef typename container_t::const_iterator const_iterator_t;

        node_val_data();
        node_val_data(IteratorT const& _first, IteratorT const& _last);
        template <typename IteratorT2>
        node_val_data(IteratorT2 const& _first, IteratorT2 const& _last);
        void swap(node_val_data& x);

        container_t::iterator begin();
        container_t::const_iterator begin() const;
        container_t::iterator end();
        container_t::const_iterator end() const;

        bool is_root() const;
        void is_root(bool b);
        parser_id id() const;
        void id(parser_id r);

        ValueT const& value() const;
        void value(ValueT const& v);
    };
```

parser_id, checking and setting

If a node is generated by a rule, it will have an `id` set. Each rule has an id that it sets of all nodes generated by that rule. The id is of type **`parser_id`**. The default id of each rule is set to the address of that rule (converted to an integer). This is not always the most convenient, since the code that processes the tree may not have access to the rules, and may not be able to compare addresses. So, you can override the default id used by each rule by **giving it a specific ID**. Then, when processing the tree you can call `node_val_data::id()` to see which rule created that node.

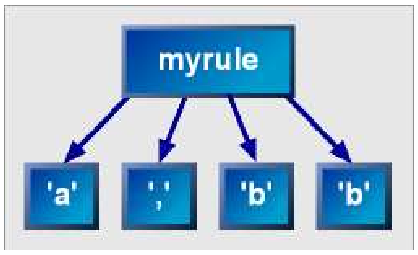
structure/layout of a parse tree

parse tree layout

The tree is organized by the rules. Each rule creates a new level in the tree. All parsers attached to a rule create a node when a sucessful match is made. These nodes become children of a node created by the rule. So, the following code:

```
    rule_t myrule = ch_p('a') >> ',' >> *ch_p('b');
    char const* input = "a,bb";
    scanner_t scanner(input, input + strlen(input));
    tree_match<> m = myrule.parse(scanner);
```

When executed, this code would return a tree_match, m. `m.trees[0]` would contain a tree like this:



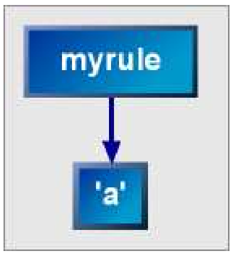
The root node would not contain any text, and it's id would be set to the address of myrule. It would have four children. Each child's id would be set to the address of myrule, would contain the text as shown in the diagram, and would have no children.

ast layout

When calling **ast_parse**, the tree gets generated differently. It mostly works the same as when generating a parse tree. The difference happens when a rule only generated one sub-node. Instead of creating a new level, **ast_parse** will not create a new level, it will just leave the existing node. So, this code:

```
    rule_t myrule = ch_p('a');
    char const* input = "a";
    ast_scanner_t scanner(input, input+strlen(input));
    tree_match<> m = myrule.parse(scanner);
```

will generate a single node that contains 'a'. If `tree_match_policy` had been used instead of `ast_match_policy`, the tree would have looked like this:



ast_match_policy has the effect of eliminating intermediate rule levels which are simply pass-through rules. This is not enough to generate abstract syntax trees, **root_node_d** is also needed. **root_node_d** will be explained later.

switching: gen_pt_node_d[] & gen_ast_node_d[]

If you want to mix and match the parse tree and ast behaviors in your application, you can use the `gen_pt_node_d[]` and `gen_ast_node_d[]` directives. When parsing passes through the `gen_pt_node_d` directive, parse tree creation behavior is turned on. When the `gen_ast_node_d` directive is used, the enclosed parser will generate a tree using the ast behavior. Note that you must pay attention to how your rules are declared if you use a rule inside of these directives. The match policy of the scanner will have to correspond to the desired behavior. If you avoid rules and use primitive parsers or grammars, then you will not have problems.

Directives

There are a few more directives that can be used to control the generation of trees. These directives only effect tree generation. Otherwise, they have no effect.

no_node_d

This directive is similar to `gen_pt_node_d` and `gen_ast_node_d`, in that it modifies the scanner's match policy used by the enclosed parser. As it's name implies, it does no tree generation, it turns it off completely. This is useful if there are parts of your grammar which are not needed in the tree. For instance: keywords, operators (`*`, `-`, `&&`, etc.) By eliminating these from the tree, both memory usage and parsing time can be lowered. This directive has the same requirements with respect to rules as `gen_pt_node_d` and `gen_ast_node_d` do. See the example file xml_grammar.hpp (in libs/spirit/example/application/xml directory) for example usage of `no_node_d[]`.

discard_node_d

This directive has a similar purpose to `no_node_d`, but works differently. It does not switch the scanner's match policy, so the enclosed parser still generates nodes. The generated nodes are discarded and do not appear in the tree. Using `discard_node_d` is slower than `no_node_d`, but it does not suffer from the drawback of having to specify a different rule type for any rule inside it.

leaf_node_d/token_node_d

Both `leaf_node_d` and `token_node_d` work the same. They group together all the nodes generated by the enclosed parser. Note that a rule should not be used inside these directives.

This rule:

```
    rule_t integer = !ch_p('-') >> *(range_p('0', '9'));
```

would generate a root node with the id of integer and a child node for each character parsed

This:

```
    rule_t integer = token_node_d[ !ch_p('-') >> *(range_p('0', '9')) ];
```

would generate a root node with only one child node that contained the entire integer.

infix_node_d

This is useful for removing separators from lists. It discards all the nodes in even positions. Thus this rule:

```
    rule_t intlist = infix_node_d[ integer >> *(',' >> integer) ];
```

would discard all the comma nodes and keep all the integer nodes.

discard_first_node_d

This discards the first node generated.

discard_last_node_d

This discards the last node generated.

inner_node_d

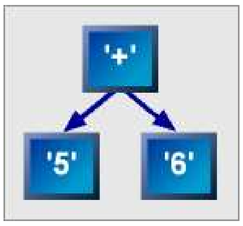
This discards the first and last node generated.

root_node_d and ast generation

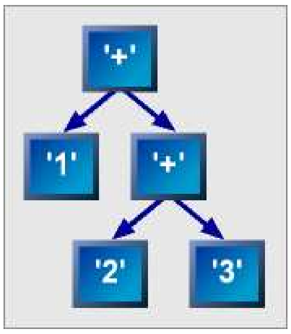
The `root_node_d` directive is used to help out ast generation. It has no effect when generating a parse tree. When a parser is enclosed in `root_node_d`, the node it generates is marked as a root. This affects the way it is treated when it's added to the list of nodes being generated. Here's an example:

```
    rule_t add = integer >> *(root_node_d[ ch_p('+') ] >> integer);
```

When parsing 5+6 the following tree will be generated:



When parsing 1+2+3 the following will be generated:



When a new node is created the following rules are used to determine how the tree will be generated:

```
    Let a be the previously generated node.
    Let b be the new node.

    If b is a root node then

        b's children become a + b's previous children.
        a is the new first child of b.

    else if a is a root node and b is not, then

        b becomes the last child of a.

    else

        a and b become siblings.
```

After parsing leaves the current rule, the root node flag on the top node is turned off. This means that the root_node_d directive only affects the current rule.

The example **ast_calc.cpp** demonstrates the use of root_node_d and **ast_parse**. The full source code can be **viewed here**. This is part of the Spirit distribution.

parse_tree_iterator

The `parse_tree_iterator` class allows you to parse a tree using spirit. The class iterates over the tokens in the leaf nodes in the same order they were created. The `parse_tree_iterator` is templated on `ParseTreeMatchT`. It is constructed with a container of trees, and a position to start. Here is an example usage:

```
    rule_t myrule = ch_p('a');
    char const* input = "a";

    // generate parse tree
    tree_parse_info<> i = pt_parse(input, myrule);

    typedef parse_tree_iterator<tree_match<> > parse_tree_iterator_t;

    // create a first and last iterator to work off the tree
    parse_tree_iterator_t first(i.trees, i.trees.begin());
    parse_tree_iterator_t last;

    // parse the tree
    rule<parse_tree_iterator_t> tree_parser =...
    tree_parser.parse(first, last);
```

advanced tree generation

node value

The `node_val_data` can contain a value. By default it contains a `void_t`, which is an empty class. You can specify the type, using a template parameter, which will then be stored in every node. The type must be default constructible, and assignable. You can get and set the value using

```
    ValueT node_val_data::value;
```

and

```
    void node_val_data::value(Value const& value);
```

To specify the value type, you must use a different **node_val_data** factory than the default. The following example shows how to modify the factory to store and retrieve a double inside each node_val_data.

```
    typedef node_val_data_factory<double> factory_t;
    my_grammar gram;
    my_skip_grammar skip;
    tree_parse_info<iterator_t, factory_t> i =
        ast_parse<factory_t>(first, last, gram, skip);
    // access the double in the root node
    double d = i.trees.begin()->value;
```

access_node_d

Now, you may be wondering, "What good does it do to have a value I can store in each node, but not to have any way of setting it?" Well, that's what `access_node_d` is for. `access_node_d` is a directive. It allows you to attach an action to it, like this:

```
    access_node_d[...some parsers...][my_action()]
```

The attached action will be passed 3 parameters: A reference to the root node of the tree generated by the parser, and the current first and last iterators. The action can set the value stored in the node.

Tree node factories

By setting the factory, you can control what type of nodes are created and how they are created. There are 3 predefined factories: node_val_data_factory, node_all_val_data_factory, and node_iter_data_factory. You can also create your own factory to support your own node types.

Using factories with grammars is quite easy, you just need to specify the factory type as explicit template parameter to the free ast_parse function:

```
    typedef node_iter_data_factory<int> factory_t;
    my_grammar gram;
    my_skip_grammar skip;
    tree_parse_info<iterator_t, factory_t> i =
        ast_parse<factory_t>(first, last, gram, skip);
```

Instead, using the factory directly with rules is slightly harder because the factory is a template parameter to the scanner match policy, so you must use a custom scanner:

```
    typedef spirit::void_t value_t;
    typedef node_val_data_factory<value_t> factory_t;
    typedef tree_match<iterator_t, factory_t> match_t;
    typedef ast_match_policy<iterator_t, factory_t> match_policy_t;
    typedef scanner<iterator_t, scanner_policies<iter_policy_t, match_policy_t> > scanner_t;
    typedef rule<scanner_t> rule_t;

    rule_t r =...;

    scanner_t scan = scanner_t(first, last);
    match_t hit = r.parse(scan);
```

node_val_data_factory

This is the default factory. It creates node_val_data nodes. Leaf nodes contain a copy of the matched text, and intermediate nodes don't. node_val_data_factory has one template parameter: ValueT. ValueT specifies the type of value that will be stored in the node_val_data.

node_all_val_data_factory

This factory also creates node_val_data. The difference between it and node_val_data_factory is that **every** node contains all the text that spans it. This means that the root node will contain a copy of the entire parsed input sequence. node_all_val_data_factory has one template parameter: ValueT. ValueT specifies the type of value that will be stored in the node_val_data.

node_iter_data_factory

This factory creates the parse_tree_iter_node. This node stores iterators into the input sequence instead of making a copy. It can use a lot less memory. However, the input sequence must stay valid for the life of the tree, and it's not a good idea to use the multi_pass iterator with this type of node. All levels of the tree will contain a begin and end iterator. node_iter_data_factory has one template parameter: ValueT. ValueT specifies the type of value that will be stored in the node_val_data.

custom

You can create your own factory. It should look like this:

```
    class my_factory
    {
    public:

        // This inner class is so that the factory can simulate
        // a template template parameter

        template <typename IteratorT>
        class factory
        {
        public:

            // This is your node type
            typedef my_node_type node_t;

            static node_t create_node(
                IteratorT const& first, IteratorT const& last, bool is_leaf_node)
            {
                // create a node and return it.
            }

            // This function is used by the leaf_node and token_node directives.
            // If you don't use them, then you can leave this function
            // unimplemented.

            template <typename ContainerT>
            static node_t group_nodes(ContainerT const& nodes)
            {
                // Group all the nodes into one and return it.
            }
        };
    };


    // Typedefs to use my_factory
    typedef my_factory factory_t;
    typedef tree_match<iterator_t, factory_t> match_t;
    typedef tree_match_policy<iterator_t, factory_t> match_policy_t;

