教程

01

1.1 术语

首先，让我们介绍一些将在文档中使用的术语：

*语义动作*是与解析器相关联的任意逻辑片段，仅在解析器匹配时执行。

简单的解析器可以组合形成更复杂的解析器。给定一些组合操作 `C`，以及解析器 `P0`、`P1`、...、`PN`，`C(P0, P1, ... PN)` 创建一个新的解析器 `Q`。这会创建一个 *解析树*。`Q` 是 `P1` 的父节点，`P2` 是 `Q` 的子节点，等等。解析器按照这种拓扑结构自上而下地应用。当你使用 `Q` 解析字符串时，它将使用 `P0`、`P1` 等来完成实际工作。如果 `P3` 正在解析输入，这意味着 `Q` 也在解析，因为 `Q` 的解析方式是通过派遣其子节点来完成部分或全部工作。在解析的任何时刻，只有一个没有子节点的解析器正在解析输入；所有其他正在使用的解析器都是解析树中的祖先节点。

*子解析器*是另一个解析器的子节点的解析器。

*顶级解析器*是解析器树的根节点。

*当前解析器*或 *最底层解析器*是当前用于解析输入的没有子节点的解析器。

*规则*是一种解析器，使构建大型复杂解析器变得更容易。*子规则*是某个其他规则的子节点的规则。*当前规则*或 *最底层规则*是当前用于解析输入且没有子规则的规则。请注意，虽然始终只有一个当前解析器，但可能有也可能没有当前规则——规则是一种解析器，你可能在解析的某个点上使用它，也可能不使用。

*顶级解析*是由顶级解析器执行的解析操作。这个术语是必要的，因为尽管大多数解析失败是特定解析器的局部失败，但有些解析失败会导致 `parse()` 调用指示整个解析失败。对于这些情况，我们说这种局部失败“导致顶级解析失败”。

在 Boost.Parser 文档中，我将提到“对 `parse()` 的调用”。将其理解为“对 `parse()` API 中描述的任何一个函数的调用”。这包括 `prefix_parse()`、`callback_parse()` 和 `callback_prefix_parse()`。

在本文档中经常出现一些特殊类型的解析器。

其中一个是 *序列解析器*；你会看到它使用 `operator>>` 创建，如 `p1 >> p2 >> p3`。序列解析器尝试按顺序将其所有子解析器匹配到输入；如果所有子解析器都匹配，则匹配输入。

另一个是 *选择解析器*；你会看到它使用 `operator|` 创建，如 `p1 | p2 | p3`。选择解析器尝试按顺序将其所有子解析器匹配到输入；它在最多匹配一个子解析器后停止。如果其中一个子解析器匹配，则匹配输入。

最后是 *排列解析器*；它使用 `operator||` 创建，如 `p1 || p2 || p3`。排列解析器尝试以任意顺序将其所有子解析器匹配到输入。因此，解析器 `p1 || p2 || p3` 等价于 `(p1 >> p2 >> p3) | (p1 >> p3 >> p2) | (p2 >> p1 >> p3) | (p2 >> p3 >> p1) | (p3 >> p1 >> p2) | (p3 >> p2 >> p1)`。希望它的简洁性是不言自明的。如果所有子解析器都匹配，则匹配输入，无论它们匹配的顺序如何。

Boost.Parser 的解析器每个都有一个 *属性*与之关联，或者明确没有属性。属性是解析器在匹配输入时生成的值。例如，解析器 `double_` 在匹配输入时生成一个 `double`。`ATTR()` 是一个概念上的宏，它扩展为传递给它的解析器的属性类型；`ATTR(double_)` 是 `double`。这类似于 `attribute` 类型特征。

接下来，我们将看一些使用 Boost.Parser 进行解析的简单程序。我们将从小处开始，然后逐步构建。

1.1.1 你好，任何人

这是使用 Boost.Parser 编写的最简约的示例之一。我们从命令行获取一个字符串，如果没有给定，则使用 `"World"`，然后我们解析它：

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


namespace bp = boost::parser;

int main(int argc, char const * argv[])
{
    std::string input = "World";
    if (1 < argc)
        input = argv[1];

    std::string result;
    bp::parse(input, *bp::char_, result);
    std::cout << "Hello, " << result << "!\n";
}
```

表达式 `*bp::char_` 是一个解析器表达式。它使用了 Boost.Parser 提供的众多解析器之一：`char_`。像所有 Boost.Parser 解析器一样，它有一些定义的操作。在这种情况下，`*bp::char_` 使用了重载的 `operator*` 作为 C++ 版本的 Kleene 星操作符。由于 C++ 没有后缀一元 `*` 操作符，我们必须使用现有的操作符，所以它被用作前缀。

因此，`*bp::char_` 的意思是“任意数量的字符”。换句话说，它实际上不会失败。即使是空字符串也会匹配。

解析操作是通过调用 `parse()` 函数来执行的，将解析器作为参数之一传递：

```
bp::parse(input, *bp::char_, result);
```

这里的参数是：`input`，要解析的范围；`*bp::char_`，用于解析的解析器；以及 `result`，一个用于存放解析结果的输出参数。不要过于纠结于这种从 `parse()` 获取解析结果的方法；有多种方法可以做到这一点，我们将在后续章节中介绍所有这些方法。

另外，暂时忽略 Boost.Parser 如何确定 `*bp::char_` 解析器的结果类型是 `std::string` 的事实。对此有明确

的规则，我们稍后会介绍。

这次调用 `parse()` 的效果并不太有趣——因为我们给它的解析器永远不会失败，并且因为我们将输出放在与输入相同的类型中，它只是将 `input` 的内容复制到 `result` 中。

1.1.2 一个简单的示例

让我们看一个稍微复杂一点的示例，即使它仍然很简单。我们不再接受任意的 `char`，而是要求一些结构。我们来解析一个或多个用逗号分隔的 `double`。

Boost.Parser 的 `double` 解析器是 `double_`。所以，要解析一个 `double`，我们只需使用它。如果我们想连续解析两个 `double`，我们会使用：

```
boost::parser::double_ >> boost::parser::double_
```

这个表达式中的 `operator>>` 是序列操作符；可以将其理解为“后跟”。如果我们将序列操作符与 Kleene 星结合起来，我们可以通过编写以下代码来获得我们想要的解析器：

```
boost::parser::double_ >> *(',' >> boost::parser::double_)
```

这是一个匹配至少一个 `double` 的解析器——因为上面表达式中的第一个 `double_`——后跟零个或多个逗号后跟一个 `double` 的实例。注意我们可以直接使用 `','`。尽管它不是解析器，`operator>>` 和定义在 Boost.Parser 解析器上的其他操作符有接受字符/解析器对参数的重载；这些操作符重载将创建正确的解析器来识别 `','`。

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


namespace bp = boost::parser;

int main()
{
    std::cout << "Enter a list of doubles, separated by commas.  No pressure. ";
    std::string input;
    std::getline(std::cin, input);

    auto const result = bp::parse(input, bp::double_ >> *(',' >> bp::double_));

    if (result) {
        std::cout << "Great! It looks like you entered:\n";
        for (double x : *result) {
            std::cout << x << "\n";
        }
    } else {
        std::cout
            << "Good job!  Please proceed to the recovery annex for cake.\n";
    }
}
```

第一个示例使用了一个输出参数来传递解析结果。这次对 `parse()` 的调用则返回一个结果。如你所见，结果可以上下文转换为 `bool`，并且 `*result` 是某种范围。实际上，这次对 `parse()` 的调用返回的类型是 `std::optional<std:`
自然地，如果解析失败，则返回 `std::nullopt`。稍后我们将看看 Boost.Parser 如何将解析器的类型映射到返回类型或填充的输出参数的类型。

Note

有一个类型特征可以告诉你解析器的属性类型，即 `attribute`（以及一个相关的别名 `attribute_t`）。
我们将在属性生成部分详细讨论它。

If I run it in a shell, this is the result:

```
$ example/trivial
Enter a list of doubles, separated by commas.  No pressure. 5.6,8.9
Great! It looks like you entered:
5.6
8.9
$ example/trivial
Enter a list of doubles, separated by commas.  No pressure. 5.6, 8.9
Good job!  Please proceed to the recovery annex for cake.
```

它无法识别 `"5.6, 8.9"`。这是因为它期望逗号后面立即跟一个 `double`，但我在逗号后插入了一个空格。如果我在逗号前、逗号前后或 `double` 列表前后放置空格，也会发生同样的解析失败。

还有一件事：有一种更好的方法来编写上面的解析器。我们可以不重复 `double_` 子解析器，而是这样写：

```
bp::double_ % ','
```

这在语义上等同于 `bp::double_ >> *(',' >> bp::double_)`。这种模式——一些输入重复一次或多次，每个实例之间有一个分隔符——非常常见，因此有一个专门的操作符 `operator%`。从现在开始我们将使用这个操作符。

1.1.3 一个优雅处理空白的简单示例

让我们修改刚才看到的简单解析器，以忽略在 `double` 和逗号之间可能出现的任何空格。为了在任何地方跳过空白字符，我们可以将一个 *跳过解析器* 传递给 `parse()` 调用（我们不需要修改传递给 `parse()` 的解析器）。在这里，我们使用 `ws`，它匹配任何 Unicode 空白字符。

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


namespace bp = boost::parser;

int main()
{
    std::cout << "Enter a list of doubles, separated by commas.  No pressure. ";
    std::string input;
```

```
    std::getline(std::cin, input);

    auto const result = bp::parse(input, bp::double_ % ',', bp::ws);

    if (result) {
        std::cout << "Great! It looks like you entered:\n";
        for (double x : *result) {
            std::cout << x << "\n";
        }
    } else {
        std::cout
            << "Good job!  Please proceed to the recovery annex for cake.\n";
    }
}
```

跳过解析器，或称 *skipper*，在传递给 `parse()` 的解析器中的子解析器之间运行。在这种情况下，skipper 在解析第一个 `double` 之前、在解析任何后续的逗号或 `double` 之前以及在结束时运行。因此，字符串 `"3.6,5.9"` 和 `" 3.6 , \t 5.9 "` 被这个程序解析的结果是相同的。

跳过是 Boost.Parser 中的一个重要概念。你可以跳过任何东西，不仅仅是空白字符；还有很多其他你可能想跳过的东西。传递给 `parse()` 的 skipper 可以是任意解析器。例如，如果你为脚本语言编写解析器，你可以编写一个 skipper 来跳过空白字符、内联注释和行尾注释。

在接下来的文档中，我们几乎会专门使用跳过解析器。忽略输入中你不关心的部分的能力是如此方便，以至于在实际操作中不使用跳过解析的情况很少见。

1.1.4 语义动作

像所有解析系统（lex & yacc、Boost.Spirit 等）一样，Boost.Parser 有一种机制可以将语义动作与解析的不同部分关联起来。这里是与我们在前一个示例中看到的几乎相同的程序，只是它是通过一个语义动作来实现的，该动作将每个解析的 `double` 附加到结果中，而不是自动构建和返回结果。为此，我们将前一个示例中的 `double_` 替换为 `double_[action]`；`action` 是我们的语义动作：

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


namespace bp = boost::parser;

int main()
{
    std::cout << "Enter a list of doubles, separated by commas. ";
    std::string input;
    std::getline(std::cin, input);

    std::vector<double> result;
    auto const action = [&result](auto & ctx) {
        std::cout << "Got one!\n";
```

```
        result.push_back(_attr(ctx));
    };
    auto const action_parser = bp::double_[action];
    auto const success = bp::parse(input, action_parser % ',', bp::ws);

    if (success) {
        std::cout << "You entered:\n";
        for (double x : result) {
            std::cout << x << "\n";
        }
    } else {
        std::cout << "Parse failure.\n";
    }
}
```

Run in a shell, it looks like this:

```
$ example/semantic_actions
Enter a list of doubles, separated by commas. 4,3
Got one!
Got one!
You entered:
4
3
```

在 Boost.Parser 中，语义动作是通过可调用对象实现的，这些对象接受一个解析上下文对象作为参数。解析上下文对象表示当前的解析状态。在示例中，我们使用了这个 lambda 作为我们的可调用对象：

```
auto const action = [&result](auto & ctx) {
    std::cout << "Got one!\n";
    result.push_back(_attr(ctx));
};
```

我们在 lambda 中同时打印一条消息到 `std::cout` 并记录解析结果。如果你愿意，它可以做这两件事中的任意一件或都不做。我们通过向解析上下文请求解析的 `double` 来在 lambda 中获取它。`_attr(ctx)` 是你向解析上下文请求解析器生成的属性的方式。还有很多类似 `_attr()` 的函数可以用来访问解析上下文中的状态。我们稍后会介绍更多这些函数。解析上下文定义了解析上下文的具体内容及其工作方式。

注意，你不能直接将一个未修饰的 lambda 写为语义动作。否则，编译器会看到两个 `'['` 字符并认为它即将解析一个属性。使用括号可以解决这个问题：

```
p[([](auto & ctx){/*...*/})]
```

在你这样做之前，请注意，你编写的作为语义动作的 lambda 函数几乎总是通用的（具有 `auto & ctx` 参数），因此非常频繁地可重用。大多数你编写的语义动作 lambda 函数应该写在行外，并给它们一个好的名字。即使它们不被重用，命名的 lambda 也能使你的解析器更小、更易读。

Important

Attaching a semantic action to a parser removes its attribute. That is, *ATTR(p[a])* is always the special no-attribute type none, regardless of what type *ATTR(p)* is.