    // Typedefs if you are using rules instead of grammars
    typedef scanner<iterator_t, scanner_policies<iter_policy_t, match_policy_t> > scanner_t;
    typedef rule<scanner_t> rule_t;
```

The multi_pass

Backtracking in Spirit requires the use of the following types of iterator: forward, bidirectional, or random access. Because of backtracking, input iterators cannot be used. Therefore, the standard library classes istreambuf_iterator and istream_iterator, that fall under the category of input iterators, cannot be used. Another input iterator that is of interest is one that wraps a lexer, such as LEX.

Input Iterators

In general, Spirit is a backtracking parser. This is not an absolute requirement though. In the future, we shall see more deterministic parsers that require no more than 1 character (token) of lookahead. Such parsers allow us to use input iterators such as the istream_iterator as is.

Unfortunately, with an input iterator, there is no way to save an iterator position, and thus input iterators will not work with backtracking in Spirit. One solution to this problem is to simply load all the data to be parsed into a container, such as a vector or deque, and then pass the begin and end of the container to Spirit. This method can be too memory intensive for certain applications, which is why the multi_pass iterator was created.

The multi_pass iterator will convert any input iterator into a forward iterator suitable for use with Spirit. multi_pass will buffer data when needed and will discard the buffer when only one copy of the iterator exists.

A grammar must be designed with care if the multi_pass iterator is used. Any rule that may need to backtrack, such as one that contains an alternative, will cause data to be buffered. The rules that are optimal to use are sequence and repetition. Sequences of the form `a >> b` will not buffer data at all. Any rule that repeats, such as kleene_star (`*a`) or positive such as (`+a`), will only buffer the data for the current repetition.

In typical grammars, ambiguity and therefore lookahead is often localized. In fact, many well designed languages are fully deterministic and require no lookahead at all. Peeking at the first character from the input will immediately determine the alternative branch to take. Yet, even with highly ambiguous grammars, alternatives are often of the form `*(a | b | c | d)`. The input iterator moves on and is never stuck at the beginning. Let's look at a Pascal snippet for example:

```
    program =
        programHeading >> block >> '.'
        ;

    block =
       *(     labelDeclarationPart
        |     constantDefinitionPart
        |     typeDefinitionPart
        |     variableDeclarationPart
        |     procedureAndFunctionDeclarationPart
        )
        >>    statementPart
        ;
```

Notice the alternatives inside the Kleene star in the rule block . The rule gobbles the input in a linear manner and throws away the past history with each iteration. As this is fully deterministic LL(1) grammar, each failed alternative only has to peek 1 character (token). The alternative that consumes more than 1 character (token) is definitely a winner. After which, the Kleene star moves on to the next.

Be mindful if you use the free parse functions. All of these make a copy of the iterator passed to them.

Now, after the lecture on the features to be careful with when using multi_pass, you may think that multi_pass is way too restrictive to use. That's not the case. If your grammar is deterministic, you can make use of flush_multi_pass in your grammar to ensure that data is not buffered when unnecessary.

Again, following up the example we started to use in the section on the scanner . Here's an example using the multi_pass: This time around we are extracting our input from the input stream using an istreambuf_iterator.

```
    #include <boost/spirit/core.hpp>
    #include <boost/spirit/iterator/multi_pass.hpp>

    using namespace boost::spirit;
    using namespace std;

    ifstream in("input_file.txt"); // we get our input from this file

    typedef char char_t;
    typedef multi_pass<istreambuf_iterator<char_t> > iterator_t;

    typedef skip_parser_iteration_policy<space_parser> iter_policy_t;
    typedef scanner_policies<iter_policy_t> scanner_policies_t;
    typedef scanner<iterator_t, scanner_policies_t> scanner_t;

    typedef rule<scanner_t> rule_t;

    iter_policy_t iter_policy(space_p);
    scanner_policies_t policies(iter_policy);
    iterator_t first(
        make_multi_pass(std::istreambuf_iterator<char_t>(in)));

    scanner_t scan(
        first, make_multi_pass(std::istreambuf_iterator<char_t>()),
        policies);

    rule_t      n_list = real_p >> *(',' >> real_p);
    match<>     m = n_list.parse(scan);
```

flush_multi_pass

There is a predefined pseudo-parser called flush_multi_pass. When this parser is used with multi_pass, it will call multi_pass::clear_queue(). 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.

multi_pass Policies

multi_pass is a templated policy driven class. The description of multi_pass above is how it was originally implemented (before it used policies), and is the default configuration now. But, multi_pass is capable of much more. Because of the open-ended nature of policies, you can write your own policy to make multi_pass behave in a way that we never before imagined.

The multi_pass class has five template parameters:

- InputT - The type multi_pass uses to acquire it's input. This is typically an input iterator, or functor.
- InputPolicy - A class that defines how multi_pass acquires it's input. The InputPolicy is parameterized by InputT.
- OwnershipPolicy - This policy determines how multi_pass deals with it's shared components.
- CheckingPolicy - This policy determines how checking for invalid iterators is done.
- StoragePolicy - The buffering scheme used by multi_pass is determined and managed by the StoragePolicy.

Predefined policies

All predefined multi_pass policies are in the namespace boost::spirit::multi_pass_policies.

Predefined InputPolicy classes

input_iterator

This policy directs multi_pass to read from an input iterator of type InputT.

lex_input

This policy obtains it's input by calling yylex(), which would typically be provided by a scanner generated by LEX. If you use this policy your code must link against a LEX generated scanner.

functor_input

This input policy obtains it's data by calling a functor of type InputT. The functor must meet certain requirements. It must have a typedef called result_type which should be the type returned from operator(). Also, since an input policy needs a way to determine when the end of input has been reached, the functor must contain a static variable named eof which is comparable to a variable of result_type.

Predefined OwnershipPolicy classes

ref_counted

This class uses a reference counting scheme. multi_pass will delete it's shared components when the count reaches zero.

first_owner

When this policy is used, the first multi_pass created will be the one that deletes the shared data. Each copy will not take ownership of the shared data. This works well for spirit, since no dynamic allocation of iterators is done. All copies are made on the stack, so the original iterator has the longest lifespan.

Predefined CheckingPolicy classes

no_check

This policy does no checking at all.

buf_id_check

buf_id_check keeps around a buffer id, or a buffer age. Every time clear_queue() is called on a multi_pass iterator, it is possible that all other iterators become invalid. When clear_queue() is called, buf_id_check increments the buffer id. When an iterator is dereferenced, this policy checks that the buffer id of the iterator matches the shared buffer id. This policy is most effective when used together with the std_deque StoragePolicy. It should not be used with the fixed_size_queue StoragePolicy, because it will not detect iterator dereferences that are out of range.

full_check

This policy has not been implemented yet. When it is, it will keep track of all iterators and make sure that they are all valid.

Predefined StoragePolicy classes

std_deque

This policy keeps all buffered data in a std::deque. All data is stored as long as there is more than one iterator. Once the iterator count goes down to one, and the queue is no longer needed, it is cleared, freeing up memory. The queue can also be forcibly cleared by calling multi_pass::clear_queue().

fixed_size_queue<N>

fixed_size_queue keeps a circular buffer that is size N+1 and stores N elements. fixed_size_queue is a template with a std::size_t parameter that specified the queue size. It is your responsibility to ensure that N is big enough for your parser. Whenever the foremost iterator is incremented, the last character of the buffer is automatically erased. Currently there is no way to tell if an iterator is trailing too far behind and has become invalid. No dynamic allocation is done by this policy during normal iterator operation, only on initial construction. The memory usage of this StoragePolicy is set at N+1 bytes, unlike std_deque, which is unbounded.

Combinations: How to specify your own custom multi_pass

The beauty of policy based designs is that you can mix and match policies to create your own custom class by selecting the policies you want. Here's an example of how to specify a custom multi_pass that wraps an istream_iterator<char>, and is slightly more efficient than the default because it uses the first_owner OwnershipPolicy and the no_check CheckingPolicy:

```
    typedef multi_pass<
        istream_iterator<char>,
        multi_pass_policies::input_iterator,
        multi_pass_policies::first_owner,
        multi_pass_policies::no_check,
        multi_pass_policies::std_deque
    > first_owner_multi_pass_t;
```

The default template parameters for multi_pass are: input_iterator InputPolicy, ref_counted OwnershipPolicy, buf_id_check CheckingPolicy and std_deque StoragePolicy. So if you use multi_pass<istream_iterator<char> > you will get those pre-defined behaviors while wrapping an istream_iterator<char>.

There is one other pre-defined class called look_ahead. look_ahead has two template parameters: InputT, the type of the input iterator to wrap, and a std::size_t N, which specifies the size of the buffer to the fixed_size_queue policy. While the default multi_pass configuration is designed for safey, look_ahead is designed for speed. look_ahead is derived from a multi_pass with the following policies: input_iterator InputPolicy, first_owner

OwnershipPolicy, no_check CheckingPolicy, and fixed_size_queue<N> StoragePolicy.

How to write a functor for use with the functor_input InputPolicy

If you want to use the functor_input InputPolicy, you can write your own functor that will supply the input to multi_pass. The functor must satisfy two requirements. It must have a typedef result_type which specifies the return type of operator(). This is standard practice in the STL. Also, it must supply a static variable called eof which is compared against to know whether the input has reached the end. Here is an example:

```
    class my_functor
    {
    public:

        typedef char result_type;

        my_functor()
        : c('A') {}

        char operator()() const
        {
            if (c == 'M')
                return eof;
            else
                return c++;
        }

        static result_type eof;

    private:

        char c;
    };

    my_functor::result_type my_functor::eof = '\0';

    typedef multi_pass<
        my_functor,
        multi_pass_policies::functor_input,
        multi_pass_policies::first_owner,
        multi_pass_policies::no_check,
        multi_pass_policies::std_deque
    > functor_multi_pass_t;

    functor_multi_pass_t first = functor_multi_pass_t(my_functor());
    functor_multi_pass_t last;
```

How to write policies for use with multi_pass

InputPolicy

An InputPolicy must have the following interface:

```
    class my_input_policy // your policy name
    {
    public:

    //  class inner will be instantiated with the type given
    //  as the InputT parameter to multi_pass.

        template <typename InputT>
        class inner
        {
        public:
            // these typedefs determine the iterator_traits for multi_pass
            typedef x value_type;
            typedef x difference_type;
            typedef x pointer;
            typedef x reference;

        protected:

            inner();
            inner(InputT x);
            inner(inner const& x);
            // delete or clean up any state
            void destroy();
            // return true if *this and x have the same input
            bool same_input(inner const& x) const;
            void swap(inner& x);

        public:

            // get an instance from the input
            result_type get_input() const;
            // advance the input
            void advance_input();
            // return true if the input is at the end
            bool input_at_eof() const;
        };
    };
```

Because of the way that multi_pass shares a buffer and input among multiple copies, class inner should keep a pointer to it's input. The copy constructor should simply copy the pointer. destroy() should delete it. same_input should compare the pointers. For more details see the various implementations of InputPolicy classes.

OwnershipPolicy

The OwnershipPolicy must have the following interface:

```
    class my_ownership_policy
    {
    protected:

        my_ownership_policy();
        my_ownership_policy(my_ownership_policy const& x);
        // clone is called when a copy of the iterator is made
        void clone();
        // called when a copy is deleted. Return true to indicate
        // resources should be released
        bool release();
        void swap(my_ownership_policy& x);

    public:
        // returns true if there is only one iterator in existence.
        // std_dequeue StoragePolicy will free it's buffered data if this
        // returns true.
        bool unique() const;
    };
```

CheckingPolicy

The CheckingPolicy must have the following interface:

```
    class my_check
    {
    protected:

        my_check();
        my_check(my_check const& x);
        void destroy();
        void swap(my_check& x);
        // check should make sure that this iterator is valid
        void check() const;
        void clear_queue();
    };
```

StoragePolicy

A StoragePolicy must have the following interface:

```
    class my_storage_policy
    {
    public:

    // class inner will be instantiated with the value_type from the InputPolicy

        template <typename ValueT>
        class inner
        {
        protected:

            inner();
            inner(inner const& x);
            // will be called from the destructor of the last iterator.
            void destroy();
            void swap(inner& x);
            // This is called when the iterator is dereferenced. It's a template
            // method so we can recover the type of the multi_pass iterator
            // and access it.
            template <typename MultiPassT>
            static ValueT dereference(MultiPassT const& mp);
            // This is called when the iterator is incremented. It's a template
            // method so we can recover the type of the multi_pass iterator
            // and access it.
            template <typename MultiPassT>
            static void increment(MultiPassT& mp);
            void clear_queue();
            // called to determine whether the iterator is an eof iterator
            template <typename MultiPassT>
            static bool is_eof(MultiPassT const& mp);
            // called by operator==
            bool equal_to(inner const& x) const;
            // called by operator<
            bool less_than(inner const& x) const;
        };  // class inner
    };
```

A StoragePolicy is the trickiest policy to write. You should study and understand the existing StoragePolicy classes before you try and write your own.

File Iterator

Since Spirit is a back-tracking parser, it requires at least a forward iterator. In particular, an input iterator is not sufficient. Many times it is convenient to read the input to a parser from a file, but the STL file iterators are input iterators. To get around this limitation, Spirit has a utility class `file_iterator`, which is a read-only random-access iterator for files.

To use the Spirit file iterator, simply create a file iterator with the path to the file you wish to parse, and then create an EOF iterator for the file:

```
    #include <boost/spirit/iterator/file_iterator.hpp> // the header file
```

```
    file_iterator<> first("input.dat");

    if (!first)
    {
       std::cout << "Unable to open file!\n";

       // Clean up, throw an exception, whatever
       return -1;
    }

    file_iterator<> last = first.make_end();
```

You now have a pair of iterators to use with Spirit . If your parser is fully parametrized (no hard-coded `<char const *>`), it is a simple matter of redefining the iterator type to `file_iterator`:

```
    typedef char                        char_t;
    typedef file_iterator <char_t>      iterator_t;
    typedef scanner<iterator_t>         scanner_t;
    typedef rule <scanner_t>            rule_t;

    rule_t my_rule;

    // Define your rule

    parse_info<iterator_t> info = parse(first, last, my_rule);
```

Of course, you don't have to deal with the **scanner-business** at all if you use grammars rather than rules as arguments to the parse functions. You simply pass the iterator pairs and the grammar as is:

```
    my_grammar g;
    parse_info<iterator_t> info = parse(first, last, g);
```

Generic iterator

The Spirit file iterator can be parameterized with any type that is default constructible and assignable. It transparently supports large files (greater than 2GB) on systems that provide an appropriate interface. The file iterator can be useful outside of Spirit as well. For instance, the Boost.Tokenizer package requires a bidirectional iterator, which is provided by file_iterator.

See **file_parser.cpp** for a compilable example. This is part of the Spirit distribution.

Position Iterator

Often, when writing a parser that is able to detect errors in the format of the input stream, we want it to communicate to the user where the error happened within that input. The classic example is when writing a compiler or interpreter that detects syntactical errors in the parsed program, indicating the line number and maybe even the position within the line where the error was found.

The class position_iterator is a tool provided within Spirit that allows parser writers to easily implement this functionality. The concept is quite simple: this class is an iterator wrapper that keeps track of the current position within the file, including current file, line and column. It requires a single template parameter, which should be the type of the iterator that is to be wrapped.

To use it, you'll need to add the following include:

```
    #include <boost/spirit/iterator/position_iterator.hpp>
```

Or include all the iterators in Spirit:

```
    #include <boost/spirit/iterator.hpp>
```

To construct the wrapper, it needs both the begin and end iterators of the input sequence, and the file name of the input sequence. Optionally, you can also specify the starting line and column numbers, which default to 1. Default construction, with no parameters, creates a generic end-of-sequence iterator, in a similar manner as it's done in the stream operators of the standard C++ library.

The wrapped iterator must belong to the input or forward iterator category, and the position_iterator just inherits that category.

For example, to create begin and end positional iterators from an input C- string, you'd use:

```
    char const* inputstring = "...";
    typedef position_iterator<char const*> iterator_t;

    iterator_t begin(inputstring, inputstring+strlen(inputstring));
    iterator_t end;
```

Operations

```
    void set_position(file_position const&);
```

Call this function when you need to change the current position stored in the iterator. For example, if parsing C-style #include directives, the included file's input must be marked by restarting the file and column to 1 and 1 and the name to the new file's name.

```
    file_position const& get_position() const;
```

Call this function to retrieve the current position.

```
    void set_tabchars(int);
```

Call this to set the number of tabs per character. This value is necessary to correctly track the column number.

file_position

file_position is a structure that holds the position within a file. Its fields are:

file_position fields

`std::string file;`	Name of the file. Hopefully a full pathname
`int line;`	Line number within the file. By default, the first line is number 1
`int column;`	Column position within the file. The first column is 1

See **position_iterator.cpp** for a compilable example. This is part of the Spirit distribution.

Debugging

The top-down nature of Spirit makes the generated parser easy to micro- debug using the standard debugger bundled with the C++ compiler we are using. With recursive-descent, the parse traversal utilizes the hardware stack through C++ function call mechanisms. There are no difficult to debug tables or state machines that obscure the parsing logic flow. The stack trace we see in the debugger follows faithfully the hierarchical grammar structure.

Since any production rule can initiate a parse traversal , it is a lot easier to pinpoint the bugs by focusing on one or a few rules. For relatively complex parsing tasks, the same way we write robust C++ programs, it is advisable to develop a grammar iteratively on a per-module basis where each module is a small subset of the complete grammar. That way, we can stress-test individual modules piecemeal until we reach the top-most module. For instance, when developing a scripting language, we can start with expressions, then move on to statements, then functions, upwards until we have a complete grammar.

At some point when the grammar gets quite complicated, it is desirable to visualize the parse traversal and see what's happening. There are some facilities in the framework that aid in the visualisation of the parse traversal for the purpose of debugging. The following macros enable these features.

Debugging Macros

BOOST_SPIRIT_ASSERT_EXCEPTION

Spirit contains assertions that may activate when spirit is used incorrectly. By default these assertions use the assert macro from the standard library. If you want spirit to throw an exception instead, define `BOOST_SPIRIT_ASSERT_EXCEPTION` to the name of the class that you want to be thrown. This class's constructor will be passed a `const char*` stringified version of the file, line, and assertion condition, when it is thrown. If you want to totally disable the assertion, `#define NDEBUG`.

BOOST_SPIRIT_DEBUG

Define this to enable debugging.

With debugging enabled, special output is generated at key points of the parse process, using the standard output operator (`operator<<`) with `BOOST_SPIRIT_DEBUG_OUT` (default is `std::cout`, see below) as its left operand.

> In order to use spirit's debugging support you must ensure that appropriate overloads of `operator<<` taking `BOOST_SPIRIT_DEBUG_OUT` as its left operand are available. The expected semantics are those of the standard output operator.
>
> These overloads may be provided either within the namespace where the corresponding class is declared (will be found

through Argument Dependent Lookup) or [within an anonymous namespace] within namespace boost::spirit, so it is visible where it is called.

⚠ Note in particular that when BOOST_SPIRIT_DEBUG_FLAGS_CLOSURES is set, overloads of operator<< taking instances of the types used in closures as their right operands are required.

You may find an example of overloading the output operator for std::pair in a **related FAQ entry**.

By default, if the BOOST_SPIRIT_DEBUG macro is defined, all available debug output is generated. To fine tune the amount of generated text you can define the BOOST_SPIRIT_DEBUG_FLAGS constant to be equal of a combination of the following flags:

Available flags to fine tune debug output

BOOST_SPIRIT_DEBUG_FLAGS_NODES	print information about nodes (general for all parsers)
BOOST_SPIRIT_DEBUG_FLAGS_TREES	print information about parse trees and AST's (general for all tree parsers)
BOOST_SPIRIT_DEBUG_FLAGS_CLOSURES	print information about closures (general for all parsers with closures)
BOOST_SPIRIT_DEBUG_FLAGS_ESCAPE_CHAR	print information out of the esc_char_parser
BOOST_SPIRIT_DEBUG_FLAGS_SLEX	print information out of the SLEX parser

BOOST_SPIRIT_DEBUG_OUT

Define this to redirect the debugging diagnostics printout to somewhere else (e.g. a file or stream). Defaults to std::cout.

BOOST_SPIRIT_DEBUG_TOKEN_PRINTER

The BOOST_SPIRIT_DEBUG_TOKEN_PRINTER macro allows you to redefine the way characters are printed on the stream.

If BOOST_SPIRIT_DEBUG_OUT is of type StreamT, the character type is CharT and BOOST_SPIRIT_DEBUG_TOKEN_PRINTER is defined to foo, it must be compatible with this usage:

```
    foo(StreamT, CharT)
```

The default printer requires operator<<(StreamT, CharT) to be defined. Additionaly, if CharT is convertible to a normal character type (char, wchar_t or int), it prints control characters in a friendly manner (e.g., when it receives '\n' it actually prints the \ and n charactes, instead of a newline).

BOOST_SPIRIT_DEBUG_PRINT_SOME

The BOOST_SPIRIT_DEBUG_PRINT_SOME constant defines the number of characters from the stream to be printed for diagnosis. This defaults to the first 20 characters.

BOOST_SPIRIT_DEBUG_TRACENODE

By default all parser nodes are traced. This constant may be used to redefine this default. If this is `1` (`true`), then tracing is enabled by default, if this constant is `0` (`false`), the tracing is disabled by default. This preprocessor constant is set to `1` (`true`) by default.

Please note, that the following `BOOST_SPIRIT_DEBUG_...()` macros are to be used at function scope only.

BOOST_SPIRIT_DEBUG_NODE(p)

Define this to print some debugging diagnostics for parser p. This macro

- Registers the parser name for debugging
- Enables/disables the tracing for parser depending on `BOOST_SPIRIT_DEBUG_TRACENODE`

Pre-parse: Before entering the rule, the rule name followed by a peek into the data at the current iterator position is printed.

Post-parse: After parsing the rule, the rule name followed by a peek into the data at the current iterator position is printed. Here, `'/'` before the rule name flags a succesful match while `'#'` before the rule name flags an unsuccesful match.

The following are synonyms for `BOOST_SPIRIT_DEBUG_NODE`

1. BOOST_SPIRIT_DEBUG_RULE
2. BOOST_SPIRIT_DEBUG_GRAMMAR

BOOST_SPIRIT_DEBUG_TRACE_NODE(p, flag)

Similar to `BOOST_SPIRIT_DEBUG_NODE`. Additionally allows selective debugging. This is useful in situations where we want to debug just a hand picked set of nodes.

The following are synonyms for `BOOST_SPIRIT_DEBUG_TRACE_NODE`

1. BOOST_SPIRIT_DEBUG_TRACE_RULE
2. BOOST_SPIRIT_DEBUG_TRACE_GRAMMAR

BOOST_SPIRIT_DEBUG_TRACE_NODE_NAME(p, name, flag)

Similar to `BOOST_SPIRIT_DEBUG_NODE`. Additionally allows selective debugging and allows to specify the name used during debug printout. This is useful in situations where we want to debug just a hand picked set of nodes. The `name` may be redefined in situations, where the parser parameter does not reflect the name of the parser to debug.

The following are synonyms for `BOOST_SPIRIT_DEBUG_TRACE_NODE`

1. BOOST_SPIRIT_DEBUG_TRACE_RULE_NAME
2. BOOST_SPIRIT_DEBUG_TRACE_GRAMMAR_NAME

Here's the original calculator with debugging features enabled:

```
    #define BOOST_SPIRIT_DEBUG  ///$$$ DEFINE THIS BEFORE ANYTHING ELSE $$$///
    #include "boost/spirit.hpp"
    /***/

    /*** CALCULATOR GRAMMAR DEFINITIONS HERE ***/

    BOOST_SPIRIT_DEBUG_RULE(integer);
    BOOST_SPIRIT_DEBUG_RULE(group);
    BOOST_SPIRIT_DEBUG_RULE(factor);
    BOOST_SPIRIT_DEBUG_RULE(term);
    BOOST_SPIRIT_DEBUG_RULE(expr);
```

Be sure to add the macros **inside** the grammar definition's constructor. Now here's a sample session with the calculator.