规则中的语义动作

在 rules 中使用语义动作时，还有一些其他形式。详情请参见“更多关于规则”的部分。

1.1.5 解析以查找子范围

到目前为止，我们已经看到了解析一些文本并生成相关属性的示例。有时，你想找到输入中包含你要查找内容的某个子范围，而不想生成属性。

有两个 指令会影响任何解析器的属性类型，raw[] 和 string_view[]。（我们将在后面的指令部分详细介绍指令。现在，你只需要知道指令包装一个解析器，并改变其某些功能方面。）

raw[]

raw[] 将其解析器的属性更改为一个 subrange，其 begin() 和 end() 返回匹配 p 的序列的边界。

```
namespace bp = boost::parser;
auto int_parser = bp::int_ % ',';            // ATTR(int_parser) is std::vector<int>
auto subrange_parser = bp::raw[int_parser];  // ATTR(subrange_parser) is a subrange

// Parse using int_parser, generating integers.
auto ints = bp::parse("1, 2, 3, 4", int_parser, bp::ws);
assert(ints);
assert(*ints == std::vector<int>({1, 2, 3, 4}));

// Parse again using int_parser, but this time generating only the
// subrange matched by int_parser.  (prefix_parse() allows matches that
// don't consume the entire input.)
auto const str = std::string("1, 2, 3, 4, a, b, c");
auto first = str.begin();
auto range = bp::prefix_parse(first, str.end(), subrange_parser, bp::ws);
assert(range);
assert(range->begin() == str.begin());
assert(range->end() == str.begin() + 10);

static_assert(std::is_same_v<
              decltype(range),
              std::optional<bp::subrange<std::string::const_iterator>>>);
```

请注意，subrange 的迭代器类型是 std::string::const_iterator，因为这是传递给 prefix_parse() 的迭代器类型。如果我们传递 char const * 迭代器给 prefix_parse()，那将是迭代器类型。唯一的例外来自

支持 Unicode 的解析（参见 Unicode 支持）。在某些情况下，解析中使用的迭代器不是你传递的那个。例如，如果你用 `char8_t *` 迭代器调用 `prefix_parse()`，它将创建一个从 UTF-8 到 UTF-32 的转码视图，并解析该视图的迭代器。在这种情况下，你会得到一个迭代器类型为转码迭代器的 `subrange`。当这种情况发生时，你可以通过调用返回的 `subrange` 中每个转码迭代器的 `.base()` 成员函数来获取底层迭代器——即你传递给 `prefix_parse()` 的那个迭代器。

```
auto const u8str = std::u8string(u8"1, 2, 3, 4, a, b, c");
auto u8first = u8str.begin();
auto u8range = bp::prefix_parse(u8first, u8str.end(), subrange_parser, bp::ws);
assert(u8range);
assert(u8range->begin().base() == u8str.begin());
assert(u8range->end().base() == u8str.begin() + 10);
```

string_view[]

`string_view[]` 的语义与 `raw[]` 非常相似，不同之处在于它生成一个 `std::basic_string_view<CharT>`（其中 `CharT` 是正在解析的底层范围的类型），而不是 `subrange`。为了使其工作，底层范围必须是连续的。在 C++20 之前无法检测迭代器的连续性，因此此指令仅在 C++20 及更高版本中可用。

```
namespace bp = boost::parser;
auto int_parser = bp::int_ % ',';              // ATTR(int_parser) is std::vector<int>
auto sv_parser = bp::string_view[int_parser];  // ATTR(sv_parser) is a string_view

auto const str = std::string("1, 2, 3, 4, a, b, c");
auto first = str.begin();
auto sv1 = bp::prefix_parse(first, str.end(), sv_parser, bp::ws);
assert(sv1);
assert(*sv1 == str.substr(0, 10));

static_assert(std::is_same_v<decltype(sv1), std::optional<std::string_view>>);
```

由于 `string_view[]` 生成 `string_view`，它不能像上面描述的 `raw[]` 那样返回转码迭代器。如果你用 `string_view[]` 解析一个 `CharT` 序列，你会得到一个 `std::basic_string_view<CharT>`。如果解析使用了 Unicode 感知路径中的转码，`string_view[]` 将根据需要分解转码迭代器。如果你将转码视图传递给 `parse()` 或将转码迭代器传递给 `prefix_parse()`，`string_view[]` 仍然会透过转码迭代器并给你一个底层范围部分的 `string_view`。

```
auto sv2 = bp::parse("1, 2, 3, 4" | bp::as_utf32, sv_parser, bp::ws);
assert(sv2);
assert(*sv2 == "1, 2, 3, 4");

static_assert(std::is_same_v<decltype(sv2), std::optional<std::string_view>>);
```

1.1.6 解析上下文

现在是详细描述解析上下文的好时机。你编写的任何语义动作都需要使用解析上下文中的状态，所以你需要知道有哪些可用的内容。

解析上下文是一个存储当前解析状态的对象——当前迭代器和结束迭代器、错误处理程序等。数据可能会

在解析过程中“添加”到或“移除”出解析上下文。例如，当一个带有语义动作 `a` 的解析器 `p` 成功时，解析上下文会将 `p` 生成的属性添加到解析上下文中，然后调用 `a`，并将上下文传递给它。

尽管解析上下文对象似乎有东西被添加或移除，但实际上并没有一个单一的上下文对象。上下文是在解析过程中在不同时间点形成的，通常是在启动子解析器时。每个上下文都是通过获取前一个上下文并根据需要添加或更改成员来形成一个新的上下文对象。当包含新上下文对象的函数返回时，其上下文对象（如果有的话）会被析构。这种操作是高效的，因为解析上下文只有大约十二个数据成员，每个数据成员的大小都小于或等于一个指针的大小。因此，在修改上下文时复制整个上下文是快速的。解析上下文不进行内存分配。

Tip

所有这些以解析上下文作为第一个参数的函数都可以通过参数依赖查找（Argument-Dependent Lookup）找到。你可能永远不需要用 `boost::parser::` 来限定它们。

Accessors for data that are always available

By convention, the names of all Boost.Parser functions that take a parse context, and are therefore intended for use inside semantic actions, contain a leading underscore.

_pass()

`_pass()` returns a reference to a `bool` indicating the success of failure of the current parse. This can be used to force the current parse to pass or fail:

```
[](auto & ctx) {
    // If the attribute fails to meet this predicate, fail the parse.
    if (!necessary_condition(_attr(ctx)))
        _pass(ctx) = false;
}
```

请注意，要执行语义动作，其关联的解析器必须已经成功。因此，除非你之前在动作中写了 `_pass(ctx) = false`，否则 `_pass(ctx) = true` 没有任何作用；它是多余的。

_begin()、_end() 和 _where()

`_begin()` 和 `_end()` 分别返回传递给 `parse()` 的范围的开始和结束。`_where()` 返回一个 `subrange`，表示当前解析匹配的输入范围的边界。如果你只想解析一些文本并返回某些元素的位置而不生成任何其他属性，`_where()` 会很有用。`_where()` 也可以在跟踪元素位置时至关重要，以便在解析的后期提供良好的诊断。想想 XML 中的不匹配标签；如果你在元素的末尾解析一个关闭标签，并且它与打开标签不匹配，你会希望生成一条提到或显示两个标签的错误消息。将 `_where(ctx).begin()` 存储在关闭标签解析器可用的某个地方将实现这一点。有关示例，请参见错误处理和调试。

_error_handler()

`_error_handler()` 返回一个引用，指向与传递给 `parse()` 的解析器关联的错误处理程序。使用 `_error_handler()`，你可以从语义动作中生成错误和警告。有关具体示例，请参见错误处理和调试。

仅在某些情况下可用的数据访问器

_attr()

`_attr()` 返回对当前解析器属性值的引用。仅当当前解析器的解析成功时才可用。如果解析器没有语义动作，则不会将任何属性添加到解析上下文中。它可以用来读取和写入当前解析器的属性：

```
[](auto & ctx) { _attr(ctx) = 3; }
```

如果当前解析器没有属性，则返回 `none`。

_val()

`_val()` 返回对当前规则的属性值的引用（如果有），即使在规则的解析成功之前也可用。它可以用来设置当前规则的属性，即使是在规则内的子解析器中。假设我们正在编写一个带有语义动作的解析器，该动作在一个规则内。如果我们想将当前规则的值设置为子解析器属性的某个函数，我们会编写这个语义动作：

```
[](auto & ctx) { _val(ctx) = some_function(_attr(ctx)); }
```

如果没有当前规则，或者当前规则没有属性，则返回 `none`。

当 `rule` 的解析器的默认属性与 `rule` 的属性类型不直接兼容时，你需要使用 `_val()`。在这些情况下，你需要编写类似上面示例的代码，以从 `rule` 的解析器生成的属性计算 `rule` 的属性。有关 `rules` 的更多信息，请参见下一页和更多关于规则的内容。

_globals()

`_globals()` 返回对用户提供的对象的引用，该对象包含你希望在解析期间使用的任何数据。解析的“全局变量”是一个对象——通常是一个结构体——你提供给顶级解析器。然后你可以在解析期间随时使用 `_globals()` 访问它。我们将在后面的 `parse()` API 中看到全局变量如何与顶级解析器关联。例如，假设你在解析的早期部分需要记录一些黑名单值，而解析的后期部分可能需要解析值，如果看到黑名单值则解析失败。在解析的早期部分，你可以编写类似这样的代码。

```
[](auto & ctx) {
    // black_list is a std::unordered_set.
    _globals(ctx).black_list.insert(_attr(ctx));
}
```

Later in the parse, you could then use `black_list` to check values as they are parsed.

```
[](auto & ctx) {
    if (_globals(ctx).black_list.contains(_attr(ctx)))
        _pass(ctx) = false;
```

```
}
```

_locals()

`_locals()` 返回对当前正在解析的规则的一个或多个局部值的引用（如果有）。如果有两个或更多局部值，`_locals()` 返回对 `boost::parser::tuple` 的引用。我们还没有涉及带有局部变量的规则（参见更多关于规则的内容），但现在你只需要知道你可以为 `rule` 提供一个模板参数（`LocalState`），并且该规则将默认构造一个该类型的对象以在规则内使用。你可以通过 `_locals()` 访问它：

```
[](auto & ctx) {
    auto & local = _locals(ctx);
    // Use local here.  If 'local' is a hana::tuple, access its members like this:
    using namespace hana::literals;
    auto & first_element = local[0_c];
    auto & second_element = local[1_c];
}
```

如果没有当前规则，或者当前规则没有局部变量，则返回 `none`。

_params()

`_params()` 类似于 `_locals()`，适用于当前用于解析的规则（如果有）（参见更多关于规则的内容）。如果当前规则只有一个参数，它返回对单个值的引用；如果当前规则有多个参数，它返回一个 `boost::parser::tuple`。如果没有当前规则，或者当前规则没有参数，则返回 `none`。

与 `_locals()` 不同，你 不需要为 `rule` 提供模板参数。相反，你调用 `rule` 的 `with()` 成员函数（同样，参见更多关于规则的内容）。

Note

`none` 是一个类型，用作 Boost.Parser 中解析上下文访问器的返回值。`none` 可以转换为任何具有默认构造函数的类型，可以从任何类型转换而来，可以从任何类型赋值，并且对所有可重载的操作符都有模板化的重载。其目的是确保误用 `_val()`、`_globals()` 等操作在编译时不会报错，而是在运行时产生断言。经验表明，使用调试器调查导致错误的堆栈比筛选编译器诊断信息提供了更好的用户体验。有关详细解释，请参见基本原理部分。

_no_case()

`_no_case()` 返回 `true`，如果当前解析上下文在一个或多个（可能嵌套的）`no_case[]` 指令内。我没有这个的用例，但如果我不暴露它，它将是上下文中唯一不能从语义动作内部检查的东西。添加它很容易，所以我就做了。

1.1.7 规则解析器

这个示例与我们迄今为止看到的其他示例非常相似。不同之处在于它使用了一个 `rule`。作为类比，可以将解析器如 `char_` 或 `double_` 看作单独的一行代码，而将 `rule` 看作一个函数。像函数一样，`rule` 有自己的名字，甚至可以被前向声明。以下是我们定义 `rule` 的方式，这类似于前向声明一个函数：

```
bp::rule<struct doubles, std::vector<double>> doubles = "doubles";
```

这声明了规则本身。`rule` 是一个解析器，我们可以立即在其他解析器中使用它。这个定义相当密集；请注意以下几点：

- 第一个模板参数是标签类型 `struct doubles`。这里我们声明了标签类型并一次性使用了它；你也可以使用先前声明的标签类型。
- 第二个模板参数是解析器的属性类型。如果你不提供这个参数，规则将没有属性。
- 这个规则对象本身叫做 `doubles`。
- 我们给 `doubles` 提供了诊断文本 `"doubles"`，以便 Boost.Parser 在调试期间生成解析器的跟踪时知道如何引用它。

好了，那么如果 `doubles` 是一个解析器，它做什么呢？我们通过定义一个现在应该看起来很熟悉的单独解析器来定义规则的行为：

```
auto const doubles_def = bp::double_ % ',';
```

这类似于为前向声明的函数编写定义。请注意，我们使用了名称 `doubles_def`。现在，`doubles` 规则解析器和 `doubles_def` 非规则解析器之间没有任何连接。这是有意的——我们希望能够单独定义它们。为了连接它们，我们声明了一个 Boost.Parser 能理解的接口函数，并使用标签类型 `struct doubles` 将它们连接在一起。我们使用一个宏来实现这一点：

```
BOOST_PARSER_DEFINE_RULES(doubles);
```

这个宏展开为使规则 `doubles` 和其解析器 `doubles_def` 一起工作的必要代码。`_def` 后缀是这个宏依赖的命名约定。标签类型允许规则解析器 `doubles` 在用作解析器时调用这些重载之一。

`BOOST_PARSER_DEFINE_RULES` 展开为两个名为 `parse_rule()` 的函数重载。在上面的例子中，每个重载都接受一个 `struct doubles` 参数（以将它们与其他规则的 `parse_rule()` 重载区分开来）并使用 `doubles_def` 进行解析。你永远不需要自己调用任何 `parse_rule()` 的重载；它在实现 `rules` 的解析器 `rule_parser` 内部使用。

以下是为每个规则展开的宏定义：

```
#define BOOST_PARSER_DEFINE_IMPL(_, rule_name_)                                \
    template<                                                                  \
        typename Iter,                                                         \
        typename Sentinel,                                                     \
        typename Context,                                                      \
        typename SkipParser>                                                   \
    decltype(rule_name_)::parser_type::attr_type parse_rule(                   \
        decltype(rule_name_)::parser_type::tag_type *,                         \
        Iter & first,                                                          \
        Sentinel last,                                                         \
        Context const & context,                                               \
```

```
    SkipParser const & skip,                                                \
    boost::parser::detail::flags flags,                                     \
    bool & success,                                                         \
    bool & dont_assign)                                                     \
{                                                                           \
    auto const & parser = BOOST_PARSER_PP_CAT(rule_name_, _def);            \
    using attr_t =                                                          \
        decltype(parser(first, last, context, skip, flags, success));       \
    using attr_type = decltype(rule_name_)::parser_type::attr_type;         \
    if constexpr (boost::parser::detail::is_nope_v<attr_t>) {               \
        dont_assign = true;                                                 \
        parser(first, last, context, skip, flags, success);                 \
        return {};                                                          \
    } else if constexpr (std::is_same_v<attr_type, attr_t>) {               \
        return parser(first, last, context, skip, flags, success);          \
    } else if constexpr (std::is_constructible_v<attr_type, attr_t>) {      \
        return attr_type(                                                   \
            parser(first, last, context, skip, flags, success));            \
    } else {                                                                \
        attr_type attr{};                                                   \
        parser(first, last, context, skip, flags, success, attr);           \
        return attr;                                                        \
    }                                                                       \
}                                                                           \
                                                                            \
template<                                                                   \
    typename Iter,                                                          \
    typename Sentinel,                                                      \
    typename Context,                                                       \
    typename SkipParser,                                                    \
    typename Attribute>                                                     \
void parse_rule(                                                            \
    decltype(rule_name_)::parser_type::tag_type *,                          \
    Iter & first,                                                           \
    Sentinel last,                                                          \
    Context const & context,                                                \
    SkipParser const & skip,                                                \
    boost::parser::detail::flags flags,                                     \
    bool & success,                                                         \
    bool & dont_assign,                                                     \
    Attribute & retval)                                                     \
{                                                                           \
    auto const & parser = BOOST_PARSER_PP_CAT(rule_name_, _def);            \
    using attr_t =                                                          \
        decltype(parser(first, last, context, skip, flags, success));       \
    if constexpr (boost::parser::detail::is_nope_v<attr_t>) {               \
        parser(first, last, context, skip, flags, success);                 \
    } else {                                                                \
        parser(first, last, context, skip, flags, success, retval);         \
    }                                                                       \
}
```

现在我们有了 doubles 解析器，我们可以像使用其他解析器一样使用它：

```
auto const result = bp::parse(input, doubles, bp::ws);
```

完整程序：

```
#include <boost/parser/parser.hpp>

#include <deque>
#include <iostream>
#include <string>


namespace bp = boost::parser;


bp::rule<struct doubles, std::vector<double>> doubles = "doubles";
auto const doubles_def = bp::double_ % ',';
BOOST_PARSER_DEFINE_RULES(doubles);

int main()
{
    std::cout << "Please enter a list of doubles, separated by commas. ";
    std::string input;
    std::getline(std::cin, input);

    auto const result = bp::parse(input, doubles, bp::ws);

    if (result) {
        std::cout << "You entered:\n";
        for (double x : *result) {
            std::cout << x << "\n";
        }
    } else {
        std::cout << "Parse failure.\n";
    }
}
```

所有这些都是为了介绍 rules 的概念。你可能仍然不清楚为什么要使用 rules。关于 rules 的使用场景和详细信息，请参见后面的章节“更多关于规则”。

Note

The existence of rules means that will probably never have to write a low-level parser. You can just put existing parsers together into rules instead.

1.1.8 解析到结构体和类

到目前为止，我们只看到了简单的解析器，它们重复解析相同的值（有或没有逗号和空格）。解析一些特定顺序的值也是很常见的。假设你想解析一个员工记录。以下是你可能编写的解析器：

```
namespace bp = boost::parser;
auto employee_parser = bp::lit("employee")
    >> '{'
    >> bp::int_ >> ','
    >> quoted_string >> ','
    >> quoted_string >> ','
    >> bp::double_
    >> '}';
```

employee_parser 的属性类型是 boost::parser::tuple<int, std::string, std::string, double>。这很好，因为你在不需要编写任何语义动作的情况下获得了记录的所有解析数据。但不太好的是，你现在必须使用 get() 按索引获取所有单独的元素。将数据解析到程序将要使用的最终数据结构中会更好。这通常是一些 struct 或 class。Boost.Parser 支持解析到任意聚合 struct，以及可以从当前元组构造的非聚合类型。

作为属性的聚合类型

如果我们有一个 struct，其数据成员类型与 employee_parser 的 boost::parser::tuple 属性类型中列出的类型相同，那么直接解析到它，而不是先解析到一个元组然后再构造我们的 struct，会更好。幸运的是，这在 Boost.Parser 中是可行的。以下是直接解析到兼容聚合类型的示例。

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


struct employee
{
    int age;
    std::string surname;
    std::string forename;
    double salary;
};

namespace bp = boost::parser;

int main()
{
    std::cout << "Enter employee record. ";
    std::string input;
    std::getline(std::cin, input);

    auto quoted_string = bp::lexeme['"' >> +(bp::char_ - '"') >> '"'];
    auto employee_p = bp::lit("employee")
        >> '{'
        >> bp::int_ >> ','
        >> quoted_string >> ','
        >> quoted_string >> ','
        >> bp::double_
        >> '}';
```

```
    employee record;
    auto const result = bp::parse(input, employee_p, bp::ws, record);

    if (result) {
        std::cout << "You entered:\nage:      " << record.age
                  << "\nsurname:  " << record.surname
                  << "\nforename: " << record.forename
                  << "\nsalary  : " << record.salary << "\n";
    } else {
        std::cout << "Parse failure.\n";
    }
}
```

不幸的是，这利用了松散的属性赋值逻辑；`employee_parser` 解析器仍然具有 `boost::parser::tuple` 属性。有关属性输出参数兼容性的描述，请参见 `parse()` API。

因此，更常见的是希望创建一个返回特定类型（如 `employee`）的规则。只需给规则一个 `struct` 类型，我们就能确保这个解析器无论在解析的哪个位置，其属性始终是一个 `employee` 结构。如果我们创建一个使用 `employee_p` 规则的简单解析器 `P`，如 `bp::int >> employee_p`，那么 `P` 的属性类型将是 `boost::parser::tuple<int, employee>`。

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


struct employee
{
    int age;
    std::string surname;
    std::string forename;
    double salary;
};

namespace bp = boost::parser;

bp::rule<struct quoted_string, std::string> quoted_string = "quoted name";
bp::rule<struct employee_p, employee> employee_p = "employee";

auto quoted_string_def = bp::lexeme['"' >> +(bp::char_ - '"') >> '"'];
auto employee_p_def = bp::lit("employee")
    >> '{'
    >> bp::int_ >> ','
    >> quoted_string >> ','
    >> quoted_string >> ','
    >> bp::double_
    >> '}';

BOOST_PARSER_DEFINE_RULES(quoted_string, employee_p);
```

```
int main()
{
    std::cout << "Enter employee record. ";
    std::string input;
    std::getline(std::cin, input);

    static_assert(std::is_aggregate_v<std::decay_t<employee &>>);

    auto const result = bp::parse(input, employee_p, bp::ws);

    if (result) {
        std::cout << "You entered:\nage:      " << result->age
                  << "\nsurname:  " << result->surname
                  << "\nforename: " << result->forename
                  << "\nsalary  : " << result->salary << "\n";
    } else {
        std::cout << "Parse failure.\n";
    }
}
```

正如当解析器的属性类型是元组时，你可以将 `struct` 作为输出参数传递给 `parse()`，当解析器的属性类型是 `struct` 时，你也可以将元组作为输出参数传递给 `parse()`：

```
// Using the employee_p rule from above, with attribute type employee...
boost::parser::tuple<int, std::string, std::string, double> tup;
auto const result = bp::parse(input, employee_p, bp::ws, tup); // Ok!
```

Important

这种将 `struct` 自动用作元组的功能依赖于一些元编程。由于编译器的限制，检测 `struct` 数据成员数量的元程序有一个最大成员数的限制。幸运的是，这个限制是可配置的；请参见 `BOOST_PARSER_MAX_AGGREGATE_SIZE`。

作为属性的一般类类型

很多时候你没有一个聚合结构体来生成解析结果。如果 Boost.Parser 能检测到元组中的成员可以用作某个类型的构造函数参数，那就更好了。所以，Boost.Parser 确实做到了这一点。

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


namespace bp = boost::parser;

int main()
```

```
{
    std::cout << "Enter a string followed by two unsigned integers. ";
    std::string input;
    std::getline(std::cin, input);

    constexpr auto string_uint_uint =
        bp::lexeme[+(bp::char_ - ' ')] >> bp::uint_ >> bp::uint_;
    std::string string_from_parse;
    if (parse(input, string_uint_uint, bp::ws, string_from_parse))
        std::cout << "That yields this string: " << string_from_parse << "\n";
    else
        std::cout << "Parse failure.\n";

    std::cout << "Enter an unsigned integer followed by a string. ";
    std::getline(std::cin, input);
    std::cout << input << "\n";

    constexpr auto uint_string = bp::uint_ >> +bp::char_;
    std::vector<std::string> vector_from_parse;
    if (parse(input, uint_string, bp::ws, vector_from_parse)) {
        std::cout << "That yields this vector of strings:\n";
        for (auto && str : vector_from_parse) {
            std::cout << "  '" << str << "'\n";
        }
    } else {
        std::cout << "Parse failure.\n";
    }
}
```

Let's look at the first parse.

```
constexpr auto string_uint_uint =
    bp::lexeme[+(bp::char_ - ' ')] >> bp::uint_ >> bp::uint_;
std::string string_from_parse;
if (parse(input, string_uint_uint, bp::ws, string_from_parse))
    std::cout << "That yields this string: " << string_from_parse << "\n";
else
    std::cout << "Parse failure.\n";
```

Here, we use the parser `string_uint_uint`, which produces a `boost::parser::tuple<std::string, unsigned int, unsigned int>` attribute. When we try to parse that into an out-param `std::string` attribute, it just works. This is because `std::string` has a constructor that takes a `std::string`, an offset, and a length. Here's the other parse:

```
constexpr auto uint_string = bp::uint_ >> +bp::char_;
std::vector<std::string> vector_from_parse;
if (parse(input, uint_string, bp::ws, vector_from_parse)) {
    std::cout << "That yields this vector of strings:\n";
    for (auto && str : vector_from_parse) {
        std::cout << "  '" << str << "'\n";
    }
```

```
} else {
    std::cout << "Parse failure.\n";
}
```

Now we have the parser `uint_string`, which produces `boost::parser::tuple<unsigned int, std::string>` attribute — the two `char`s at the end combine into a `std::string`. Those two values can be used to construct a `std::vector<std::string>`, via the count, `T` constructor.

Just like with using aggregates in place of tuples, non-aggregate `class` types can be substituted for tuples in most places. That includes using a non-aggregate `class` type as the attribute type of a `rule`.

However, while compatible tuples can be substituted for aggregates, you **can't** substitute a tuple for some `class` type `T` just because the tuple could have been used to construct `T`. Think of trying to invert the substitution in the second parse above. Converting a `std::vector<std::string>` into a `boost::parser::tuple<unsigned int, std::string>` makes no sense.

1.1.9 Alternative Parsers

Frequently, you need to parse something that might have one of several forms. `operator|` is overloaded to form alternative parsers. For example:

```
namespace bp = boost::parser;
auto const parser_1 = bp::int_ | bp::eps;
```

`parser_1` matches an integer, or if that fails, it matches *epsilon*, the empty string. This is equivalent to writing:

```
namespace bp = boost::parser;
auto const parser_2 = -bp::int_;
```

However, neither `parser_1` nor `parser_2` is equivalent to writing this:

```
namespace bp = boost::parser;
auto const parser_3 = bp::eps | bp::int_; // Does not do what you think.
```

The reason is that alternative parsers try each of their subparsers, one at a time, and stop on the first one that matches. *Epsilon* matches anything, since it is zero length and consumes no input. It even matches the end of input. This means that `parser_3` is equivalent to `eps` by itself.

Note

For this reason, writing `eps | p` for any parser p is considered a bug. Debug builds will assert when `eps | p` is encountered.

Warning

This kind of error is very common when `eps` is involved, and also very easy to detect. However, it is possible to write `P1 >> P2`, where `P1` is a prefix of `P2`, such as `int_ | int >> int_`, or `repeat(4)[hex_digit] | repeat(8)[hex_digit]`. This is almost certainly an error, but is impossible to detect in the general case — remember that `rules` can be separately compiled, and consider a pair of rules whose associated `_def` parsers are `int_` and `int_ >> int_`, respectively.

1.1.10 Parsing Quoted Strings

It is very common to need to parse quoted strings. Quoted strings are slightly tricky, though, when using a skipper (and you should be using a skipper 99% of the time). You don't want to allow arbitrary whitespace in the middle of your strings, and you also don't want to remove all whitespace from your strings. Both of these things will happen with the typical skipper, `ws`.

So, here is how most people would write a quoted string parser:

```
namespace bp = boost::parser;
const auto string = bp::lexeme['"' >> *(bp::char_ - '"') > '"'];
```

Some things to note:

- the result is a string;
- the quotes are not included in the result;
- there is an expectation point before the close-quote;
- the use of `lexeme[]` disables skipping in the parser, and it must be written around the quotes, not around the `operator*` expression; and
- there's no way to write a quote in the middle of the string.

This is a very common pattern. I have written a quoted string parser like this dozens of times. The parser above is the quick-and-dirty version. A more robust version would be able to handle escaped quotes within the string, and then would immediately also need to support escaped escape characters.

Boost.Parser provides `quoted_string` to use in place of this very common pattern. It supports quote- and escaped-character-escaping, using backslash as the escape character.

```
namespace bp = boost::parser;

auto result1 = bp::parse("\"some text\"", bp::quoted_string, bp::ws);
assert(result1);
std::cout << *result1 << "\n"; // Prints: some text

auto result2 =
    bp::parse("\"some \\\"text\\\"\"", bp::quoted_string, bp::ws);
assert(result2);
std::cout << *result2 << "\n"; // Prints: some "text"
```

As common as this use case is, there are very similar use cases that it does not cover. So, `quoted_string` has some options. If you call it with a single character, it returns a `quoted_string` that uses that single character as the quote-character.

```
auto result3 = bp::parse("!some text!", bp::quoted_string('!'), bp::ws);
assert(result3);
std::cout << *result3 << "\n"; // Prints: some text
```

You can also supply a range of characters. One of the characters from the range must quote both ends of the string; mismatches are not allowed. Think of how Python allows you to quote a string with either `'"'` or `'\''`, but the same character must be used on both sides.

```
auto result4 = bp::parse("'some text'", bp::quoted_string("'\""), bp::ws);
assert(result4);
std::cout << *result4 << "\n"; // Prints: some text
```

Another common thing to do in a quoted string parser is to recognize escape sequences. If you have simple escape sequencecs that do not require any real parsing, like say the simple escape sequences from C++, you can provide a `symbols` object as well. The template parameter `T` to `symbols<T>` must be `char` or `char32_t`. You don't need to include the escaped backslash or the escaped quote character, since those always work.

```
// the c++ simple escapes
bp::symbols<char> const escapes = {
    {"'", '\''},
    {"?", '\?'},
    {"a", '\a'},
    {"b", '\b'},
    {"f", '\f'},
    {"n", '\n'},
    {"r", '\r'},
    {"t", '\t'},
    {"v", '\v'}};
auto result5 =
    bp::parse("\"some text\r\"", bp::quoted_string('"', escapes), bp::ws);
assert(result5);
std::cout << *result5 << "\n"; // Prints (with a CRLF newline): some text
```

1.1.11 Parsing In Detail

Now that you've seen some examples, let's see how parsing works in a bit more detail. Consider this example.

```
namespace bp = boost::parser;
auto int_pair = bp::int_ >> bp::int_;         // Attribute: tuple<int, int>
auto int_pairs_plus = +int_pair >> bp::int_;  // Attribute: tuple<std::vector<tuple<
    int, int>>, int>
```

`int_pairs_plus` must match a pair of `int`s (using `int_pair`) one or more times, and then must match an additional `int`. In other words, it matches any odd number (greater than 1) of `int`s in the input. Let's look at how this parse proceeds.