```
    Type an expression...or [q or Q] to quit

    1 + 2

    grammar(calc):      "1 + 2"
      rule(expression): "1 + 2"
        rule(term):     "1 + 2"
          rule(factor): "1 + 2"
            rule(integer):      "1 + 2"
    push        1
            /rule(integer):     " + 2"
          /rule(factor):        " + 2"
        /rule(term):    " + 2"
        rule(term):     "2"
          rule(factor): "2"
            rule(integer):      "2"
    push        2
            /rule(integer):     ""
          /rule(factor):        ""
        /rule(term):    ""
    popped 1 and 2 from the stack. pushing 3 onto the stack.
      /rule(expression):        ""
    /grammar(calc):     ""
    -------------------------
    Parsing succeeded
    result = 3
    -------------------------
```

We typed in "1 + 2". Notice that there are two successful branches from the top rule `expr`. The text in red is generated by the parser's semantic actions while the others are generated by the debug-diagnostics of our rules. Notice how the first `integer` rule took "1", the first `term` rule took "+" and finally the second `integer` rule took "2".

Please note the special meaning of the first characters appearing on the printed lines:

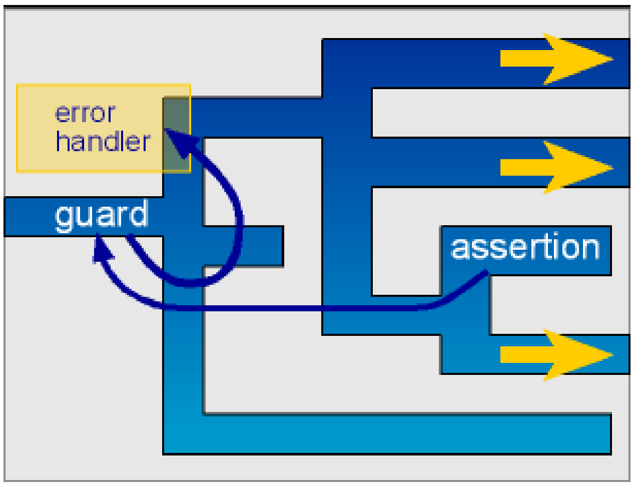
- a single '/' starts a line containing the information about a successfully matched parser node (`rule<>`, `grammar<>` or `subrule<>`)
- a single '#' starts a line containing the information about a failed parser node
- a single '^' starts a line containing the first member (return value/synthesised attribute) of the closure of a successfully matched parser node.

Check out **calc_debug.cpp** to see debugging in action.

Error Handling

C++'s exception handling mechanism is a perfect match for error handling in the framework. Imagine a complete parser as a maze. At each branch, the input dictates where we will turn. Given an erroneous input, we may reach a dead end. If we ever reach one, it would be a waste of time to backtrack from where we came from. Instead, we supply guards in strategic points. Beyond a certain point, we put put parser assertions in places where one is not allowed to go.

The assertions are like springs that catapult us back to the guard. If we ever reach a brick wall given a specific input pattern, everything unwinds quickly and we are thrown right back to the guard. This can be a very effective optimization when used wisely. Right back at the guard, we have a chance to correct the situation, if possible. The following illustration depicts the scenario.



Parser Errors

The `parser_error` class is the generic parser exception class used by Spirit. This is the base class for all parser exceptions.

```
    template <typename ErrorDescrT, typename IteratorT = char const*>
    class parser_error
    {
    public:
        parser_error(IteratorT where, ErrorDescrT descriptor);
        IteratorT where;
        ErrorDescrT descriptor;
    };
```

The exception holds the iterator position where the error was encountered in its `where` member variable. In addition to the iterator, `parser_error` also holds information regarding the error (error descriptor) in its `descriptor` member variable.

Semantic actions are free to throw parser exceptions when necessary. A utility function throw_ may be called. This function creates and throws a parser_error given an iterator and an error descriptor:

```
    template <typename ErrorDescrT, typename IteratorT>
    void throw_(IteratorT where, ErrorDescrT descriptor);
```

Parser Assertions

Assertions may be put in places where we don't have any other option other than expect parsing to succeed. If parsing fails, a specific type of exception is thrown.

Before declaring the grammar, we declare some assertion objects. assertion is a template class parameterized by the type of error that will be thrown once the assertion fails. The following assertions are parameterized by a user defined Error enumeration.

Examples

```
    enum Errors
    {
        program_expected,
        begin_expected,
        end_expected
    };

    assertion<Errors> expect_program(program_expected);
    assertion<Errors> expect_begin(begin_expected);
    assertion<Errors> expect_end(end_expected);
```

The example above uses enums to hold the information regarding the error, we are free to use other types such as integers and strings. For example, assertion<string> accepts a string as its info. It is advisable to use light-weight objects though, after all, error descriptors are usually static. Enums are convenient for error handlers to detect and easily catch since C++ treats enums as unique types.

The assertive_parser

Actually, the expression expect_end(str_p("end"))creates an assertive_parser object. An assertive_parser is a parser that throws an exception in response to a parsing failure. The assertive_parser throws a parser_error exception rather than returning an unsuccessful match to signal that the parser failed to match the input. During parsing, parsers are given an iterator of type IteratorT. This is combined with the error descriptor type ErrorDescrT of the assertion (in this case enum Errors). Both are used to create a parser_error<Errors, IteratorT> which is then thrown to signal the exception.

The predeclared expect_end assertion object may now be used in the grammar as wrappers around parsers. For example:

```
    expect_end(str_p("end"))
```

This will throw an exception if it fails to see "end" from the input.

The Guard

The `guard` is used to catch a specific type of `parser_error`. guards are typically predeclared just like assertions. Extending our previous example:

```
    guard<Errors> my_guard;
```

`Errors`, in this example is the error descriptor type we want to detect. This is the same enum as above. `my_guard` may now be used in a grammar declaration:

```
    my_guard(p)[error_handler]
```

where `p` is an expression that evaluates to a parser. Somewhere inside `p`, a parser may throw a parser exception. `error_handler` is the error handler which may be a function or functor compatible with the interface:

```
    error_status<T>
    f(ScannerT const& scan, ErrorT error);
```

Where scan points to the scanner state prior to parsing and error is the error that arose. The handler is allowed to move the scanner position as it sees fit, possibly in an attempt to perform error correction. The handler must then return an `error_status<T>` object.

The fallback_parser

The expression `my_guard(expr, error_handler)`creates a fallback_parser object. The fallback_parser handles parser_error exceptions of a specific type. Since `my_guard` is declared as `guard<Errors>`, the fallback_parser catches `Errors` specific parser errors: `parser_error<Errors, IteratorT>`. The class sets up a try block. When an exception is caught, the catch block then calls the error_handler.

error_status<T>

```
    template <typename T = nil_t>
    struct error_status
    {
        enum result_t { fail, retry, accept, rethrow };

        error_status(
            result_t result = fail,
            int length      = -1,
            T const& value  = T());

        result_t    result;
        int          length;
        T            value;
    };
```

Where T is an attribute type compatible with the match attribute of the fallback_parser's subject (defaults to nil_t). The class error_status reports the result of an error handler. This result can be one of:

error_status result	
fail	quit and fail. Return a no_match
retry	attempt error recovery, possibly moving the scanner
accept	force success returning a matching length, moving the scanner appropriately and returning an attribute value
rethrow	rethrows the error

See **error_handling.cpp** for a compilable example. This is part of the Spirit distribution.

Quick Reference

This isn't intended to be a full, detailed reference; nor is it intended to be of any use to readers who aren't already familiar with Spirit. It's just a brief reminder of the syntax and behaviour of each component, with links to the full documentation.

Null parsers

`end_p`	Matches EOF	*iter,iter*

`eps_p`
`eps_p(P)`
	Matches without consuming text	*iter,iter*

`epsilon_p`
`epsilon_p(P)`
	Synonym for **eps_p**	*iter,iter*

`nothing_p`	Always fails	*iter,iter*

Character parsers

`alnum_p`	Matches any alphanumeric character	*char*
`alpha_p`	Matches any letter	*char*
`anychar_p`	Matches any character	*char*
`blank_p`	Matches a space or tab	*char*
`ch_p(char)`	Matches a character	*char*

Compound parsers

`confix_p(open, exp, close)`	Matches **open >> (exp - close) >> close**
`do_p[P].while_p(cond)`	Matches while a condition is true (at least once)
`for_p(init, cond, step)[P]`	Matches in a loop
`functor_parser<func>`	Wraps an external parser

`if_p(cond)[P]`
`if_p(cond)[P].else_p[P]`
	Matches depending on a condition

`lazy_p(P)`	Evaluates a parser at run time

`list_p`
`list_p(del)`
`list_p(item, del)`
	Matches a delimited list

`chset_p(charset)`	Matches a character in the set	*char*
`cntrl_p`	Matches any control character	*char*
`digit_p`	Matches any digit	*char*
`f_ch_p(func)`	Matches a character	*char*
`f_range_p(func1, func2)`	Matches any character in the inclusive range	*char*
`graph_p`	Matches any non-space printable character	*char*
`lower_p`	Matches any lower-case letter	*char*
`print_p`	Matches any printable character	*char*
`punct_p`	Matches any punctuation mark	*char*
`range_p(char1, char2)`	Matches any character in the inclusive range	*char*
`sign_p`	Matches a plus or minus sign	*bool*
`space_p`	Matches any whitespace character	*char*
`upper_p`	Matches any upper-case letter	*char*
`xdigit_p`	Matches any hexadecimal digit	*char*

Number parsers

`bin_p`	Matches an unsigned binary integer	*numeric*

`list_p(item, del, end)`	

`repeat_p(num)[P]`
`repeat_p(min, max)[P]`
`repeat_p(min, more)[P]`
	Matches multiple times

`while_p (cond) [P]`	Matches while a condition is true

General directives

`as_lower_d[P]`	Converts text to lower case before matching
`attach_action_d[(P1 op P2)[act]]`	Transforms to **P1 [act] op P2 [act]**
`lexeme_d[P]`	Turns off whitespace skipping
`limit_d[P](min, max)`	Matches only if the value is within the range
`longest_d[P]`	Matches the longest of alternatives
`max_limit_d[P](max)`	Matches only if **value <= max**
`min_limit_d[P](min)`	Matches only if **value >= min**
`refactor_action_d[P1 [act] op P2]`	Transforms to **(P1 op P2) [act]**
`refactor_unary_d[op1 P1 op2 P2]`	Transforms to **op1 (P1 op2 P2)**
`scoped_lock_d[P](mutex)`	Locks a mutex while matching
`shortest_d[P]`	Matches the shortest of alternatives

Tree-specific directives

`access_node_d[P]`	Passes node value to action

`hex_p`	Matches an unsigned hexadecimal integer	*numeric*
`int_p`	Matches a signed decimal integer	*numeric*
`int_parser<type, base, min, max>`	Matches a signed integer with **min** to **max** digits	*numeric*
`oct_p`	Matches an unsigned octal integer	*numeric*
`real_p`	Matches a floating point number	*numeric*
`real_parser<type, policy>`	Matches a floating point number	*numeric*
`strict_real_p`	Matches a floating point number (requires decimal point)	*numeric*
`strict_ureal_p`	Matches an unsigned FP number (requires decimal point)	*numeric*
`uint_p`	Matches an unsigned decimal integer	*numeric*
`uint_parser<type, base, min, max>`	Matches an unsigned integer with **min** to **max** digits	*numeric*
`ureal_p`	Matches an unsigned FP number	*numeric*

Other lexeme parsers

`c_escape_ch_p`	Matches a C escape code	*char*

`discard_first_node_d[P]`	Discards first node
`discard_last_node_d[P]`	Discards last node
`discard_node_d[P]`	Discards the generated node
`infix_node_d[P]`	Discards even-position nodes
`inner_node_d[P]`	Discards first and last nodes
`leaf_node_d[P]`	Generates a single node with no children
`no_node_d[P]`	Does not generate a node
`root_node_d[P]`	Identifies root nodes for an AST
`token_node_d[P]`	Synonym for **leaf_node_d**

Unary operators

`!P`	Matches **P** or an empty string
`*P`	Matches **P** zero or more times
`+P`	Matches **P** one or more times
`~P`	Matches anything that does not match **P**

Binary operators

`P1 % P2`	Matches one or more **P1** separated by **P2**
`P1 - P2`	Matches **P1** but not **P2**

`comment_p(string)` `comment_p(string1, string2)`	Matches C++ or C-style comments	*iter,iter*
`eol_p`	Matches CR, LF, or any combination	*iter,iter*
`f_str_p(func1, func2)`	Matches a string	*iter,iter*
`lex_escape_ch_p`	Matches a C escape code or any backslash escape	*char*
`regex_p(regex)`	Matches a regular expression	*iter,iter*
`str_p(string)` `str_p(iter1, iter2)`	Matches a string	*iter,iter*

Text parsers

`chseq_p(string)` `chseq_p(iter1, iter2)`	Matches a string, possibly with embedded whitespace	*iter,iter*
`f_chseq_p(func1, func2)`	Matches a string, possibly with embedded whitespace	*iter,iter*

`P1 >> P2`	Matches **P1** followed by **P2**
`P1 & P2`	Matches both **P1** and **P2**
`P1 ^ P2`	Matches **P1** or **P2**, but not both
`P1 | P2`	Matches **P1** or **P2**
`P1 && P2`	Synonym for **P1 >> P2**
`P1 || P2`	Matches **P1 | P2 | P1 >> P2**

Includes

Modules

Spirit is designed to be header only. Generally, there are no libraries to build and link against. Certain features, however, require additional libraries; in particular the **regular expression parser** requires **Boost.Regex** and **multithreading support** requires **Boost.Threads.**

Using Spirit is as easy as including the main header file:

```
    #include <boost/spirit.hpp>
```

Doing so will include all the header files. This might not be desirable. A low cholesterol alternative is to include only the module that you need. Each of the modules has its own header file. The master spirit header file actually includes all the module files. To avoid unnecessary inclusion of features that you do not need, it is better to include only the modules that you need.