```
auto result = bp::parse("1 2 3", int_pairs_plus, bp::ws);
```

At the beginning of the parse, the top level parser uses its first subparser (if any) to start parsing. So, `int_pairs_plus`, being a sequence parser, would pass control to its first parser `+int_pair`. Then `+int_pair` would use `int_pair` to do its parsing, which would in turn use `bp::int_`. This creates a stack of parsers, each one using a particular subparser.

Step 1) The input is `"1 2 3"`, and the stack of active parsers is `int_pairs_plus` -> `+int_pair` -> `int_pair` -> `bp::int_`. (Read "->" as "uses".) This parses `"1"`, and the whitespace after is skipped by `bp::ws`. Control passes to the second `bp::int_` parser in `int_pair`.

Step 2) The input is `"2 3"` and the stack of parsers looks the same, except the active parser is the second `bp::int_` from `int_pair`. This parser consumes `"2"` and then `bp::ws` skips the subsequent space. Since we've finished with `int_pair`'s match, its `boost::parser::tuple<int, int>` attribute is complete. It's parent is `+int_pair`, so this tuple attribute is pushed onto the back of `+int_pair`'s attribute, which is a `std::vector<boost::parser::tuple<int, int>>`. Control passes up to the parent of `int_pair`, `+int_pair`. Since `+int_pair` is a one-or-more parser, it starts a new iteration; control passes to `int_pair` again.

Step 3) The input is `"3"` and the stack of parsers looks the same, except the active parser is the first `bp::int_` from `int_pair` again, and we're in the second iteration of `+int_pair`. This parser consumes `"3"`. Since this is the end of the input, the second `bp::int_` of `int_pair` does not match. This partial match of `"3"` should not count, since it was not part of a full match. So, `int_pair` indicates its failure, and `+int_pair` stops iterating. Since it did match once, `+int_pair` does not fail; it is a zero-or-more parser; failure of its subparser after the first success does not cause it to fail. Control passes to the next parser in sequence within `int_pairs_plus`.

Step 4) The input is `"3"` again, and the stack of parsers is `int_pairs_plus` -> `bp::int_`. This parses the `"3"`, and the parse reaches the end of input. Control passes to `int_pairs_plus`, which has just successfully matched with all parser in its sequence. It then produces its attribute, a `boost::parser::tuple<std::vector<boost::parser::t int>>, int>`, which gets returned from `bp::parse()`.

Something to take note of between Steps #3 and #4: at the beginning of #4, the input position had returned to where is was at the beginning of #3. This kind of backtracking happens in alternative parsers when an alternative fails. The next page has more details on the semantics of backtracking.

Parsers in detail

So far, parsers have been presented as somewhat abstract entities. You may be wanting more detail. A Boost.Parser parser `P` is an invocable object with a pair of call operator overloads. The two functions are very similar, and in many parsers one is implemented in terms of the other. The first function does the parsing and returns the default attribute for the parser. The second function does exactly the same parsing, but takes an out-param into which it writes the attribute for the parser. The out-param does not need to be the same type as the default attribute, but they need to be compatible.

Compatibility means that the default attribute is assignable to the out-param in some fashion. This usually means direct assignment, but it may also mean a tuple -> aggregate or aggregate -> tuple conversion. For sequence types, compatibility means that the sequence type has `insert` or `push_back` with the usual semantics. This means that the parser `+boost::parser::int_` can fill a `std::set<int>` just as well as a `std::vector<int>`.

Some parsers also have additional state that is required to perform a match. For instance, `char_` parsers can be parameterized with a single code point to match; the exact value of that code point is stored in the parser

object.

No parser has direct support for all the operations defined on parsers (`operator|`, `operator>>`, etc.). Instead, there is a template called `parser_interface` that supports all of these operations. `parser_interface` wraps each parser, storing it as a data member, adapting it for general use. You should only ever see `parser_interface` in the debugger, or possibly in some of the reference documentation. You should never have to write it in your own code.

1.1.12 Backtracking

As described in the previous page, backtracking occurs when the parse attempts to match the current parser `P`, matches part of the input, but fails to match all of `P`. The part of the input consumed during the parse of `P` is essentially "given back".

This is necessary because `P` may consist of subparsers, and each subparser that succeeds will try to consume input, produce attributes, etc. When a later subparser fails, the parse of `P` fails, and the input must be rewound to where it was when `P` started its parse, not where the latest matching subparser stopped.

Alternative parsers will often evaluate multiple subparsers one at a time, advancing and then restoring the input position, until one of the subparsers succeeds. Consider this example.

```
namespace bp = boost::parser;
auto const parser = repeat(53)[other_parser] | repeat(10)[other_parser];
```

Evaluating `parser` means trying to match `other_parser` 53 times, and if that fails, trying to match `other_parser` 10 times. Say you parse input that matches `other_parser` 11 times. `parser` will match it. It will also evaluate `other_parser` 21 times during the parse.

The attributes of the `repeat(53)[other_parser]` and `repeat(10)[other_parser]` are each `std::vector<ATTR(other_parser` let's say that *`ATTR`*`(other_parser)` is `int`. The attribute of `parser` as a whole is the same, `std::vector<int>`. Since `other_parser` is busy producing `int`s — 21 of them to be exact — you may be wondering what happens to the ones produced during the evaluation of `repeat(53)[other_parser]` when it fails to find all 53 inputs. Its `std::vector<int>` will contain 11 `int`s at that point.

When a repeat-parser fails, and attributes are being generated, it clears its container. This applies to parsers such as the ones above, but also all the other repeat parsers, including ones made using `operator+` or `operator*`.

So, at the end of a successful parse by `parser` of 10 inputs (since the right side of the alternative only eats 10 repetitions), the `std::vector<int>` attribute of `parser` would contain 10 `int`s.

Note

Users of Boost.Spirit may be familiar with the `hold[]` directive. Because of the behavior described above, there is no such directive in Boost.Parser.

Expectation points

Ok, so if parsers all try their best to match the input, and are all-or-nothing, doesn't that leave room for all kinds of bad input to be ignored? Consider the top-level parser from the Parsing JSON example.

```
auto const value_p_def =
    number | bp::bool_ | null | string | array_p | object_p;
```

What happens if I use this to parse `"\""`? The parse tries `number`, fails. It then tries `bp::bool_`, fails. Then `null` fails too. Finally, it starts parsing `string`. Good news, the first character is the open-quote of a JSON string. Unfortunately, that's also the end of the input, so `string` must fail too. However, we probably don't want to just give up on parsing `string` now and try `array_p`, right? If the user wrote an open-quote with no matching close-quote, that's not the prefix of some later alternative of `value_p_def`; it's ill-formed JSON. Here's the parser for the `string` rule:

```
auto const string_def = bp::lexeme['"' >> *(string_char - '"') > '"'];
```

Notice that `operator>` is used on the right instead of `operator>>`. This indicates the same sequence operation as `operator>>`, except that it also represents an expectation. If the parse before the `operator>` succeeds, whatever comes after it **must** also succeed. Otherwise, the top-level parse is failed, and a diagnostic is emitted. It will say something like "Expected '"' here.", quoting the line, with a caret pointing to the place in the input where it expected the right-side match.

Choosing to use `>` versus `>>` is how you indicate to Boost.Parser that parse failure is or is not a hard error, respectively.

1.1.13 Symbol Tables

When writing a parser, it often comes up that there is a set of strings that, when parsed, are associated with a set of values one-to-one. It is tedious to write parsers that recognize all the possible input strings when you have to associate each one with an attribute via a semantic action. Instead, we can use a symbol table.

Say we want to parse Roman numerals, one of the most common work-related parsing problems. We want to recognize numbers that start with any number of "M"s, representing thousands, followed by the hundreds, the tens, and the ones. Any of these may be absent from the input, but not all. Here are three symbol Boost.Parser tables that we can use to recognize ones, tens, and hundreds values, respectively:

```
bp::symbols<int> const ones = {
    {"I", 1},
    {"II", 2},
    {"III", 3},
    {"IV", 4},
    {"V", 5},
    {"VI", 6},
    {"VII", 7},
    {"VIII", 8},
    {"IX", 9}};

bp::symbols<int> const tens = {
    {"X", 10},
```

```
    {"XX", 20},
    {"XXX", 30},
    {"XL", 40},
    {"L", 50},
    {"LX", 60},
    {"LXX", 70},
    {"LXXX", 80},
    {"XC", 90}};

bp::symbols<int> const hundreds = {
    {"C", 100},
    {"CC", 200},
    {"CCC", 300},
    {"CD", 400},
    {"D", 500},
    {"DC", 600},
    {"DCC", 700},
    {"DCCC", 800},
    {"CM", 900}};
```

A `symbols` maps strings of `char` to their associated attributes. The type of the attribute must be specified as a template parameter to `symbols` — in this case, `int`.

Any "M"s we encounter should add 1000 to the result, and all other values come from the symbol tables. Here are the semantic actions we'll need to do that:

```
int result = 0;
auto const add_1000 = [&result](auto & ctx) { result += 1000; };
auto const add = [&result](auto & ctx) { result += _attr(ctx); };
```

`add_1000` just adds `1000` to `result`. `add` adds whatever attribute is produced by its parser to `result`.

Now we just need to put the pieces together to make a parser:

```
using namespace bp::literals;
auto const parser =
    *'M'_l[add_1000] >> -hundreds[add] >> -tens[add] >> -ones[add];
```

We've got a few new bits in play here, so let's break it down. `'M'_l` is a *literal parser*. That is, it is a parser that parses a literal `char`, code point, or string. In this case, a `char 'M'` is being parsed. The `_l` bit at the end is a UDL suffix that you can put after any `char`, `char32_t`, or `char const *` to form a literal parser. You can also make a literal parser by writing `lit()`, passing an argument of one of the previously mentioned types.

Why do we need any of this, considering that we just used a literal `','` in our previous example? The reason is that `'M'` is not used in an expression with another Boost.Parser parser. It is used within `*'M'_l[add_1000]`. If we'd written `*'M'[add_1000]`, clearly that would be ill-formed; `char` has no `operator*`, nor an `operator[]`, associated with it.

Tip

Any time you want to use a `char`, `char32_t`, or string literal in a Boost.Parser parser, write it as-is if it is combined with a preexisting Boost.Parser subparser `p`, as in `'x' >> p`. Otherwise, you need to wrap it in a call to `lit()`, or use the `_l` UDL suffix.

On to the next bit: `-hundreds[add]`. By now, the use of the index operator should be pretty familiar; it associates the semantic action `add` with the parser `hundreds`. The `operator-` at the beginning is new. It means that the parser it is applied to is optional. You can read it as "zero or one". So, if `hundreds` is not successfully parsed after `*'M'[add_1000]`, nothing happens, because `hundreds` is allowed to be missing — it's optional. If `hundreds` is parsed successfully, say by matching `"CC"`, the resulting attribute, `200`, is added to `result` inside `add`.

Here is the full listing of the program. Notice that it would have been inappropriate to use a whitespace skipper here, since the entire parse is a single number, so it was removed.

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


namespace bp = boost::parser;

int main()
{
    std::cout << "Enter a number using Roman numerals. ";
    std::string input;
    std::getline(std::cin, input);

    bp::symbols<int> const ones = {
        {"I", 1},
        {"II", 2},
        {"III", 3},
        {"IV", 4},
        {"V", 5},
        {"VI", 6},
        {"VII", 7},
        {"VIII", 8},
        {"IX", 9}};

    bp::symbols<int> const tens = {
        {"X", 10},
        {"XX", 20},
        {"XXX", 30},
        {"XL", 40},
        {"L", 50},
        {"LX", 60},
        {"LXX", 70},
        {"LXXX", 80},
```

```
        {"XC", 90}};

    bp::symbols<int> const hundreds = {
        {"C", 100},
        {"CC", 200},
        {"CCC", 300},
        {"CD", 400},
        {"D", 500},
        {"DC", 600},
        {"DCC", 700},
        {"DCCC", 800},
        {"CM", 900}};

    int result = 0;
    auto const add_1000 = [&result](auto & ctx) { result += 1000; };
    auto const add = [&result](auto & ctx) { result += _attr(ctx); };

    using namespace bp::literals;
    auto const parser =
        *'M'_l[add_1000] >> -hundreds[add] >> -tens[add] >> -ones[add];

    if (bp::parse(input, parser) && result != 0)
        std::cout << "That's " << result << " in Arabic numerals.\n";
    else
        std::cout << "That's not a Roman number.\n";
}
```

Important

`symbols` stores all its strings in UTF-32 internally. If you do Unicode or ASCII parsing, this will not matter to you at all. If you do non-Unicode parsing of a character encoding that is not a subset of Unicode (EBCDIC, for instance), it could cause problems. See the section on Unicode Support for more information.

Diagnostic messages

Just like with a `rule`, you can give a `symbols` a bit of diagnostic text that will be used in error messages generated by Boost.Parser when the parse fails at an expectation point, as described in Error Handling and Debugging. See the `symbols` constructors for details.

1.1.14 Mutable Symbol Tables

The previous example showed how to use a symbol table as a fixed lookup table. What if we want to add things to the table during the parse? We can do that, but we need to do so within a semantic action. First, here is our symbol table, already with a single value in it:

```
bp::symbols<int> const symbols = {{"c", 8}};
assert(parse("c", symbols));
```

No surprise that it works to use the symbol table as a parser to parse the one string in the symbol table. Now, here's our parser:

```
auto const parser = (bp::char_ >> bp::int_)[add_symbol] >> symbols;
```

Here, we've attached the semantic action not to a simple parser like `double_`, but to the sequence parser `(bp::char_ >> bp::int_)`. This sequence parser contains two parsers, each with its own attribute, so it produces two attributes as a tuple.

```
auto const add_symbol = [&symbols](auto & ctx) {
    using namespace bp::literals;
    // symbols::insert() requires a string, not a single character.
    char chars[2] = {_attr(ctx)[0_c], 0};
    symbols.insert(ctx, chars, _attr(ctx)[1_c]);
};
```

Inside the semantic action, we can get the first element of the attribute tuple using UDLs provided by Boost.Hana, and `boost::hana::tuple::operator[]()`. The first attribute, from the `char_`, is `_attr(ctx)[0_c]`, and the second, from the `int_`, is `_attr(ctx)[1_c]` (if `boost::parser::tuple` aliases to `std::tuple`, you'd use `std::get` or `boost::parser::get` instead). To add the symbol to the symbol table, we call `insert()`.

```
auto const parser = (bp::char_ >> bp::int_)[add_symbol] >> symbols;
```

During the parse, `("X", 9)` is parsed and added to the symbol table. Then, the second `'X'` is recognized by the symbol table parser. However:

```
assert(!parse("X", symbols));
```

If we parse again, we find that `"X"` did not stay in the symbol table. The fact that `symbols` was declared const might have given you a hint that this would happen.