```
    #include <boost/spirit/actor.hpp>
    #include <boost/spirit/attribute.hpp>
    #include <boost/spirit/core.hpp>
    #include <boost/spirit/debug.hpp>
    #include <boost/spirit/dynamic.hpp>
    #include <boost/spirit/error_handling.hpp>
    #include <boost/spirit/iterator.hpp>
    #include <boost/spirit/meta.hpp>
    #include <boost/spirit/symbols.hpp>
    #include <boost/spirit/tree.hpp>
    #include <boost/spirit/utility.hpp>
```

Sub-Modules

For even finer control over header file inclusion, you can include only the specific files that you need. Each module is in its own sub-directory:

actor

```
    #include <boost/spirit/actor/assign_actor.hpp>
    #include <boost/spirit/actor/assign_key.hpp>
    #include <boost/spirit/actor/clear_actor.hpp>
    #include <boost/spirit/actor/decrement_actor.hpp>
    #include <boost/spirit/actor/erase_actor.hpp>
    #include <boost/spirit/actor/increment_actor.hpp>
    #include <boost/spirit/actor/insert_key_actor.hpp>
    #include <boost/spirit/actor/push_back_actor.hpp>
    #include <boost/spirit/actor/push_front_actor.hpp>
    #include <boost/spirit/actor/swap_actor.hpp>
```

attribute

```
    #include <boost/spirit/attribute/closure.hpp>
    #include <boost/spirit/attribute/closure_context.hpp>
    #include <boost/spirit/attribute/parametric.hpp>
```

debug

⚠ The debug module should not be directly included. See **Debugging** for more info on how to use Spirit's debugger.

dynamic

```
    #include <boost/spirit/dynamic/for.hpp>
    #include <boost/spirit/dynamic/if.hpp>
    #include <boost/spirit/dynamic/lazy.hpp>
    #include <boost/spirit/dynamic/rule_alias.hpp>
    #include <boost/spirit/dynamic/select.hpp>
    #include <boost/spirit/dynamic/stored_rule.hpp>
    #include <boost/spirit/dynamic/switch.hpp>
    #include <boost/spirit/dynamic/while.hpp>
```

error_handling

```
    #include <boost/spirit/error_handling/exceptions.hpp>
```

iterator

```
    #include <boost/spirit/iterator/file_iterator.hpp>
    #include <boost/spirit/iterator/fixed_size_queue.hpp>
    #include <boost/spirit/iterator/multi_pass.hpp>
    #include <boost/spirit/iterator/position_iterator.hpp>
```

meta

```
    #include <boost/spirit/meta/as_parser.hpp>
    #include <boost/spirit/meta/fundamental.hpp>
    #include <boost/spirit/meta/parser_traits.hpp>
    #include <boost/spirit/meta/refactoring.hpp>
    #include <boost/spirit/meta/traverse.hpp>
```

tree

```
    #include <boost/spirit/tree/ast.hpp>
    #include <boost/spirit/tree/parse_tree.hpp>
    #include <boost/spirit/tree/parse_tree_utils.hpp>
    #include <boost/spirit/tree/tree_to_xml.hpp>
```

utility

```
    #include <boost/spirit/utility/chset.hpp>
    #include <boost/spirit/utility/chset_operators.hpp>
    #include <boost/spirit/utility/confix.hpp>
    #include <boost/spirit/utility/distinct.hpp>
    #include <boost/spirit/utility/escape_char.hpp>
    #include <boost/spirit/utility/flush_multi_pass.hpp>
    #include <boost/spirit/utility/functor_parser.hpp>
    #include <boost/spirit/utility/lists.hpp>
    #include <boost/spirit/utility/loops.hpp>
    #include <boost/spirit/utility/regex.hpp>
    #include <boost/spirit/utility/scoped_lock.hpp>
```

Portability

Historically, Spirit supported a lot of compilers, including (to some extent) poorly conforming compilers such as VC6. Spirit v1.6.x will be the last release that will support older poorly conforming compilers. Starting from Spirit v1.8.0, ill conforming compilers will not be supported. If you are still using one of these older compilers, you can still use Spirit v1.6.x.

The reason why Spirit v1.6.x worked on old non-conforming compilers is that the authors laboriously took the trouble of searching for workarounds to make these compilers happy. The process takes a lot of time and energy, especially when one encounters the dreaded ICE or "Internal Compiler Error". Sometimes searching for a single workaround takes days or even weeks. Sometimes, there are no known workarounds. This stifles progress a lot. And, as the library gets more progressive and takes on more advanced C++ techniques, the difficulty is escalated to even new heights.

Spirit v1.6.x will still be supported. Maintenance will still happen and bug fixes will still be applied. There will still be active development for the back-porting of new features introduced in Spirit v1.8.0 (and Spirit 1.9.0) to lesser able compilers; hopefully, fueled by contributions from the community. We welcome active support from the C++ community, especially those with special expertise on compilers such as older Borland and MSVC++ compilers.

Spirit 1.8 has been tested to compile and run properly on these compilers:

1. g++ 3.1 and above
2. Comeau 4.24.5
3. MSVC 7.1
4. Intel 7.1

If your compiler is sufficiently conforming, chances are, you can compile Spirit as it is or with minimal portability fixes here and there. Please inform us if your compiler is known to be ISO/ANSI conforming and is not in this list above. Feel free to post feedback to **Spirit-general mailing list** [Spirit-general@lists.sourceforge.net].

Style Guide

At some point, especially when there are lots of semantic actions attached to various points, the grammar tends to be quite difficult to follow. In order to keep an easy-to-read, consistent en aesthetically pleasing look to the Spirit code, the following coding styleguide is advised.

This coding style is adapted and extended from the ANTLR/PCCTS style (Terrence Parr) and **Boost coding guidelines** (David Abrahams and Nathan Myers) and is the combined work of Joel de Guzman, Chris Uzdavinis and Hartmut Kaiser.

- Rule names use std C++ (Boost) convention. The rule name may be very long.
- The '=' is neatly indented 4 spaces below. Like Boost, use spaces instead of tabs.
- Breaking the operands into separate lines puts the semantic actions neatly to the right.
- Semicolon at the last line terminates the rule.
- The adjacent parts of a sequence should be indented accordingly to have all, what belongs to one level, at one indentation level.

```
    program
        =   program_heading [heading_action]
            >> block [block_action]
            >> '.'
        |   another_sequence
            >> etc
        ;
```

- Prefer literals in the grammar instead of identifiers. e.g. `"program"` instead of `PROGRAM`, `'>='` instead of `GTE` and `'.'` instead of `DOT`. This makes it much easier to read. If this isn't possible (for instance where the used tokens must be identified through integers) capitalized identifiers should be used instead.
- Breaking the operands may not be needed for short expressions. e.g. `*(',' >> file_identifier)` as long as the line does not exceed 80 characters.
- If a sequence fits on one line, put spaces inside the parentheses to clearly separate them from the rules.

```
    program_heading
        =   as_lower_d["program"]
            >> identifier
            >> '('
            >> file_identifier
            >> *( ',' >> file_identifier )
            >> ')'
            >> ';'
        ;
```

- Nesting directives: If a rule does not fit on one line (80 characters) it should be continued on the next line intended by one level.
- The brackets of directives, semantic expressions (using Phoenix or LL lambda expressions) or parsers should be placed as follows.

```
    identifier
        =   nocase
            [
                lexeme
                [
                    alpha >> *(alnum | '_') [id_action]
                ]
            ]
        ;
```

- Nesting unary operators (e.g.Kleene star)
- Unary rule operators (Kleene star, `'!'`, `'+'` etc.) should be moved out one space before the corresponding indentation level, if this rule has a body or a sequence after it, which does not fit on on line. This makes the formatting more consistent and moves the rule 'body' at the same indentation level as the rule itself, highlighting the unary operator.

```
    block
        =  *(   label_declaration_part
            |   constant_definition_part
            |   type_definition_part
            |   variable_declaration_part
            |   procedure_and_function_declaration_part
            )
            >> statement_part
        ;
```

Techniques

Templatized Functors

For the sake of genericity, it is often better to make the functor's member `operator()` a template. That way, we do not have to concern ourselves with the type of the argument to expect as long as the behavior is appropriate. For instance, rather than hard-coding `char const*` as the argument of a generic semantic action, it is better to make it a template member function. That way, it can accept any type of iterator:

```
    struct my_functor
    {
        template <typename IteratorT>
        void operator()(IteratorT first, IteratorT last) const;
    };
```

Take note that this is only possible with functors. It is not possible to pass in template functions as semantic actions unless you cast it to the correct function signature; in which case, you *monomorphize* the function. This clearly shows that functors are superior to plain functions.

Rule With Multiple Scanners

As of v1.8.0, rules can use one or more scanner types. There are cases, for instance, where we need a rule that can work on the phrase and character levels. Rule/scanner mismatch has been a source of confusion and is the no. 1 **FAQ**. To address this issue, we now have **multiple scanner support**.

Here is an example of a grammar with a rule `r` that can be called with 3 types of scanners (phrase-level, lexeme, and lower-case). See the **rule**, **grammar**, **lexeme_scanner** and **as_lower_scanner** for more information.

Here's the grammar (see **multiple_scanners.cpp**):

```
    struct my_grammar : grammar<my_grammar>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(my_grammar const& self)
            {
                r = lower_p;
                rr = +(lexeme_d[r] >> as_lower_d[r] >> r);
            }

            typedef scanner_list<
                ScannerT
              , typename lexeme_scanner<ScannerT>::type
              , typename as_lower_scanner<ScannerT>::type
            > scanners;

            rule<scanners> r;
            rule<ScannerT> rr;
            rule<ScannerT> const& start() const { return rr; }
        };
    };
```

By default support for multiple scanners is disabled. The macro `BOOST_SPIRIT_RULE_SCANNERTYPE_LIMIT` must be defined to the maximum number of scanners allowed in a scanner_list. The value must be greater than 1 to enable multiple scanners. Given the example above, to define a limit of three scanners for the list, the following line must be inserted into the source file before the inclusion of Spirit headers:

```
    #define BOOST_SPIRIT_RULE_SCANNERTYPE_LIMIT 3
```

Look Ma' No Rules

You use grammars and you use lots of 'em? Want a fly-weight, no-cholesterol, super-optimized grammar? Read on...

I have a love-hate relationship with rules. I guess you know the reasons why. A lot of problems stem from the limitation of rules. Dynamic polymorphism and static polymorphism in C++ do not mix well. There is no notion of virtual template functions in C++; at least not just yet. Thus, the **rule is tied to a specific scanner type**. This results in problems such as the **scanner business**, our no. 1 FAQ. Apart from that, the virtual functions in rules slow down parsing, kill all meta-information, and kills inlining, hence bloating the generated code, especially for very tiny rules such as:

```
    r = ch_p('x') >> uint_p;
```

The rule's limitation is the main reason why the grammar is designed the way it is now, with a nested template definition class. The rule's limitation is also the reason why subrules exists. But do we really need rules? Of course! Before C++ adopts some sort of auto-type deduction, such as that proposed by David Abrahams in clc++m:

```
    auto r = ...definition ...
```

we are tied to the rule as RHS placeholders. However.... in some occasions we can get by without rules! For instance, rather than writing:

```
    rule<> x = ch_p('x');
```

It's better to write:

```
    chlit<> x = ch_p('x');
```

That's trivial. But what if the rule is rather complicated? Ok, let's proceed stepwise... I'll investigate a simple skip_parser based on the C grammar from Hartmut Kaiser. Basically, the grammar is written as (see **no_rule1.cpp**):

```
    struct skip_grammar : grammar<skip_grammar>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(skip_grammar const& /*self*/)
            {
                skip
                    =   space_p
                    |   "//" >> *(anychar_p - '\n') >> '\n'
                    |   "/*" >> *(anychar_p - "*/") >> "*/"
                    ;
            }

            rule<ScannerT> skip;

            rule<ScannerT> const&
            start() const { return skip; }
        };
    };
```

Ok, so far so good. Can we do better? Well... since there are no recursive rules there (in fact there's only one rule), you can expand the type of rule's RHS as the rule type (see **no_rule2.cpp**):

```
    struct skip_grammar : grammar<skip_grammar>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(skip_grammar const& /*self*/)
            : skip
                (       space_p
                    |   "//" >> *(anychar_p - '\n') >> '\n'
                    |   "/*" >> *(anychar_p - "*/") >> "*/"
                )
            {
            }

            typedef
               alternative<alternative<space_parser, sequence<sequence<
               strlit<const char*>, kleene_star<difference<anychar_parser,
               chlit<char> > > >, chlit<char> > >, sequence<sequence<
               strlit<const char*>, kleene_star<difference<anychar_parser,
               strlit<const char*> > > >, strlit<const char*> > >
            skip_t;
            skip_t skip;

            skip_t const&
            start() const { return skip; }
        };
    };
```

Ughhh! How did I do that? How was I able to get at the complex typedef? Am I insane? Well, not really... there's a trick! What you do is define the typedef `skip_t` first as int:

```
    typedef int skip_t;
```

Try to compile. Then, the compiler will generate an obnoxious error message such as:

```
    "cannot convert boost::spirit::alternative<... blah blah...to int".
```

THERE YOU GO! You got it's type! I just copy and paste the correct type (removing explicit qualifications, if preferred).

Can we still go further? Yes. Remember that the grammar was designed for rules. The nested template definition class is needed to get around the rule's limitations. Without rules, I propose a new class called `sub_grammar`, the grammar's low-fat counterpart:

```
    namespace boost { namespace spirit
    {
        template <typename DerivedT>
        struct sub_grammar : parser<DerivedT>
        {
            typedef sub_grammar self_t;
            typedef DerivedT const& embed_t;

            template <typename ScannerT>
            struct result
            {
                typedef typename parser_result<
                    typename DerivedT::start_t, ScannerT>::type
                type;
            };

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

            template <typename ScannerT>
            typename parser_result<self_t, ScannerT>::type
            parse(ScannerT const& scan) const
            {
                return derived().start.parse(scan);
            }
        };
    }}
```

With the `sub_grammar` class, we can define our skipper grammar this way (see **no_rule3.cpp**):

```
    struct skip_grammar : sub_grammar<skip_grammar>
    {
        typedef
           alternative<alternative<space_parser, sequence<sequence<
           strlit<const char*>, kleene_star<difference<anychar_parser,
           chlit<char> > > >, chlit<char> > >, sequence<sequence<
           strlit<const char*>, kleene_star<difference<anychar_parser,
           strlit<const char*> > > >, strlit<const char*> > >
        start_t;

        skip_grammar()
        : start
            (
                space_p
            |   "//" >> *(anychar_p - '\n') >> '\n'
            |   "/*" >> *(anychar_p - "*/") >> "*/"
            )
        {}

        start_t start;
    };
```

But what for, you ask? You can simply use the `start_t` type above as-is. It's already a parser! We can just type:

```
    skipper_t skipper =
        space_p
        |   "//" >> *(anychar_p - '\n') >> '\n'
        |   "/*" >> *(anychar_p - "*/") >> "*/"
        ;
```

and use `skipper` just as we would any parser? Well, a subtle difference is that `skipper`, used this way will be embedded **by value** when you compose more complex parsers using it. That is, if we use `skipper` inside another production, the whole thing will be stored in the composite. Heavy!

The proposed `sub_grammar` OTOH will be held by reference. Note:

```
    typedef DerivedT const& embed_t;
```

The proposed `sub_grammar` does not have the inherent limitations of rules, is very lighweight, and should be blazingly fast (can be fully inlined and does not use virtual functions). Perhaps this class will be part of a future spirit release.

The no-rules result

So, how much did we save? On MSVCV7.1, the original code: **no_rule1.cpp** compiles to **28k**. Eliding rules, **no_rule2.cpp**, we got **24k**. Not bad, we shaved off 4k amounting to a 14% reduction. But you'll be in for a surprise. The last version, using the sub-grammar: **no_rule3.cpp**, compiles to **5.5k**! That's a whopping 80% reduction.

no_rule1.cpp	**28k**	standard rule and grammar
no_rule2.cpp	**24k**	standard grammar, no rule
no_rule3.cpp	**5.5k**	sub_grammar, no rule, no grammar

typeof

Some compilers already support the `typeof` keyword. Examples are g++ and Metrowerks CodeWarrior. Someday, `typeof` will become commonplace. It is worth noting that we can use `typeof` to define non-recursive rules without using the rule class. To give an example, we'll use the skipper example above; this time using `typeof`. First, to avoid redundancy, we'll introduce a macro `RULE`:

```
    #define RULE(name, definition) typeof(definition) name = definition
```

Then, simply:

```
    RULE(
        skipper,
        (       space_p
            |   "//" >> *(anychar_p - '\n') >> '\n'
            |   "/*" >> *(anychar_p - "*/") >> "*/"
        )
    );
```

(see **typeof.cpp**)

That's it! Now you can use skipper just as you would any parser. Be reminded, however, that `skipper` above will be embedded by value when you compose more complex parsers using it (see `sub_grammar` rationale above). You can use the `sub_grammar` class to avoid this problem.

Nabialek trick

This technique, I'll call the ***"Nabialek trick"*** (from the name of its inventor, Sam Nabialek), can improve the rule dispatch from linear non-deterministic to deterministic. The trick applies to grammars where a keyword (operator, etc), precedes a production. There are lots of grammars similar to this:

```
    r =
            keyword1 >> production1
        |   keyword2 >> production2
        |   keyword3 >> production3
        |   keyword4 >> production4
        |   keyword5 >> production5
        /*** etc ***/
        ;
```

The cascaded alternatives are tried one at a time through trial and error until something matches. The Nabialek trick takes advantage of the **symbol table**'s search properties to optimize the dispatching of the alternatives. For an example, see **nabialek.cpp**. The grammar works as follows. There are two rules (`one` and `two`). When "one" is recognized, rule `one` is invoked. When "two" is recognized, rule `two` is invoked. Here's the grammar:

```
    one = name;
    two = name >> ',' >> name;

    continuations.add
        ("one", &one)
        ("two", &two)
    ;

    line = continuations[set_rest<rule_t>(rest)] >> rest;
```

where continuations is a **symbol table** with pointer to rule_t slots. one, two, name, line and rest are rules:

```
    rule_t name;
    rule_t line;
    rule_t rest;
    rule_t one;
    rule_t two;

    symbols<rule_t*> continuations;
```

set_rest, the semantic action attached to continuations is:

```
    template <typename Rule>
    struct set_rest
    {
        set_rest(Rule& the_rule)
        : the_rule(the_rule) {}

        void operator()(Rule* newRule) const
        { m_theRule = *newRule; }

        Rule& the_rule;
    };
```

Notice how the rest `rule` gets set dynamically when the set_rule action is called. The dynamic grammar parses inputs such as:

"one only"
"one again"
"two first, second"

The cool part is that the `rest` rule is set (by the `set_rest` action) depending on what the symbol table got. If it got a *"one"* then rest = one. If it got *"two"*, then rest = two. Very nifty! This technique should be very fast, especially when there are lots of keywords. It would be nice to add special facilities to make this easy to use. I imagine:

```
    r = keywords >> rest;
```

where `keywords` is a special parser (based on the symbol table) that automatically sets its RHS (rest) depending on the acquired symbol. This, I think, is mighty cool! Someday perhaps...

Also, see the **switch parser** for another deterministic parsing trick for character/token prefixes.

FAQ

The Scanner Business

Question: Why doesn't this compile?

```
    rule<> r = /*...*/;
    parse("hello world", r, space_p); // BAD [attempts phrase level parsing]
```

But if I remove the skip-parser, everything goes back to normal again:

```
    rule<> r = *anychar_p;
    parse("hello world", r); // OK [character level parsing]
```

Sometimes you'll want to pass in a rule to one of the functions parse functions that Spirit provides. The problem is that the rule is a template class that is parameterized by the scanner type. This is rather awkward but unavoidable: **the rule is tied to a scanner**. What's not obvious is that this scanner must be compatible with the scanner that is ultimately passed to the rule's parse member function. Otherwise, the compiler will complain.

Why does the first call to parse not compile? Because of scanner incompatibility. Behind the scenes, the free parse function creates a scanner from the iterators passed in. In the first call to parse, the scanner created is a plain vanilla `scanner<>`. This is compatible with the default scanner type of `rule<>` [see default template parameters of **the rule**]. The second call creates a scanner of type **`phrase_scanner_t`**. Thus, in order for the second call to succeed, the rule must be parameterized as `rule<phrase_scanner_t>`:

```
    rule<phrase_scanner_t> r = *anychar_p;
    parse("hello world", r, space_p);       //  OK [phrase level parsing]
```

Take note however that `phrase_scanner_t` is compatible only when you are using `char const*` iterators and `space_p` as the skip parser. Other than that, you'll have to find the right type of scanner. This is tedious to do correctly. In light of this issue, **it is best to avoid rules as arguments to the parse functions**. Keep in mind that this happens only with rules. The rule is the only parser that has to be tied to a particular scanner type. For instance:

```
    parse("hello world", *anychar_p);             //  OK  [character level parsing]
    parse("hello world", *anychar_p, space_p);  //  OK  [phrase level parsing]
```

Multiple Scanner Support

As of v1.8.0, rules can use one or more scanner types. There are cases, for instance, where we need a rule that can work on the phrase and character levels. Rule/scanner mismatch has been a source of confusion and is the no. 1 **FAQ**. To address this issue, we now have **multiple scanner support**.

See the techniques section for an **example** of a **grammar** using a multiple scanner enabled rule, **lexeme_scanner** and **as_lower_scanner.**

Eliminating Left Recursion

Question: I ported a grammar from YACC. It's "kinda" working - the parser itself compiles with no errors. But when I try to parse, it gives me an "invalid page fault". I tracked down the problem to this grammar snippet:

```
    or_expr = xor_expr | (or_expr >> VBAR >> xor_expr);
```

What you should do is to eliminate direct and indirect left-recursion. This causes the invalid page fault because the program enters an infinite loop. The code above is good for bottom up parsers such as YACC but not for LL parsers such as Spirit.

This is similar to a rule in Hartmut Kaiser's C parser (this should be available for download from **Spirit's site** as soon as you read this).

```
    inclusive_or_expression
    = exclusive_or_expression
    | inclusive_or_expression >> OR >> exclusive_or_expression
    ;
```

Transforming left recursion to right recursion, we have:

```
    inclusive_or_expression
    = exclusive_or_expression >> inclusive_or_expression_helper
    ;

    inclusive_or_expression_helper
    = OR >> exclusive_or_expression >> inclusive_or_expression_helper
    | epsilon_p
    ;
```

I'd go further. Since:

```
    r = a | epsilon_p;
```

is equivalent to:

```
    r = !a;
```

we can simplify `inclusive_or_expression_helper` thus:

```
    inclusive_or_expression_helper
    = !(OR >> exclusive_or_expression >> inclusive_or_expression_helper)
    ;
```

Now, since:

```
    r = !(a >> r);
```

is equivalent to:

```
    r = *a;
```

we have:

```
    inclusive_or_expression_helper
    = *(OR >> exclusive_or_expression)
    ;
```

Now simplifying `inclusive_or_expression` fully, we have:

```
    inclusive_or_expression
    = exclusive_or_expression >> *(OR >> exclusive_or_expression)
    ;
```

Reminds me of the calculators. So in short:

```
    a = b | a >> op >> b;
```

in pseudo-YACC is:

```
    a = b >> *(op >> b);
```

in Spirit. What could be simpler? Look Ma, no recursion, just iteration.

The lexeme_d directive and rules

Question: Does lexeme_d not support expressions which include rules? In the example below, the definition of atomicRule compiles,

```
    rule<phrase_scanner_t> atomicRule
        = lexeme_d[(alpha_p | '_') >> *(alnum_p | '.' | '-' | '_')];
```

but if I move `alnum_p | '.' | '-' | '_'` into its own rule, the compiler complains about conversion from `const scanner<...>` to `const phrase_scaner_t&`.