The full program:

```
#include <boost/parser/parser.hpp>

#include <iostream>
#include <string>


namespace bp = boost::parser;

int main()
{
    bp::symbols<int> const symbols = {{"c", 8}};
    assert(parse("c", symbols));

    auto const add_symbol = [&symbols](auto & ctx) {
        using namespace bp::literals;
        // symbols::insert() requires a string, not a single character.
```

```
        char chars[2] = {_attr(ctx)[0_c], 0};
        symbols.insert(ctx, chars, _attr(ctx)[1_c]);
    };
    auto const parser = (bp::char_ >> bp::int_)[add_symbol] >> symbols;

    auto const result = parse("X 9 X", parser, bp::ws);
    assert(result && *result == 9);
    (void)result;

    assert(!parse("X", symbols));
}
```

Important

`symbols` stores all its strings in UTF-32 internally. If you do Unicode or ASCII parsing, this will not matter to you at all. If you do non-Unicode parsing of a character encoding that is not a subset of Unicode (EBCDIC, for instance), it could cause problems. See the section on Unicode Support for more information.

It is possible to add symbols to a `symbols` permanently. To do so, you have to use a mutable `symbols` object `s`, and add the symbols by calling `s.insert_for_next_parse()`, instead of `s.insert()`. These two operations are orthogonal, so if you want to both add a symbol to the table for the current top-level parse, and leave it in the table for subsequent top-level parses, you need to call both functions.

It is also possible to erase a single entry from the symbol table, or to clear the symbol table entirely. Just as with insertion, there are versions of erase and clear for the current parse, and another that applies only to subsequent parses. The full set of operations can be found in the `symbols` API docs.

[mpte There are two versions of each of the `symbols` `*_for_next_parse()` functions — one that takes a context, and one that does not. The one with the context is meant to be used within a semantic action. The one without the context is for use outside of any parse.]

1.1.15 The Parsers And Their Uses

Boost.Parser comes with all the parsers most parsing tasks will ever need. Each one is a `constexpr` object, or a `constexpr` function. Some of the non-functions are also callable, such as `char_`, which may be used directly, or with arguments, as in `char_('a', 'z')`. Any parser that can be called, whether a function or callable object, will be called a *callable parser* from now on. Note that there are no nullary callable parsers; they each take one or more arguments.

Each callable parser takes one or more *parse arguments*. A parse argument may be a value or an invocable object that accepts a reference to the parse context. The reference parameter may be mutable or constant. For example:

```
struct get_attribute
{
    template<typename Context>
```

```
    auto operator()(Context & ctx)
    {
        return _attr(ctx);
    }
};
```

This can also be a lambda. For example:

```
[](auto const & ctx) { return _attr(ctx); }
```

The operation that produces a value from a parse argument, which may be a value or a callable taking a parse context argument, is referred to as *resolving* the parse argument. If a parse argument `arg` can be called with the current context, then the resolved value of `arg` is `arg(ctx)`; otherwise, the resolved value is just `arg`.

Some callable parsers take a *parse predicate*. A parse predicate is not quite the same as a parse argument, because it must be a callable object, and cannot be a value. A parse predicate's return type must be contextually convertible to `bool`. For example:

```
struct equals_three
{
    template<typename Context>
    bool operator()(Context const & ctx)
    {
        return _attr(ctx) == 3;
    }
};
```

This may of course be a lambda:

```
[](auto & ctx) { return _attr(ctx) == 3; }
```

The notional macro *`RESOLVE()`* expands to the result of resolving a parse argument or parse predicate. You'll see it used in the rest of the documentation.

An example of how parse arguments are used:

```
namespace bp = boost::parser;
// This parser matches one code point that is at least 'a', and at most
// the value of last_char, which comes from the globals.
auto last_char = [](auto & ctx) { return _globals(ctx).last_char; }
auto subparser = bp::char_('a', last_char);
```

Don't worry for now about what the globals are for now; the take-away is that you can make any argument you pass to a parser depend on the current state of the parse, by using the parse context:

```
namespace bp = boost::parser;
// This parser parses two code points.  For the parse to succeed, the
// second one must be >= 'a' and <= the first one.
auto set_last_char = [](auto & ctx) { _globals(ctx).last_char = _attr(x); };
auto parser = bp::char_[set_last_char] >> subparser;
```

Each callable parser returns a new parser, parameterized using the arguments given in the invocation.

This table lists all the Boost.Parser parsers. For the callable parsers, a separate entry exists for each possible arity of arguments. For a parser `p`, if there is no entry for `p` without arguments, `p` is a function, and cannot itself be used as a parser; it must be called. In the table below:

- each entry is a global object usable directly in your parsers, unless otherwise noted;
- "code point" is used to refer to the elements of the input range, which assumes that the parse is being done in the Unicode-aware code path (if the parse is being done in the non-Unicode code path, read "code point" as "`char`");
- *`RESOLVE()`* is a notional macro that expands to the resolution of parse argument or evaluation of a parse predicate (see The Parsers And Their Uses);
- "*`RESOLVE(pred) == true`*" is a shorthand notation for "*`RESOLVE(pred)`* is contextually convertible to `bool` and `true`"; likewise for `false`;
- `c` is a character of type `char`, `char8_t`, or `char32_t`;
- `str` is a string literal of type `char const[]`, `char8_t const []`, or `char32_t const []`;
- `pred` is a parse predicate;
- `arg0`, `arg1`, `arg2`, ... are parse arguments;
- `a` is a semantic action;
- `r` is an object whose type models `parsable_range`;
- `p`, `p1`, `p2`, ... are parsers; and
- `escapes` is a `symbols<T>` object, where `T` is `char` or `char32_t`.

Note

The definition of `parsable_range` is:

```
template<typename T>
concept parsable_range = std::ranges::
    forward_range<T> &&
     code_unit<std::ranges::range_value_t<T
    >>;
```

Note

Some of the parsers in this table consume no input. All parsers consume the input they match unless otherwise stated in the table below.

Table 26.6. Parsers and Their Semantics

Parser	Semantics
`eps`	Matches *epsilon*, the empty string. Always matches, and consumes no input.
`eps(pred)`	Fails to match the input if *`RESOLVE(pred) == false`*. Otherwise, the semantic
`ws`	Matches a single whitespace code point (see note), according to the Unicode W
`eol`	Matches a single newline (see note), following the "hard" line breaks in the Un
`eoi`	Matches only at the end of input, and consumes no input.
`attr(arg0)`	Always matches, and consumes no input. Generates the attribute *`RESOLVE(arg`*
`char_`	Matches any single code point.
`char_(arg0)`	Matches exactly the code point *`RESOLVE(arg0)`*.
`char_(arg0, arg1)`	Matches the next code point `n` in the input, if *`RESOLVE(arg0) <= n && n <= R`*

Parser	Semantics
`char_(r)`	Matches the next code point `n` in the input, if `n` is one of the code points
`cp`	Matches a single code point.
`cu`	Matches a single code point.
`blank`	Equivalent to `ws - eol`.
`control`	Matches a single control-character code point.
`digit`	Matches a single decimal digit code point.
`punct`	Matches a single punctuation code point.
`hex_digit`	Matches a single hexidecimal digit code point.
`lower`	Matches a single lower-case code point.
`upper`	Matches a single upper-case code point.
`lit(c)`	Matches exactly the given code point `c`.
`c_l`	Matches exactly the given code point `c`.
`lit(r)`	Matches exactly the given string `r`.
`str_l`	Matches exactly the given string `str`.
`string(r)`	Matches exactly `r`, and generates the match as an attribute.
`str_p`	Matches exactly `str`, and generates the match as an attribute.
`bool_`	Matches `"true"` or `"false"`.
`bin`	Matches a binary unsigned integral value.
`bin(arg0)`	Matches exactly the binary unsigned integral value *`RESOLVE`*`(arg0)`.
`oct`	Matches an octal unsigned integral value.
`oct(arg0)`	Matches exactly the octal unsigned integral value *`RESOLVE`*`(arg0)`.
`hex`	Matches a hexadecimal unsigned integral value.
`hex(arg0)`	Matches exactly the hexadecimal unsigned integral value *`RESOLVE`*`(arg0`
`ushort_`	Matches an unsigned integral value.
`ushort_(arg0)`	Matches exactly the unsigned integral value *`RESOLVE`*`(arg0)`.
`uint_`	Matches an unsigned integral value.
`uint_(arg0)`	Matches exactly the unsigned integral value *`RESOLVE`*`(arg0)`.
`ulong_`	Matches an unsigned integral value.
`ulong_(arg0)`	Matches exactly the unsigned integral value *`RESOLVE`*`(arg0)`.
`ulong_long`	Matches an unsigned integral value.
`ulong_long(arg0)`	Matches exactly the unsigned integral value *`RESOLVE`*`(arg0)`.
`short_`	Matches a signed integral value.
`short_(arg0)`	Matches exactly the signed integral value *`RESOLVE`*`(arg0)`.
`int_`	Matches a signed integral value.
`int_(arg0)`	Matches exactly the signed integral value *`RESOLVE`*`(arg0)`.
`long_`	Matches a signed integral value.
`long_(arg0)`	Matches exactly the signed integral value *`RESOLVE`*`(arg0)`.
`long_long`	Matches a signed integral value.
`long_long(arg0)`	Matches exactly the signed integral value *`RESOLVE`*`(arg0)`.
`float_`	Matches a floating-point number. `float_` uses parsing implementation
`double_`	Matches a floating-point number. `double_` uses parsing implementation
`repeat(arg0)[p]`	Matches iff `p` matches exactly *`RESOLVE`*`(arg0)` times.
`repeat(arg0, arg1)[p]`	Matches iff `p` matches between *`RESOLVE`*`(arg0)` and *`RESOLVE`*`(arg1)` time
`if_(pred)[p]`	Equivalent to `eps(pred) >> p`.
`switch_(arg0)(arg1, p1)(arg2, p2) ...`	Equivalent to `p1` when *`RESOLVE`*`(arg0) == `*`RESOLVE`*`(arg1)`, `p2` when *`RES`
`symbols<T>`	`symbols` is an associative container of key, value pairs. Each key is a `st`
`quoted_string`	Matches `'"'`, followed by zero or more characters, followed by `'"'`.
`quoted_string(c)`	Matches `c`, followed by zero or more characters, followed by `c`.
`quoted_string(r)`	Matches some character `Q` in `r`, followed by zero or more characters, foll

Parser	Semantics
`quoted_string(c, symbols)`	Matches `c`, followed by zero or more characters, followed by `c`.
`quoted_string(r, symbols)`	Matches some character `Q` in `r`, followed by zero or more characters, followed

Important

All the character parsers, like `char_`, `cp` and `cu` produce either `char` or `char32_t` attributes. So when you see "`std::string` if *`ATTR`*`(p)` is `char` or `char32_t`, otherwise `std::vector<`*`ATTR`*`(p)>`" in the table above, that effectively means that every sequences of character attributes get turned into a `std::string`. The only time this does not happen is when you introduce your own rules with attributes using another character type (or use `attribute` to do so).

Note

A slightly more complete description of the attributes generated by these parsers is in a subsequent section. The attributes are repeated here so you can use see all the properties of the parsers in one place.

If you have an integral type `IntType` that is not covered by any of the Boost.Parser parsers, you can use a more verbose declaration to declare a parser for `IntType`. If `IntType` were unsigned, you would use `uint_parser`. If it were signed, you would use `int_parser`. For example:

```
constexpr parser_interface<int_parser<IntType>> hex_int;
```

`uint_parser` and `int_parser` accept three more non-type template parameters after the type parameter. They are `Radix`, `MinDigits`, and `MaxDigits`. `Radix` defaults to `10`, `MinDigits` to `1`, and `MaxDigits` to `-1`, which is a sentinel value meaning that there is no max number of digits.

So, if you wanted to parse exactly eight hexadecimal digits in a row in order to recognize Unicode character literals like C++ has (e.g. `\Udeadbeef`), you could use this parser for the digits at the end:

```
constexpr parser_interface<uint_parser<unsigned int, 16, 8, 8>> hex_int;
```

1.1.16 Directives

A directive is an element of your parser that doesn't have any meaning by itself. Some are second-order parsers that need a first-order parser to do the actual parsing. Others influence the parse in some way. You

can often spot a directive lexically by its use of `[]`; directives always `[]`. Non-directives might, but only when attaching a semantic action.

The directives that are second order parsers are technically directives, but since they are also used to create parsers, it is more useful just to focus on that. The directives `repeat()` and `if_()` were already described in the section on parsers; we won't say much about them here.

Interaction with sequence, alternative, and permutation parsers

Sequence, alternative, and permutation parsers do not nest in most cases. (Let's consider just sequence parsers to keep thinkgs simple, but most of this logic applies to alternative parsers as well.) `a >> b >> c` is the same as `(a >> b) >> c` and `a >> (b >> c)`, and they are each represented by a single `seq_parser` with three subparsers, `a`, `b`, and `c`. However, if something prevents two `seq_parsers` from interacting directly, they **will** nest. For instance, `lexeme[a >> b] >> c` is a `seq_parser` containing two parsers, `lexeme[a >> b]` and `c`. This is because `lexeme[]` takes its given parser and wraps it in a `lexeme_parser`. This in turn turns off the sequence parser combining logic, since both sides of the second `operator>>` in `lexeme[a >> b] >> c` are not `seq_parsers`. Sequence parsers have several rules that govern what the overall attribute type of the parser is, based on the positions and attributes of it subparsers (see Attribute Generation). Therefore, it's important to know which directives create a new parser (and what kind), and which ones do not; this is indicated for each directive below.

The directives

repeat()

See The Parsers And Their Uses. Creates a `repeat_parser`.

if_()

See The Parsers And Their Uses. Creates a `seq_parser`.

omit[]

`omit[p]` disables attribute generation for the parser `p`. Not only does `omit[p]` have no attribute, but any attribute generation work that normally happens within `p` is skipped.

This directive can be useful in cases like this: say you have some fairly complicated parser `p` that generates a large and expensive-to-construct attribute. Now say that you want to write a function that just counts how many times `p` can match a string (where the matches are non-overlapping). Instead of using `p` directly, and building all those attributes, or rewriting `p` without the attribute generation, use `omit[]`.

Creates an `omit_parser`.

raw[]

`raw[p]` changes the attribute from *ATTR*`(p)` to to a view that delimits the subrange of the input that was matched by `p`. The type of the view is `subrange<I>`, where `I` is the type of the iterator used within the parse. Note that this may not be the same as the iterator type passed to `parse()`. For instance, when parsing UTF-8, the iterator passed to `parse()` may be `char8_t const *`, but within the parse it will be a UTF-8 to UTF-32 transcoding (converting) iterator. Just like `omit[]`, `raw[]` causes all attribute-generation work within `p` to be skipped.

Similar to the re-use scenario for `omit[]` above, `raw[]` could be used to find the **locations** of all non-overlapping matches of `p` in a string.

Creates a `raw_parser`.

string_view[]

`string_view[p]` is very similar to `raw[p]`, except that it changes the attribute of `p` to `std::basic_string_view<C>`, where `C` is the character type of the underlying range being parsed. `string_view[]` requires that the underlying range being parsed is contiguous. Since this can only be detected in C++20 and later, `string_view[]` is not available in C++17 mode.

Similar to the re-use scenario for `omit[]` above, `string_view[]` could be used to find the **locations** of all non-overlapping matches of `p` in a string. Whether `raw[]` or `string_view[]` is more natural to use to report the locations depends on your use case, but they are essentially the same.

Creates a `string_view_parser`.

no_case[]

`no_case[p]` enables case-insensitive parsing within the parse of `p`. This applies to the text parsed by `char_()`, `string()`, and `bool_` parsers. The number parsers are already case-insensitive. The case-insensitivity is achieved by doing Unicode case folding on the text being parsed and the values in the parser being matched (see note below if you want to know more about Unicode case folding). In the non-Unicode code path, a full Unicode case folding is not done; instead, only the transformations of values less than `0x100` are done. Examples:

```
#include <boost/parser/transcode_view.hpp> // For as_utfN.

namespace bp = boost::parser;
auto const street_parser = bp::string(u8"Tobias Straße");
assert(!bp::parse("Tobias Strasse" | bp::as_utf32, street_parser));             // No
    match.
assert(bp::parse("Tobias Strasse" | bp::as_utf32, bp::no_case[street_parser])); //
    Match!

auto const alpha_parser = bp::no_case[bp::char_('a', 'z')];
assert(bp::parse("a" | bp::as_utf32, bp::no_case[alpha_parser])); // Match!
assert(bp::parse("B" | bp::as_utf32, bp::no_case[alpha_parser])); // Match!
```

Everything pretty much does what you'd naively expect inside `no_case[]`, except that the two-character range version of `char_` has a limitation. It only compares a code point from the input to its two arguments (e.g. `'a'` and `'z'` in the example above). It does not do anything special for multi-code point case folding expansions. For instance, `char_(U'ß', U'ß')` matches the input `U"s"`, which makes sense, since `U'ß'` expands to `U"ss"`. However, that same parser **does not** match the input `U"ß"`! In short, stick to pairs of code points that have single-code point case folding expansions. If you need to support the multi-expanding code points, use the other overload, like: `char_(U"abcd/*...*/ß")`.

Note
Unicode case folding is an operation that makes text uniformly one case, and if you do it to two bits of text `A` and `B`, then you can compare them bitwise to see if they are the same, except of case. Case folding may sometimes expand a code point into multiple code points (e.g. case folding `"ß"` yields `"ss"`. When such a multi-code point expansion occurs, the expanded code points are in the NFKC normalization form.

Creates a `no_case_parser`.

lexeme[]

`lexeme[p]` disables use of the skipper, if a skipper is being used, within the parse of `p`. This is useful, for instance, if you want to enable skipping in most parts of your parser, but disable it only in one section where it doesn't belong. If you are skipping whitespace in most of your parser, but want to parse strings that may contain spaces, you should use `lexeme[]`:

```
namespace bp = boost::parser;
auto const string_parser = bp::lexeme['"' >> *(bp::char_ - '"') >> '"'];
```

Without `lexeme[]`, our string parser would correctly match `"foo bar"`, but the generated attribute would be `"foobar"`.

Creates a `lexeme_parser`.

skip[]

`skip[]` is like the inverse of `lexeme[]`. It enables skipping in the parse, even if it was not enabled before. For example, within a call to `parse()` that uses a skipper, let's say we have these parsers in use:

```
namespace bp = boost::parser;
auto const one_or_more = +bp::char_;
auto const skip_or_skip_not_there_is_no_try = bp::lexeme[bp::skip[one_or_more] >>
    one_or_more];
```

The use of `lexeme[]` disables skipping, but then the use of `skip[]` turns it back on. The net result is that the first occurrence of `one_or_more` will use the skipper passed to `parse()`; the second will not.

`skip[]` has another use. You can parameterize skip with a different parser to change the skipper just within the scope of the directive. Let's say we passed `ws` to `parse()`, and we're using these parsers somewhere within that `parse()` call:

```
namespace bp = boost::parser;
auto const zero_or_more = *bp::char_;
auto const skip_both_ways = zero_or_more >> bp::skip(bp::blank)[zero_or_more];
```

The first occurrence of `zero_or_more` will use the skipper passed to `parse()`, which is `ws`; the second will use `blank` as its skipper.

Creates a `skip_parser`.

merge[], separate[], and transform(f)[]

These directives influence the generation of attributes. See Attribute Generation section for more details on them.

`merge[]` and `separate[]` create a copy of the given `seq_parser`.

`transform(f)[]` creates a `tranform_parser`.

1.1.17 Combining Operations

Certain overloaded operators are defined for all parsers in Boost.Parser. We've already seen some of them used in this tutorial, especially `operator>>`, `operator|`, and `operator||`, which are used to form sequence parsers, alternative parsers, and permutation parsers, respectively.

Here are all the operator overloaded for parsers. In the tables below:

- `c` is a character of type `char` or `char32_t`;
- `a` is a semantic action;
- `r` is an object whose type models `parsable_range` (see Concepts); and
- `p`, `p1`, `p2`, ... are parsers.

Note

Some of the expressions in this table consume no input. All parsers consume the input they match unless otherwise stated in the table below.

Table 26.7. Combining Operations and Their Semantics

Expression	Semantics
`!p`	Matches iff `p` does not match; consumes no input.
`&p`	Matches iff `p` matches; consumes no input.
`*p`	Parses using `p` repeatedly until `p` no longer matches; always matches.

Expression	Semantics
`+p`	Parses using `p` repeatedly until `p` no longer matches; matches iff `p` matches at least once.
`-p`	Equivalent to `p | eps`.
`p1 >> p2`	Matches iff `p1` matches and then `p2` matches.
`p >> c`	Equivalent to `p >> lit(c)`.
`p >> r`	Equivalent to `p >> lit(r)`.
`p1 > p2`	Matches iff `p1` matches and then `p2` matches. No back-tracking is allowed after `p1` matches; if `p1` matches
`p > c`	Equivalent to `p > lit(c)`.
`p > r`	Equivalent to `p > lit(r)`.
`p1 | p2`	Matches iff either `p1` matches or `p2` matches.
`p | c`	Equivalent to `p | lit(c)`.
`p | r`	Equivalent to `p | lit(r)`.
`p1 || p2`	Matches iff `p1` matches and `p2` matches, regardless of the order they match in.
`p1 - p2`	Equivalent to `!p2 >> p1`.
`p - c`	Equivalent to `p - lit(c)`.
`p - r`	Equivalent to `p - lit(r)`.
`p1 % p2`	Equivalent to `p1 >> *(p2 >> p1)`.
`p % c`	Equivalent to `p % lit(c)`.
`p % r`	Equivalent to `p % lit(r)`.
`p[a]`	Matches iff `p` matches. If `p` matches, the semantic action `a` is executed.

Important

All the character parsers, like `char_`, `cp` and `cu` produce either `char` or `char32_t` attributes. So when you see "`std::string` if *`ATTR`*`(p)` is `char` or `char32_t`, otherwise `std::vector<`*`ATTR`*`(p)>`" in the table above, that effectively means that every sequences of character attributes get turned into a `std::string`. The only time this does not happen is when you introduce your own rules with attributes using another character type (or use `attribute` to do so).

There are a couple of special rules not captured in the table above:

First, the zero-or-more and one-or-more repetitions (`operator*()` and `operator+()`, respectively) may collapse when combined. For any parser `p`, `+(+p)` collapses to `+p`; `**p`, `*+p`, and `+*p` each collapse to just `*p`.

Second, using `eps` in an alternative parser as any alternative **except** the last one is a common source of errors; Boost.Parser disallows it. This is true because, for any parser `p`, `eps | p` is equivalent to `eps`, since `eps` always matches. This is not true for `eps` parameterized with a condition. For any condition `cond`, `eps(cond)` is allowed to appear anywhere within an alternative parser.

Note

When looking at Boost.Parser parsers in a debugger, or when looking at their reference documentation, you may see reference to the template `parser_interface`. This template exists to provide the operator overloads described above. It allows the parsers themselves to be very simple — most parsers are just a struct with two member functions. `parser_interface` is essentially invisible when using Boost.Parser, and you should never have to name this template in your own code.

1.1.18 Attribute Generation

So far, we've seen several different types of attributes that come from different parsers, `int` for `int_`, `boost::parser::tuple<ch` `int>` for `boost::parser::char_ >> boost::parser::int_`, etc. Let's get into how this works with more rigor.

Note

Some parsers have no attribute at all. In the tables below, the type of the attribute is listed as "None." There is a non-`void` type that is returned from each parser that lacks an attribute. This keeps the logic simple; having to handle the two cases — `void` or non-`void` — would make the library significantly more complicated. The type of this non-`void` attribute associated with these parsers is an implementation detail. The type comes from the `boost::parser::detail` namespace and is pretty useless. You should never see this type in practice. Within semantic actions, asking for the attribute of a non-attribute-producing parser (using `_attr(ctx)`) will yield a value of the special type `boost::parser::none`. When calling `parse()` in a form that returns the attribute parsed, when there is no attribute, simply returns `bool`; this indicates the success of failure of the parse.

Warning

Boost.Parser assumes that all attributes are semi-regular (see `std::semiregular`). Within the Boost.Parser code, attributes are assigned, moved, copy, and default constructed. There is no support for move-only or non-default-constructible types.

The attribute type trait, attribute

You can use `attribute` (and the associated alias, `attribute_t`) to determine the attribute a parser would have if it were passed to `parse()`. Since at least one parser (`char_`) has a polymorphic attribute type, `attribute` also takes the type of the range being parsed. If a parser produces no attribute, `attribute` will produce `none`, not `void`.

If you want to feed an iterator/sentinel pair to `attribute`, create a range from it like so:

```
constexpr auto parser = /* ... */;
auto first = /* ... */;
auto const last = /* ... */;

namespace bp = boost::parser;
// You can of course use std::ranges::subrange directly in C++20 and later.
using attr_type = bp::attribute_t<decltype(BOOST_PARSER_SUBRANGE(first, last)),
    decltype(parser)>;
```

There is no single attribute type for any parser, since a parser can be placed within `omit[]`, which makes its attribute type `none`. Therefore, `attribute` cannot tell you what attribute your parser will produce under all circumstances; it only tells you what it would produce if it were passed to `parse()`.

Parser attributes

This table summarizes the attributes generated for all Boost.Parser parsers. In the table below:

- *`RESOLVE()`* is a notional macro that expands to the resolution of parse argument or evaluation of a parse predicate (see The Parsers And Their Uses); and
- `x` and `y` represent arbitrary objects.

Table 26.8. Parsers and Their Attributes

Parser	Attribute Type	Notes
`eps`	None.	
`eol`	None.	
`eoi`	None.	
`attr(x)`	`decltype(`*`RESOLVE`*`(x))`	
`char_`	The code point type in Unicode parsing, or `char` in non-Unicode parsing; see below.	Includes all the _
`cp`	`char32_t`	
`cu`	`char`	
`lit(x)`	None.	Includes all the _
`string(x)`	`std::string`	Includes all the _
`bool_`	`bool`	
`bin`	`unsigned int`	
`oct`	`unsigned int`	
`hex`	`unsigned int`	
`ushort_`	`unsigned short`	
`uint_`	`unsigned int`	

Parser	Attribute Type	Notes
`ulong_`	`unsigned long`	
`ulong_long`	`unsigned long long`	
`short_`	`short`	
`int_`	`int`	
`long_`	`long`	
`long_long`	`long long`	
`float_`	`float`	
`double_`	`double`	
`symbols<T>`	`T`	

`char_` is a bit odd, since its attribute type is polymorphic. When you use `char_` to parse text in the non-Unicode code path (i.e. a string of `char`), the attribute is `char`. When you use the exact same `char_` to parse in the Unicode-aware code path, all matching is code point based, and so the attribute type is the type used to represent code points, `char32_t`. All parsing of UTF-8 falls under this case.

Here, we're parsing plain `char`s, meaning that the parsing is in the non-Unicode code path, the attribute of `char_` is `char`:

```
auto result = parse("some text", boost::parser::char_);
static_assert(std::is_same_v<decltype(result), std::optional<char>>));
```

When you parse UTF-8, the matching is done on a code point basis, so the attribute type is `char32_t`:

```
auto result = parse("some text" | boost::parser::as_utf8, boost::parser::char_);
static_assert(std::is_same_v<decltype(result), std::optional<char32_t>>));
```

The good news is that usually you don't parse characters individually. When you parse with `char_`, you usually parse repetition of then, which will produce a `std::string`, regardless of whether you're in Unicode parsing mode or not. If you do need to parse individual characters, and want to lock down their attribute type, you can use `cp` and/or `cu` to enforce a non-polymorphic attribute type.

Combining operation attributes

Combining operations of course affect the generation of attributes. In the tables below:

- `m` and `n` are parse arguments that resolve to integral values;
- `pred` is a parse predicate;
- `arg0`, `arg1`, `arg2`, ... are parse arguments;
- `a` is a semantic action; and
- `p`, `p1`, `p2`, ... are parsers that generate attributes.

Table 26.9. Combining Operations and Their Attributes

Parser	Attribute Type
`!p`	None.
`&p`	None.
`*p`	`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<ATT`
`+p`	`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<ATT`
`+*p`	`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<ATT`
`*+p`	`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<ATT`
`-p`	`std::optional<`*`ATTR(p)`*`>`
`p1 >> p2`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`>`
`p1 > p2`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`>`
`p1 >> p2 >> p3`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`, `*`ATTR(p3)`*`>`
`p1 > p2 >> p3`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`, `*`ATTR(p3)`*`>`
`p1 >> p2 > p3`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`, `*`ATTR(p3)`*`>`
`p1 > p2 > p3`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`, `*`ATTR(p3)`*`>`
`p1 | p2`	`std::variant<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`>`
`p1 | p2 | p3`	`std::variant<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`, `*`ATTR(p3)`*`>`
`p1 || p2`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`>`
`p1 || p2 || p3`	`boost::parser::tuple<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`, `*`ATTR(p3)`*`>`
`p1 % p2`	`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<ATT`
`p[a]`	None.
`repeat(arg0)[p]`	`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<ATT`
`repeat(arg0, arg1)[p]`	`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<ATT`
`if_(pred)[p]`	`std::optional<`*`ATTR(p)`*`>`
`switch_(arg0)(arg1, p1)(arg2, p2)...`	`std::variant<`*`ATTR(p1)`*`, `*`ATTR(p2)`*`, ...>`

Important

All the character parsers, like `char_`, `cp` and `cu` produce either `char` or `char32_t` attributes. So when you see "`std::string` if *`ATTR(p)`* is `char` or `char32_t`, otherwise `std::vector<`*`ATTR(p)`*`>`" in the table above, that effectively means that every sequences of character attributes get turned into a `std::string`. The only time this does not happen is when you introduce your own rules with attributes using another character type (or use `attribute` to do so).