```
    rule<phrase_scanner_t> ch
        = alnum_p | '.' | '-' | '_';

    rule<phrase_scanner_t> compositeRule
        = lexeme_d[(alpha_p | '_') >> *(ch)]; // <- error source
```

You might get the impression that the `lexeme_d` directive and rules do not mix. Actually, this problem is related to the first FAQ entry: The Scanner Business. More precisely, the `lexeme_d` directive and rules with incompatible scanner types do not mix. This problem is more subtle. What's causing the scanner incompatibility is the directive itself. The `lexeme_d` directive transforms the scanner it receives into something that disables the skip parser. This non-skipping scanner, unfortunately, is incompatible with the original scanner before transformation took place.

The simplest solution is not to use rules in the `lexeme_d`. Instead, you can definitely apply `lexeme_d` to subrules and grammars if you really need more complex parsers inside the `lexeme_d`. If you really must use a rule, you need to know the exact scanner used by the directive. The `lexeme_scanner` metafunction is your friend here. The example above will work as expected once we give the `ch` rule a correct scanner type:

```
    rule<lexeme_scanner<phrase_scanner_t>::type> ch
```

```
        = alnum_p | '.' | '-' | '_';
```

Note: make sure to add "typename" before lexeme_scanner when this is used inside a template class or function.

The same thing happens when rules are used inside the as_lower_d directive. In such cases, you can use the as_lower_scanner. See the **lexeme_scanner** and **as_lower_scanner**.

> See the techniques section for an **example** of a **grammar** using a **multiple scanner enabled rule, lexeme_scanner** and **as_lower_scanner.**

Kleene Star infinite loop

Question: Why Does This Loop Forever?

```
    rule<> optional = !(str_p("optional"));
    rule<> list_of_optional = *optional;
```

The problem with this is that the kleene star will continue looping until it gets a no-match from it's enclosed parser. Because the optional rule is optional, it will always return a match. Even if the input doesn't match "optional" it will return a zero length match. list_of_optional will keep calling optional forever since optional will never return a no-match. So in general, any rule that can be "nullable" (meaning it can return a zero length match) must not be put inside a kleene star.

Boost CVS and Spirit CVS

Question: There is Boost CVS and Spirit CVS. Which is used for further development of Spirit?

Generally, development takes place in Spirit's CVS. However, from time to time a new version of Spirit will be integrated in Boost. When this happens development takes place in the Boost CVS. There will be announcements on the Spirit mailing lists whenever the status of the Spirit CVS changes.

> During development of Spirit v1.8.1 (released as part of boost-1.32.0) and v1.6.2, Spirit's developers decided to stop maintaining Spirit CVS for BRANCH_1_8 and BRANCH_1_6. This was necessary to reduce the added work of maintaining and synch'ing two repositories. The maintenance of these branches will take place on Boost CVS. At this time, new developments towards Spirit v2 and other experimental developments are expected to happen in Spirit CVS.

How to reduce compilation times with complex Spirit grammars

Question: Are there any techniques to minimize compile times using spirit? For simple parsers compile time doesn't seem to be a big issue, but recently I created a parser with about 78 rules and it took about 2 hours to compile. I would like to break the grammar up into smaller chunks, but it is not as easy as I thought it would be because rules in two grammar capsules are defined in terms of each other. Any thoughts?

The only way to reduce compile times is

- to split up your grammars into smaller chunks

- prevent the compiler from seeing all grammar definitions at the same time (in the same compilation unit)

The first task is merely logistical, the second is rather a technical one.

A good example of solving the first task is given in the Spirit cpp_lexer example written by JCAB (you may find it on the **applications' repository**).

The cross referencing problems may be solved by some kind of forward declaration, or, if this does not work, by introducing some dummy template argument to the non-templated grammars. Thus allows the instantiation time to be defered until the compiler has seen all the defintions:

```
    template <typename T = int>
    grammar2;



    template <typename T = int>
    struct grammar1 : public grammar<grammar1>
    {
    // refers to grammar2<>
    };

    template <typename T>
    struct grammar2 : public grammar<grammar2>
    {
    // refers to grammar1<>
    };

    //...
    grammar1<> g; // both grammars instantiated here
```

The second task is slightly more complex. You must ensure that in the first compilation unit the compiler sees only some function/template **declaration** and in the second compilation unit the function/template **definition**. Still no problem, if no templates are involved. If templates are involved, you need to manually (explicitly) instantiate these templates with the correct template parameters inside a separate compilation unit. This way the compilation time is split between several compilation units, reducing the overall required time drastically too.

For a sample, showing how to achieve this, you may want to look at the `Wave` preprocessor library, where this technique is used extensively. (this should be available for download from **Spirit's site** as soon as you read this).

Closure frame assertion

Question: When I run the parser I get an assertion *"frame.get() != 0 in file closures.hpp"*. What am I doing wrong?

Basically, the assertion fires when you are accessing a closure variable that is not constructed yet. Here's an example. We have three rules `a`, `b` and `c`. Consider that the rule `a` has a closure member `m`. Now:

```
    a = b;
    b = int_p[a.m = 123];
    c = b;
```

When the rule `a` is invoked, its frame is set, along with its member `m`. So, when `b` is called from `a`, the semantic action `[a.m = 123]`will store `123` into `a`'s closure member `m`. On the other hand, when `c` is invoked, and `c` attempts to call `b`, no frame for `a` is set. Thus, when `b` is called from `c`, the semantic action `[a.m = 123]`will fire the *"frame.get() != 0 in file closures.hpp"* assertion.

Greedy RD

Question: I'm wondering why the this won't work when parsed:

```
    a = +anychar_p;
    b = '(' >> a >> ')';
```

Try this:

```
    a = +(anychar_p - ')');
    b = '(' >> a >> ')';
```

David Held writes: That's because it's like the langoliers--it eats everything up. You usually want to say what it shouldn't eat up by subtracting the terminating character from the parser. The moral being: Using `*anychar_p` or `+anychar_p` all by itself is usually a *Bad Thing*™.

In other words: Recursive Descent is inherently greedy (however, see **Exhaustive backtracking and greedy RD**).

Referencing a rule at construction time

Question: The code below terminates with a segmentation fault, but I'm (obviously) confused about what I'm doing wrong.

```
    rule<ScannerT, clos::context_t> id = int_p[id.i = arg1];
```

You have a rule `id` being constructed. Before it is constructed, you reference `id.i` in the RHS of the constructor. It's a chicken and egg thing. The closure member `id.i` is not yet constructed at that point. Using assignment will solve the problem. Try this instead:

```
    rule<ScannerT, clos::context_t> id;
    id = int_p[id.i = arg1];
```

Storing Rules

Question: Why can't I store rules in STL containers for later use and why can't I pass and return rules to and from functions by value?

EBNF is primarily declarative. Like in functional programming, It's a static recipe and there's no notion of do this then that. However, in Spirit, we managed to coax imperative C++ to take in declarative EBNF. Hah! Fun!... We did that by masquerading the C++ assignment operator to mimic EBNF's `::=`, among other things (e.g. `>>`, `|`, `&` etc.). We used the rule class to let us do that by giving its assignment operator (and copy constructor) a different meaning and semantics. Doing so made the rule unlike any other C++ object. You can't copy it. You can't assign it. You can't place it in a container (vector, stack, etc).Heck, you can't even return it from a function *by value*.

⚠ The rule is a weird object, unlike any other C++ object. It does not have the proper copy and assignment semantics and cannot be stored and passed around by value.

However nice declarative EBNF is, the dynamic nature of C++ can be an advantage. We've seen this in action here and there. There are indeed some interesting applications of dynamic parsers using Spirit. Yet, we haven't fully utilized the power of dynamic parsing, unless(!), we have a rule that's not so alien to C++ (i.e. behaves as a good C++ object). With such a beast, we can write parsers that's defined at run time, as opposed to at compile time.

Now that I started focusing on rules (hey, check out the hunky new rule features), it might be a good time to implement the rule-holder. It is basically just a rule, but with C++ object semantics. Yet it's not as simple. Without true garbage collection, the implementation will be a bit tricky. We can't simply use reference counting because a rule-holder (hey, anyone here has a better name?) *is-a* rule, and rules are typically recursive and thus cyclic. The problem is which will own which.

Ok... this will do for now. You'll definitely see more of the rule-holder in the coming days.

Parsing Ints and Reals

Question: I was trying to parse an int or float value with the `longest_d` directive and put some actors on the alternatives to visualize the results. When I parse "123.456", the output reports:

1. (int) has been matched: full match = false
2. (double) has been matched: full match = true

That is not what I expected. What am I missing?

Actually, the problem is that both semantic actions of the int and real branch will be triggered because both branches will be tried. This doesn't buy us much. What actually wins in the end is what you expected. But there's no easy way to know which one wins. The problem stems from the ambiguity.

> Case1: Consider this input: "2". Is it an int or a real? They are both (strictly following the grammar of a real).
>
> Case2 : Now how about "1.0"? Is it an int or a real? They are both, albeit the int part gets a partial match: "1". That is why you are getting a (partial) match for your *int* rule (full match = false).

Instead of using the `longest_d` to parse ints and reals, what I suggest is to remove the ambiguity and use the plain short-circuiting alternatives. The first step is to use **`strict_real_p`** to make the first case unambiguous. Unlike `real_p`, `strict_real_p` requires a dot to be present for a number to be considered a successful match. Your grammar can be written unambiguously as:

```
    strict_real_p | int_p
```

Note that because ambiguity is resolved, attaching actions to both branches is safe. Only one will be triggered:

```
    strict_real_p[R] | int_p[I]
```

> "1.0" ---> triggers R
> "2" ---> triggers I

Again, as a rule of thumb, it is always best to resolve as much ambiguity as possible. The best grammars are those which involve no backtracking at all: an LL(1) grammar. Backtracking and semantic actions do not mix well.

BOOST_SPIRIT_DEBUG and missing `operator<<`

Question: My code compiles fine in release mode but when I try to define `BOOST_SPIRIT_DEBUG` the compiler complains about a missing `operator<<`.

When `BOOST_SPIRIT_DEBUG` is defined debug output is generated for spirit parsers. To this end it is expected that each closure member has the default output operator defined.

You may provide the operator overload either in the namespace where the class is declared (will be found through Argument Dependent Lookup) or make it visible where it is used, that is `namespace boost::spirit`. Here's an example for `std::pair`:

```
    #include <iosfwd>
    #include <utility>

    namespace std {

        template <
            typename C,
            typename E,
            typename T1,
            typename T2
        >
        basic_ostream<C, E> & operator<<(
            basic_ostream<C, E> & out,
            pair<T1, T2> const & what)
        {
            return out << '(' << what.first << ", "
                << what.second << ')';
        }

    }
```

Applications that used to be part of spirit

Question: Where can I find <*insert great application*>, that used to be part of the Spirit distribution?

Old versions of Spirit used to include applications built with it. In order to streamline the distribution they were moved to a separate **applications repository**. In that page you'll find links to full applications that use the Spirit parser framework. We encourage you to send in your own applications for inclusion (see the page for instructions).

You may also check out the **grammars' repository**.

You'll still find the example applications that complement (actually are part of) the documentation in the usual place: **libs/spirit/example**.

⚠ The applications and grammars listed in the repositories are works of the respective authors. It is the author's responsibility to provide support and maintenance. Should you have any questions, please send the author an email.

Rationale

Virtual functions: From static to dynamic C++

Rules straddle the border between static and dynamic C++. In effect, a rule transforms compile-time polymorphism (using templates) into run-time polymorphism (using virtual functions). This is necessary due to C++'s inability to automatically declare a variable of a type deduced from an arbitrarily complex expression in the right-hand side (rhs) of an assignment. Basically, we want to do something like:

```
    T rule = an_arbitrarily_complex_expression;
```

without having to know or care about the resulting type of the right-hand side (rhs) of the assignment expression. Apart from this, we also need a facility to forward declare an unknown type:

```
    T rule;
    ...
    rule = a | b;
```

These limitations lead us to this implementation of rules. This comes at the expense of the overhead of a virtual function call, once through each invocation of a rule.

Multiple declaration

Some BNF variants allow multiple declarations of a `rule`. The declarations are taken as alternatives. Example:

```
   r = a;
   r = b;
```

is equivalent to:

```
   r = a | b;
```

Spirit v1.3 allowed this behavior. However, the current version of Spirit **no longer** allows this because experience shows that this behavior leads to unwanted gotchas (for instance, it does not allow rules to be held in containers). In the current release of Spirit, a second assignment to a rule will simply redefine it. The old definition is destructed. This follows more closely C++ semantics and is more in line with what the user expects the rule to behave.

Sequencing Syntax

The comma operator as in a, b seems to be a better candidate, syntax-wise. But then the problem is with its precedence. It has the lowest precedence in C/C++, which makes it virtually useless.

Bjarne Stroustrup, in his article **"Generalizing Overloading for C++2000"** talks about overloading whitespace. Such a feature would allow juxtapositioning of parser objects exactly as we do in (E)BNF (e.g. a b | c instead of a >> b | c). Unfortunately, the article was dated April 1, 1998. Oh well.

Forward iterators

In general, the scanner expects at least a standard conforming forward iterator. Forward iterators are needed for backtracking where the iterator needs to be saved and restored later. Generally speaking, Spirit is a backtracking parser. The implication of this is that at some point, the iterator position will have to be saved to allow the parser to backtrack to a previous point. Thus, for backtracking to work, the framework requires at least a forward iterator.

Some parsers might require more specialized iterators (bi-directional or even random access). Perhaps in the future, deterministic parsers when added to the framework, will perform no backtracking and will need just a single token lookahead, hence will require input iterators only.

Why are subrules important?

Subrules open up the oportunity to do aggressive meta programming as well because they do not rely on virtual functions. The virtual function is the meta-programmer's hell. Not only does it slow down the program due to the virtual function indirect call, it is also an opaque wall where no metaprogram can get past. It kills all meta-information beyond the virtual function call. Worse, the virtual function cannot be templated. Which means that its arguments have to be tied to a actual types. Many problems stem from this limitation.

While Spirit is a currently classified as a non-deterministic recursive-descent parser, Doug Gregor first noted that other parsing techniques apart from top-down recursive descent may be applied. For instance, apart from non-deterministic recursive descent, deterministic LL(1) and LR(1) can theoretically be implemented using the same expression template front end. Spirit rules use virtual functions to encode the RHS parser expression in an opaque abstract parser type. While it serves its purpose well, the rule's virtual functions are the stumbling blocks to more advanced metaprogramming. Subrules are free from virtual functions.

Exhaustive backtracking and greedy RD

Spirit doesn't do exhaustive backtracking like regular expressions are expected to. For example:

```
    *chlit_p('a') >> chlit_p('a');
```

will always fail to match because Spirit's Kleene star does not back off when the rest of the rule fails to match.

Actually, there's a solution to this greedy RD problem. Such a scheme is discussed in section 6.6.2 of **Parsing Techniques: A Practical Guide**. The trick involves passing a *tail* parser (in addition to the scanner) to each parser. The start parser will then simply be: `start >> end_p;` (end_p is the start's tail).

Spirit is greedy --using straight forward, naive RD. It is certainly possible to implement the fully backtracking scheme presented above, but there will be also certainly be a performance hit. The scheme will always try to match all possible parser paths (full parser hierarchy traversal) until it reaches a point of certainty, that the whole thing matches or fails to match.

Backtracking and Greedy RD

Spirit is quite consistent and intuitive about when it backtracks and to where, although it may not be obvious to those coming from different backgrounds. In general, any (sub)parser will, given the same input, always match the same portion of the input (or fail to match the input at all). This means that Spirit is inherently greedy. Spirit will only backtrack when a (sub)parser fails to match the input, and it will always backtrack to the next choice point upward (not backward) in the parser structure. In other words abb|ab will match "ab", as will a(bb|b), but (ab|a)b won't because the (ab|a) subparser will always match the 'b' after the 'a' if it is available.

--Rainer Deyke

There's a strong preference on "simplicity with all the knobs when you need them" approach, right now. On the other hand, the flexibility of Spirit makes it possible to have different optional schemes available. It might be possible to implement an exhaustive backtracking RD scheme as an optional feature in the future.

Acknowledgments

Special thanks to

Dan Nuffer for his work on lexers, parse trees, ASTs, XML parsers, the multi-pass iterator as well as administering Spirit's site, editing, maintaining the CVS and doing the releases plus a zillion of other chores that were almost taken for granted.

Hartmut Kaiser for his work on the C parser, the work on the C/C++ preprocessor, utility parsers, the original port to Intel 5.0, various work on Phoenix, porting to v1.5, the meta-parsers, the grouping-parsers, extensive testing and painstaking attention to details.

Martin Wille who improved grammar multi thread safety, contributed the eol_p parser, the dynamic parsers, documentation and for taking an active role in almost every aspect from brainstorming and design to coding. And, as always, helps keep the regression tests for g++ on Linux as green as ever :-).

Martijn W. Van Der Lee our Web site administrator and for contributing the RFC821 parser**.**

Giovanni Bajo for last minute tweaks of Spirit 1.8.0 for CodeWarrior 8.3. Actually, I'm ashamed Giovanni was not in this list already. He's done a lot since Spirit 1.5, the first Boost.Spirit release. He's instrumental in the porting of the Spirit iterators stuff to the new Boost Iterators Library (version 2). He also did various bug fixes and wrote some tests here and there.

Juan Carlos Arevalo-Baeza (JCAB) for his work on the C++ parser, the position iterator, ports to v1.5 and keeping the mailing list discussions alive and kicking.

Vaclav Vesely, lots of stuff, the no_actions directive, various patches fixes, the distinct parsers, the lazy parser, some phoenix tweaks and add-ons (e.g. `new_`). Also, **Stefan Slapeta** and **wife** for editing Vaclav's distinct parser doc.

Raghavendra Satish for doing the original v1.3 port to VC++ and his work on Phoenix.

Noah Stein for following up and helping Ragav on the VC++ ports.

Hakki Dogusan, for his original v1.0 Pascal parser.

John (EBo) David for his work on the VM and watching over my shoulder as I code giving the impression of distance eXtreme programming.

Chris Uzdavinis for feeding in comments and valuable suggestions as well as editing the documentation.

Carsten Stoll, for his work on dynamic parsers.

Andy Elvey and his conifer parser.

Bruce Florman, who did the original v1.0 port to VC++.

Jeff Westfahl for porting the loop parsers to v1.5 and contributing the file iterator.

Peter Simons for the RFC date parser example and tutorial plus helping out with some nitty gritty details.

Markus Schöpflin for suggesting the end_p parser and lots of other nifty things and his active presence in the mailing list.

Doug Gregor for mentoring and his ability to see things that others don't.

David Abrahams for giving me a job that allows me to still work on Spirit, plus countless advice and help on C++ and specifically template metaprogramming.

Aleksey Gurtovoy for his MPL library from which I stole many metaprogramming tricks especially for less conforming compilers such as Borland and VC6/7.

Gustavo Guerra for his last minute review of Spirit and constant feedback, plus patches here and there (e.g. proposing the new dot behavior of the real numerics parsers).

Nicola Musatti, Paul Snively, Alisdair Meredith and **Hugo Duncan** for testing and sending in various patches.

Steve Rowe for his splendid work on the TSTs that will soon be taken into Spirit.

Jonathan de Halleux for his work on actors.

Angus Leeming for last minute editing work on the 1.8.0 release documentation, his work on Phoenix and his active presence in the Spirit mailing list.

Joao Abecasis for his active presence in the Spirit mailing list, providing user support, participating in the discussions and so on.

Guillaume Melquiond for a last minute patch to `multi_pass` for 1.8.1.

Peder Holt for his porting work on Phoenix, Fusion and Spirit to VC6.

To my wife **Mariel** who did the graphics in this document.

My, there's a lot in this list! And it's a continuing list. I add people to this list everytime. I hope I did not forget anyone. If I missed
someone you know who has helped in any way, please inform me.

Special thanks also to people who gave feedback and valuable comments, particularly members of Boost and Spirit mailing lists. This includes all those who participated in the review:

John Maddock, our review manager
Aleksey Gurtovoy
Andre Hentz
Beman Dawes
Carl Daniel
Christopher Currie
Dan Gohman

Dan Nuffer
Daryle Walker
David Abrahams
David B. Held
Dirk Gerrits
Douglas Gregor
Hartmut Kaiser
Iain K.Hanson
Juan Carlos Arevalo-Baeza
Larry Evans
Martin Wille
Mattias Flodin
Noah Stein
Nuno Lucas
Peter Dimov
Peter Simons
Petr Kocmid
Ross Smith
Scott Kirkwood
Steve Cleary
Thorsten Ottosen
Tom Wenisch
Vladimir Prus

Finally thanks to **SourceForge** for hosting the Spirit project and **Boost**: a C++ community comprised of extremely talented library authors who participate in the discussion and peer review of well crafted C++ libraries.

References

1. Todd Veldhuizen — "**Expression Templates**". C++ Report, June 1995.
2. Peter Naur (ed.) — "**Report on the Algorithmic Language ALGOL 60**". CACM, May 1960.
3. ISO/IEC — "**ISO-EBNF**", ISO/IEC 14977: 1996(E).
4. Richard J. Botting, Ph.D. — "**XBNF**" (citing Leu-Weiner, 1973). California State University, San Bernardino, 1998.
5. James Coplien. — "**Curiously Recurring Template Pattern**". C++ Report, Feb. 1995.
6. Thierry Géraud and Alexandre Duret-Lutz — **Generic Programming Redesign of Patterns** Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000) Irsee, Germany, July 2000.
7. Geoffrey Furnish — **"Disambiguated Glommable Expression Templates Reintroduced"** C++ Report, May 2000
8. Erich Gamma, Richard Helm, Ralph Jhonson, and John Vlissides — **Design Patterns, Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.
9. Alfred V. Aho Revi Sethi Feffrey D. Ulman — **Compilers, Principles, Techniques and Tools** Addison-Wesley, June 1987.
10. Dick Grune and Ceriel Jacobs — **Parsing Techniques: A Practical Guide**. Ellis Horwood Ltd.: West Sussex, England, 1990. (electronic copy, 1998).
11. T. J. Parr, H. G. Dietz, and W. E. Cohen — **PCCTS Reference Manual (Version 1.00)**. School of Electrical Engineering, Purdue University, West Lafayette, August 1991.
12. Adrian Johnstone and Elizabeth Scott. — **RDP, A Recursive Descent Compiler Compiler**. Technical Report CSD TR 97 25, Dept. of Computer Science, Egham, Surrey, England, Dec. 20, 1997.
13. Adrian Johnstone — **Languages and Architectures, Parser generators with backtrack or extended lookahead capability** Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, England
14. Damian Conway — **Parsing with C++ Classes.** ACM SIGPLAN Notices, 29:1, 1994.

15.	Joel de Guzman	**"Spirit Version 1.3"**. http://spirit.sourceforge.net/, November 2001.
16.	S. Doaitse Swierstra and Luc Duponcheel	**Deterministic, Error-Correcting Combinator Parsers** Dept. of Computer Science, Utrecht University P.O.Box 80.089, 3508 TB Utrecht, The Netherland
17.	Bjarne Stroustrup	**Generalizing Overloading for C++2000** Overload, Issue 25. April 1, 1998.
18.	Dr. John Maddock	**Regex++ Documentation** http://www.boost.org/libs/regex/index.htm
19.	Anonymous Edited by Graham Hutton	Frequently Asked Questions for comp.lang.functional. Edited by Graham Hutton, University of Nottingham. **http://www.cs.nott.ac.uk/~gmh//faq.html**
20.	Hewlett-Packard	Standard Template Library Programmer's Guide. **http://www.sgi.com/tech/stl/**, Hewlett-Packard Company, 1994
21.	boost.org	Boost Libraries Documentation. **http://www.boost.org/**
22.	Brian McNamara and Yannis Smaragdakis	FC++: Functional Programming in C++. **http://www.cc.gatech.edu/~yannis/fc++/**
23.	Todd Veldhuizen	**Techniques for Scientic C++.**