Important

In case you did not notice it above, adding a semantic action to a parser erases the parser's attribute. The attribute is still available inside the semantic action as `_attr(ctx)`.

There are a relatively small number of rules that define how sequence parsers and alternative parsers' attributes are generated. (Don't worry, there are examples below.)

Sequence parser attribute rules

The attribute generation behavior of sequence parsers is conceptually pretty simple:

- the attributes of subparsers form a tuple of values;
- subparsers that do not generate attributes do not contribute to the sequence's attribute;
- subparsers that do generate attributes usually contribute an individual element to the tuple result; except
- when containers of the same element type are next to each other, or individual elements are next to containers of their type, the two adjacent attributes collapse into one attribute; and
- if the result of all that is a degenerate tuple `boost::parser::tuple<T>` (even if `T` is a type that means "no attribute"), the attribute becomes `T`.

More formally, the attribute generation algorithm works like this. For a sequence parser `p`, let the list of attribute types for the subparsers of `p` be `a0, a1, a2, ..., an`.

We get the attribute of `p` by evaluating a compile-time left fold operation, `left-fold({a1, a2, ..., an}, tuple<a0>, OP)`. `OP` is the combining operation that takes the current attribute type (initially `boost::parser::tuple<a0>`) and the next attribute type, and returns the new current attribute type. The current attribute type at the end of the fold operation is the attribute type for `p`.

`OP` attempts to apply a series of rules, one at a time. The rules are noted as `X >> Y -> Z`, where `X` is the type of the current attribute, `Y` is the type of the next attribute, and `Z` is the new current attribute type. In these rules, `C<T>` is a container of `T`; `none` is a special type that indicates that there is no attribute; `T` is a type; `CHAR` is a character type, either `char` or `char32_t`; and `Ts...` is a parameter pack of one or more types. Note that `T` may be the special type `none`. The current attribute is always a tuple (call it `Tup`), so the "current attribute `X`" refers to the last element of `Tup`, not `Tup` itself, except for those rules that explicitly mention `boost::parser::tuple<>` as part of `X`'s type.

- `none >> T -> T`
- `CHAR` » `CHAR` -> `std::string`
- `T >> none -> T`
- `C<T> >> T -> C<T>`
- `T >> C<T> -> C<T>`
- `C<T> >> optional<T> -> C<T>`
- `optional<T> >> C<T> -> C<T>`
- `boost::parser::tuple<none> >> T -> boost::parser::tuple<T>`
- `boost::parser::tuple<Ts...> >> T -> boost::parser::tuple<Ts..., T>`

The rules that combine containers with (possibly optional) adjacent values (e.g. `C<T> >> optional<T> -> C<T>`) have a special case for strings. If `C<T>` is exactly `std::string`, and `T` is either `char` or `char32_t`, the combination yields a `std::string`.

Again, if the final result is that the attribute is `boost::parser::tuple<T>`, the attribute becomes `T`.

Note

What constitutes a container in the rules above is determined by the `container` concept:

```
template<typename T>
concept container = std::ranges::
    common_range<T> && requires(T t) {
    { t.insert(t.begin(), *t.begin()) }
        -> std::same_as<std::ranges::
    iterator_t<T>>;
};
```

Alternative parser attribute rules

The rules for alternative parsers are much simpler. For an alternative parer `p`, let the list of attribute types for the subparsers of `p` be `a0, a1, a2, ..., an`. The attribute of `p` is `std::variant<a0, a1, a2, ..., an>`, with the following steps applied:

- all the `none` attributes are left out, and if any are, the attribute is wrapped in a `std::optional`, like `std::optional<std::variant</*...*/>>`;
- duplicates in the `std::variant` template parameters `<T1, T2, ... Tn>` are removed; every type that appears does so exacly once;
- if the attribute is `std::variant<T>` or `std::optional<std::variant<T>>`, the attribute becomes instead `T` or `std::optional<T>`, respectively; and
- if the attribute is `std::variant<>` or `std::optional<std::variant<>>`, the result becomes `none` instead.

Formation of containers in attributes

The rule for forming containers from non-containers is simple. You get a vector from any of the repeating parsers, like `+p`, `*p`, `repeat(3)[p]`, etc. The value type of the vector is *`ATTR`*`(p)`.

Another rule for sequence containers is that a value `x` and a container `c` containing elements of `x`'s type will form a single container. However, `x`'s type must be exactly the same as the elements in `c`. There is an exception to this in the special case for strings and characters noted above. For instance, consider the attribute of `char_ >> string("str")`. In the non-Unicode code path, `char_`'s attribute type is guaranteed to be `char`, so *`ATTR`*`(char_ >> string("str"))` is `std::string`. If you are parsing UTF-8 in the Unicode code path, `char_`'s attribute type is `char32_t`, and the special rule makes it also produce a `std::string`. Otherwise, the attribute for *`ATTR`*`(char_ >> string("str"))` would be `boost::parser::tuple<char32_t, std::string>`.

Again, there are no special rules for combining values and containers. Every combination results from an exact match, or fall into the string+character special case.

Another special case: `std::string` assignment

`std::string` can be assigned from a `char`. This is dumb. But, we're stuck with it. When you write a parser with a `char` attribute, and you try to parse it into a `std::string`, you've almost certainly made a mistake. More importantly, if you write this:

```
namespace bp = boost::parser;
std::string result;
auto b = bp::parse("3", bp::int_, bp::ws, result);
```

... you are even more likely to have made a mistake. Though this should work, because the assignment in `std::string s; s = 3;` is well-formed, Boost.Parser forbids it. If you write parsing code like the snippet above, you will get a static assertion. If you really do want to assign a `float` or whatever to a `std::string`, do it in a semantic action.

Examples of attributes generated by sequence and alternative parsers

In the table: `a` is a semantic action; and `p`, `p1`, `p2`, ... are parsers that generate attributes. Note that only `>>` is used here; `>` has the exact same attribute generation rules.

Table 26.10. Sequence and Alternative Combining Operations and Their Attributes

Expression	Attribute Type
`eps >> eps`	None.
`p >> eps`	*ATTR*(p)
`eps >> p`	*ATTR*(p)
`cu >> string("str")`	`std::string`
`string("str") >> cu`	`std::string`
`*cu >> string("str")`	`boost::parser::tuple<std::string, std::string>`
`string("str") >> *cu`	`boost::parser::tuple<std::string, std::string>`
`p >> p`	`boost::parser::tuple<`*ATTR*(p), *ATTR*(p)`>`
`*p >> p`	`std::string` if *ATTR*(p) is `char` or `char32_t`, otherwise `std::vector<`*ATTR*(p)`>`
`p >> *p`	`std::string` if *ATTR*(p) is `char` or `char32_t`, otherwise `std::vector<`*ATTR*(p)`>`
`*p >> -p`	`std::string` if *ATTR*(p) is `char` or `char32_t`, otherwise `std::vector<`*ATTR*(p)`>`
`-p >> *p`	`std::string` if *ATTR*(p) is `char` or `char32_t`, otherwise `std::vector<`*ATTR*(p)`>`
`string("str") >> -cu`	`std::string`
`-cu >> string("str")`	`std::string`
`!p1 | p2[a]`	None.
`p | p`	*ATTR*(p)
`p1 | p2`	`std::variant<`*ATTR*(p1), *ATTR*(p2)`>`
`p | eps`	`std::optional<`*ATTR*(p)`>`
`p1 | p2 | eps`	`std::optional<std::variant<`*ATTR*(p1), *ATTR*(p2)`>>`
`p1 | p2[a] | p3`	`std::optional<std::variant<`*ATTR*(p1), *ATTR*(p3)`>>`

Controlling attribute generation with merge[] and separate[]

As we saw in the previous Parsing into `struct`s and `class`es section, if you parse two strings in a row, you get two separate strings in the resulting attribute. The parser from that example was this:

```
namespace bp = boost::parser;
```

```
auto employee_parser = bp::lit("employee")
    >> '{'
    >> bp::int_ >> ','
    >> quoted_string >> ','
    >> quoted_string >> ','
    >> bp::double_
    >> '}';
```

`employee_parser`'s attribute is `boost::parser::tuple<int, std::string, std::string, double>`. The two `quoted_string` parsers produce `std::string` attributes, and those attributes are not combined. That is the default behavior, and it is just what we want for this case; we don't want the first and last name fields to be jammed together such that we can't tell where one name ends and the other begins. What if we were parsing some string that consisted of a prefix and a suffix, and the prefix and suffix were defined separately for reuse elsewhere?

```
namespace bp = boost::parser;
auto prefix = /* ... */;
auto suffix = /* ... */;
auto special_string = prefix >> suffix;
// Continue to use prefix and suffix to make other parsers....
```

In this case, we might want to use these separate parsers, but want `special_string` to produce a single `std::string` for its attribute. `merge[]` exists for this purpose.

```
namespace bp = boost::parser;
auto prefix = /* ... */;
auto suffix = /* ... */;
auto special_string = bp::merge[prefix >> suffix];
```

`merge[]` only applies to sequence parsers (like `p1 >> p2`), and forces all subparsers in the sequence parser to use the same variable for their attribute.

Another directive, `separate[]`, also applies only to sequence parsers, but does the opposite of `merge[]`. If forces all the attributes produced by the subparsers of the sequence parser to stay separate, even if they would have combined. For instance, consider this parser.

```
namespace bp = boost::parser;
auto string_and_char = +bp::char_('a') >> ' ' >> bp::cp;
```

`string_and_char` matches one or more `'a'`s, followed by some other character. As written above, `string_and_char` produces a `std::string`, and the final character is appended to the string, after all the `'a'`s. However, if you wanted to store the final character as a separate value, you would use `separate[]`.

```
namespace bp = boost::parser;
auto string_and_char = bp::separate[+bp::char_('a') >> ' ' >> bp::cp];
```

With this change, `string_and_char` produces the attribute `boost::parser::tuple<std::string, char32_t>`.

merge[] and separate[] in more detail

As mentioned previously, `merge[]` applies only to sequence parsers. All subparsers must have the same attribute, or produce no attribute at all. At least one subparser must produce an attribute. When you use `merge[]`, you create a *combining group*. Every parser in a combining group uses the same variable for its attribute. No parser in a combining group interacts with the attributes of any parsers outside of its combining group. Combining groups are disjoint; `merge[/*...*/] >> merge[/*...*/]` will produce a tuple of two attributes, not one.

`separate[]` also applies only to sequence parsers. When you use `separate[]`, you disable interaction of all the subparsers' attributes with adjacent attributes, whether they are inside or outside the `separate[]` directive; you force each subparser to have a separate attribute.

The rules for `merge[]` and `separate[]` overrule the steps of the algorithm described above for combining the attributes of a sequence parser. Consider an example.

```
namespace bp = boost::parser;
constexpr auto parser =
    bp::char_ >> bp::merge[(bp::string("abc") >> bp::char_ >> bp::char_) >> bp::string
    ("ghi")];
```

You might think that *ATTR*`(parser)` would be `bp::tuple<char, std::string>`. It is not. The parser above does not even compile. Since we created a merge group above, we disabled the default behavior in which the `char_` parsers would have collapsed into the `string` parser that preceded them. Since they are all treated as separate entities, and since they have different attribute types, the use of `merge[]` is an error.

Many directives create a new parser out of the parser they are given. `merge[]` and `separate[]` do not. Since they operate only on sequence parsers, all they do is create a copy of the sequence parser they are given. The `seq_parser` template has a template parameter `CombiningGroups`, and all `merge[]` and `separate[]` do is take a given `seq_parser` and create a copy of it with a different `CombiningGroups` template parameter. This means that `merge[]` and `separate[]` are can be ignored in `operator>>` expressions much like parentheses are. Consider an example.

```
namespace bp = boost::parser;
constexpr auto parser1 = bp::separate[bp::int_ >> bp::int_] >> bp::int_;
constexpr auto parser2 = bp::lexeme[bp::int_ >> ' ' >> bp::int_] >> bp::int_;
```

Note that `separate[]` is a no-op here; it's only being used this way for this example. These parsers have different attribute types. *ATTR*`(parser1)` is `boost::parser::tuple(int, int, int)`. *ATTR*`(parser2)` is `boost::parser::tuple(boost::parser::tuple(int, int), int)`. This is because `bp::lexeme[]` wraps its given parser in a new parser. `merge[]` does not. That's why, even though `parser1` and `parser2` look so structurally similar, they have different attributes.

`transform(f)[]`

`transform(f)[]` is a directive that transforms the attribute of a parser using the given function `f`. For example:

```
auto str_sum = [&](std::string const & s) {
    int retval = 0;
    for (auto ch : s) {
        retval += ch - '0';
```

```
    }
    return retval;
};

namespace bp = boost::parser;
constexpr auto parser = +bp::char_;
std::string str = "012345";

auto result = bp::parse(str, bp::transform(str_sum)[parser]);
assert(result);
assert(*result == 15);
static_assert(std::is_same_v<decltype(result), std::optional<int>>);
```

Here, we have a function `str_sum` that we use for `f`. It assumes each character in the given `std::string s` is a digit, and returns the sum of all the digits in `s`. Out parser `parser` would normally return a `std::string`. However, since `str_sum` returns a different type — `int` — that is the attribute type of the full parser, `bp::transform(by_value_`
as you can see from the `static_assert`.

As is the case with attributes all throughout Boost.Parser, the attribute passed to `f` will be moved. You can take it by `const &`, `&&`, or by value.

No distinction is made between parsers with and without an attribute, because there is a Regular special no-attribute type that is generated by parsers with no attribute. You may therefore write something like `transform(f)[eps]`, and Boost.Parser will happily call `f` with this special no-attribute type.

Other directives that affect attribute generation

`omit[p]` disables attribute generation for the parser `p`. `raw[p]` changes the attribute from *`ATTR(p)`* to a view that indicates the subrange of the input that was matched by `p`. `string_view[p]` is just like `raw[p]`, except that it produces `std::basic_string_view`s. See Directives for details.

1.1.19 The `parse()` API

There are multiple top-level parse functions. They have some things in common:

- They each return a value contextually convertible to `bool`.
- They each take at least a range to parse and a parser. The "range to parse" may be an iterator/sentinel pair or an single range object.
- They each require forward iterability of the range to parse.
- They each accept any range with a character element type. This means that they can each parse ranges of `char`, `wchar_t`, `char8_t`, `char16_t`, or `char32_t`.
- The overloads with `prefix_` in their name take an iterator/sentinel pair. For example `prefix_parse(first, last, p, ws)`, which parses the range `[first, last)`, advancing `first` as it goes. If the parse succeeds, the entire input may or may not have been matched. The value of `first` will indicate the last location within the input that `p` matched. The **whole** input was matched if and only if `first == last` after the call to `parse()`.
- When you call any of the range overloads of `parse()`, for example `parse(r, p, ws)`, `parse()` only indicates success if **all** of `r` was matched by `p`.

Note

`wchar_t` is an accepted value type for the input. Please note that this is interpreted as UTF-16 on MSVC, and UTF-32 everywhere else.

The overloads

There are eight overloads of `parse()` and `prefix_parse()` combined, because there are three either/or options in how you call them.

Iterator/sentinel versus range

You can call `prefix_parse()` with an iterator and sentinel that delimit a range of character values. For example:

```
namespace bp = boost::parser;
auto const p = /* some parser ... */;

char const * str_1 = /* ... */;
// Using null_sentinel, str_1 can point to three billion characters, and
// we can call prefix_parse() without having to find the end of the string first.
auto result_1 = bp::prefix_parse(str_1, bp::null_sentinel, p, bp::ws);

char str_2[] = /* ... */;
auto result_2 = bp::prefix_parse(std::begin(str_2), std::end(str_2), p, bp::ws);
```

The iterator/sentinel overloads can parse successfully without matching the entire input. You can tell if the entire input was matched by checking if `first == last` is true after `prefix_parse()` returns.

By contrast, you call `parse()` with a range of character values. When the range is a reference to an array of characters, any terminating `0` is ignored; this allows calls like `parse("str", p)` to work naturally.

```
namespace bp = boost::parser;
auto const p = /* some parser ... */;

std::u8string str_1 = "str";
auto result_1 = bp::parse(str_1, p, bp::ws);

// The null terminator is ignored.  This call parses s-t-r, not s-t-r-0.
auto result_2 = bp::parse(U"str", p, bp::ws);

char const * str_3 = "str";
auto result_3 = bp::parse(bp::null_term(str_3) | bp::as_utf16, p, bp::ws);
```

Since there is no way to indicate that `p` matches the input, but only a prefix of the input was matched, the range (non-iterator/sentinel) overloads of `parse()` indicate failure if the entire input is not matched.

With or without an attribute out-parameter

```
namespace bp = boost::parser;
auto const p = '"' >> *(bp::char_ - '"') >> '"';
char const * str = "\"two words\"" ;

std::string result_1;
bool const success = bp::parse(str, p, result_1);   // success is true; result_1 is "
    two words"
auto result_2 = bp::parse(str, p);                  // !!result_2 is true; *result_2
    is "two words"
```

When you call `parse()` **with** an attribute out-parameter and parser `p`, the expected type is **something like** *`ATTR(p)`*. It doesn't have to be exactly that; I'll explain in a bit. The return type is `bool`.

When you call `parse()` **without** an attribute out-parameter and parser `p`, the return type is `std::optional<ATTR(p)>`. Note that when *`ATTR(p)`* is itself an `optional`, the return type is `std::optional<std::optional<...>>`. Each of those optionals tells you something different. The outer one tells you whether the parse succeeded. If so, the parser was successful, but it still generates an attribute that is an `optional` — that's the inner one.

With or without a skipper

```
namespace bp = boost::parser;
auto const p = '"' >> *(bp::char_ - '"') >> '"';
char const * str = "\"two words\"" ;

auto result_1 = bp::parse(str, p);         // !!result_1 is true; *result_1 is "two
    words"
auto result_2 = bp::parse(str, p, bp::ws); // !!result_2 is true; *result_2 is "
    twowords"
```

Compatibility of attribute out-parameters

For any call to `parse()` that takes an attribute out-parameter, like `parse("str", p, bp::ws, out)`, the call is well-formed for a number of possible types of `out`; `decltype(out)` does not need to be exactly *`ATTR(p)`*.

For instance, this is well-formed code that does not abort (remember that the attribute type of `string()` is `std::string`):

```
namespace bp = boost::parser;
auto const p = bp::string("foo");

std::vector<char> result;
bool const success = bp::parse("foo", p, result);
assert(success && result == std::vector<char>({'f', 'o', 'o'}));
```

Even though `p` generates a `std::string` attribute, when it actually takes the data it generates and writes it into an attribute, it only assumes that the attribute is a `container` (see Concepts), not that it is some particular con-

tainer type. It will happily `insert()` into a `std::string` or a `std::vector<char>` all the same. `std::string` and `std::vector<char>` are both containers of `char`, but it will also insert into a container with a different element type. `p` just needs to be able to insert the elements it produces into the attribute-container. As long as an implicit conversion allows that to work, everything is fine:

```
namespace bp = boost::parser;
auto const p = bp::string("foo");

std::deque<int> result;
bool const success = bp::parse("foo", p, result);
assert(success && result == std::deque<int>({'f', 'o', 'o'}));
```

This works, too, even though it requires inserting elements from a generated sequence of `char32_t` into a container of `char` (remember that the attribute type of `+cp` is `std::vector<char32_t>`):

```
namespace bp = boost::parser;
auto const p = +bp::cp;

std::string result;
bool const success = bp::parse("foo", p, result);
assert(success && result == "foo");
```

This next example works as well, even though the change to a container is not at the top level. It is an element of the result tuple:

```
namespace bp = boost::parser;
auto const p = +(bp::cp - ' ') >> ' ' >> string("foo");

using attr_type = decltype(bp::parse(u8"", p));
static_assert(std::is_same_v<
              attr_type,
              std::optional<bp::tuple<std::string, std::string>>>);

using namespace bp::literals;

{
    // This is similar to attr_type, with the first std::string changed to a std::
    vector<int>.
    bp::tuple<std::vector<int>, std::string> result;
    bool const success = bp::parse(u8"rôle foo" | bp::as_utf8, p, result);
    assert(success);
    assert(bp::get(result, 0_c) == std::vector<int>({'r', U'ô', 'l', 'e'}));
    assert(bp::get(result, 1_c) == "foo");
}
{
    // This time, we have a std::vector<char> instead of a std::vector<int>.
    bp::tuple<std::vector<char>, std::string> result;
    bool const success = bp::parse(u8"rôle foo" | bp::as_utf8, p, result);
    assert(success);
    // The 4 code points "rôle" get transcoded to 5 UTF-8 code points to fit in the
    std::string.
    assert(bp::get(result, 0_c) == std::vector<char>({'r', (char)0xc3, (char)0xb4, 'l'
```

```
    , 'e'}));
    assert(bp::get(result, 1_c) == "foo");
}
```

As indicated in the inline comments, there are a couple of things to take away from this example:

- If you change an attribute out-param (such as `std::string` to `std::vector<int>`, or `std::vector<char32_t>` to `std::deque<int>`), the call to `parse()` will often still be well-formed.
- When changing out a container type, if both containers contain character values, the removed container's element type is `char32_t` (or `wchar_t` for non-MSVC builds), and the new container's element type is `char` or `char8_t`, Boost.Parser assumes that this is a UTF-32-to-UTF-8 conversion, and silently transcodes the data when inserting into the new container.

Let's look at a case where another simple-seeming type replacement does **not** work. First, the case that works:

```
namespace bp = boost::parser;
auto parser = -(bp::char_ % ',');
std::vector<int> result;
auto b = bp::parse("a, b", parser, bp::ws, result);
```

ATTR`(parser)` is `std::optional<std::string>`. Even though we pass a `std::vector<int>`, everything is fine. However, if we modify this case only sightly, so that the `std::optional<std::string>` is nested within the attribute, the code becomes ill-formed.

```
struct S
{
    std::vector<int> chars;
    int i;
};
namespace bp = boost::parser;
auto parser = -(bp::char_ % ',') >> bp::int_;
S result;
auto b = bp::parse("a, b 42", parser, bp::ws, result);
```

If we change `chars` to a `std::vector<char>`, the code is still ill-formed. Same if we change `chars` to a `std::string`. We must actually use `std::optional<std::string>` exactly to make the code well-formed again.

The reason the same looseness from the top-level parser does not apply to a nested parser is that, at some point in the code, the parser `-(bp::char_ % ',')` would try to assign a `std::optional<std::string>` — the element type of the attribute type it normally generates — to a `chars`. If there's no implicit conversion there, the code is ill-formed.

The take-away for this last example is that the ability to arbitrarily swap out data types within the type of the attribute you pass to `parse()` is very flexible, but is also limited to structurally simple cases. When we discuss `rules` in the next section, we'll see how this flexibility in the types of attributes can help when writing complicated parsers.

Those were examples of swapping out one container type for another. They make good examples because that is more likely to be surprising, and so it's getting lots of coverage here. You can also do much simpler things like parse using a `uint_`, and writing its attribute into a `double`. In general, you can swap any type `T` out of

the attribute, as long as the swap would not result in some ill-formed assignment within the parse.

Here is another example that also produces surprising results, for a different reason.

```
namespace bp = boost::parser;
constexpr auto parser = bp::char_('a') >> bp::char_('b') >> bp::char_('c') |
                        bp::char_('x') >> bp::char_('y') >> bp::char_('z');
std::string str = "abc";
bp::tuple<char, char, char> chars;
bool b = bp::parse(str, parser, chars);
assert(b);
assert(chars == bp::tuple('c', '\0', '\0'));
```

This looks wrong, but is expected behavior. At every stage of the parse that produces an attribute, Boost.Parser tries to assign that attribute to some part of the out-param attribute provided to `parse()`, if there is one. Note that *`ATTR(parser)`* is `std::string`, because each sequence parser is three `char_` parsers in a row, which forms a `std::string`; there are two such alternatives, so the overall attribute is also `std::string`. During the parse, when the first parser `bp::char_('a')` matches the input, it produces the attribute `'a'` and needs to assign it to its destination. Some logic inside the sequence parser indicates that this `'a'` contributes to the value in the `0`th position in the result tuple, if the result is being written into a tuple. Here, we passed a `bp::tuple<char, char, char>`, so it writes `'a'` into the first element. Each subsequent `char_` parser does the same thing, and writes over the first element. If we had passed a `std::string` as the out-param instead, the logic would have seen that the out-param attribute is a string, and would have appended `'a'` to it. Then each subsequent parser would have appended to the string.

Boost.Parser never looks at the arity of the tuple passed to `parse()` to see if there are too many or too few elements in it, compared to the expected attribute for the parser. In this case, there are two extra elements that are never touched. If there had been too few elements in the tuple, you would have seen a compilation error. The reason that Boost.Parser never does this kind of type-checking up front is that the loose assignment logic is spread out among the individual parsers; the top-level parse can determine what the expected attribute is, but not whether a passed attribute of another type is a suitable stand-in.

Compatibility of `variant` attribute out-parameters

The use of a variant in an out-param is compatible if the default attribute can be assigned to the `variant`. No other work is done to make the assignment compatible. For instance, this will work as you'd expect:

```
namespace bp = boost::parser;
std::variant<int, double> v;
auto b = bp::parse("42", bp::int_, v);
assert(b);
assert(v.index() == 0);
assert(std::get<0>(v) == 42);
```

Again, this works because `v = 42` is well-formed. However, other kinds of substitutions will not work. In particular, the `boost::parser::tuple` to aggregate or aggregate to `boost::parser::tuple` transformations will not work. Here's an example.

```
struct key_value
{
    int key;
```

```
    double value;
};

namespace bp = boost::parser;
std::variant<key_value, double> kv_or_d;
key_value kv;
bp::parse("42 13.0", bp::int_ >> bp::double_, kv);      // Ok.
bp::parse("42 13.0", bp::int_ >> bp::double_, kv_or_d); // Error: ill-formed!
```

In this case, it would be easy for Boost.Parser to look at the alternative types covered by the variant, and do a conversion. However, there are many cases in which there is no obviously correct variant alternative type, or in which the user might expect one variant alternative type and get another. Consider a couple of cases.

```
struct i_d { int i; double d; };
struct d_i { double d; int i; };
using v1 = std::variant<i_d, d_i>;

struct i_s { int i; short s; };
struct d_d { double d1; double d2; };
using v2 = std::variant<i_s, d_d>;

using tup_t = boost::parser::tuple<short, short>;
```

If we have a parser that produces a `tup_t`, and we have a `v1` attribute out-param, the correct variant alternative type clearly does not exist — this case is ambiguous, and anyone can see that neither variant alternative is a better match. If we were assigning a `tup_t` to `v2`, it's even worse. The same ambiguity exists, but to the user, `i_s` is clearly "closer" than `d_d`.

So, Boost.Parser only does assignment. If some parser `P` generates a default attribute that is not assignable to a variant alternative that you want to assign it to, you can just create a `rule` that creates either an exact variant alternative type, or the variant itself, and use `P` as your rule's parser.

Unicode versus non-Unicode parsing

A call to `parse()` either considers the entire input to be in a UTF format (UTF-8, UTF-16, or UTF-32), or it considers the entire input to be in some unknown encoding. Here is how it deduces which case the call falls under:

- If the range is a sequence of `char8_t`, or if the input is a `boost::parser::utf8_view`, the input is UTF-8.
- Otherwise, if the value type of the range is `char`, the input is in an unknown encoding.
- Otherwise, the input is in a UTF encoding.

Tip

if you want to want to parse in ASCII-only mode, or in some other non-Unicode encoding, use only sequences of `char`, like `std::string` or `char const *`.

Tip

If you want to ensure all input is parsed as Unicode, pass the input range `r` as `r | boost::parser::as_utf32` — that's the first thing that happens to it inside `parse()` in the Unicode parsing path anyway.

Note

Since passing `boost::parser::utfN_view` is a special case, and since a sequence of `char`s `r` is otherwise considered an unknown encoding, `boost::parser::parse(r | boost::parser::as_utf8, p)` treats `r` as UTF-8, whereas `boost::parser::parse(r, p)` does not.

The `trace_mode` parameter to parse()

Debugging parsers is notoriously difficult once they reach a certain size. To get a verbose trace of your parse, pass `boost::parser::trace::on` as the final parameter to `parse()`. It will show you the current parser being matched, the next few characters to be parsed, and any attributes generated. See the Error Handling and Debugging section of the tutorial for details.

Globals and error handlers

Each call to `parse()` can optionally have a globals object associated with it. To use a particular globals object with you parser, you call `with_globals()` to create a new parser with the globals object in it:

```
struct globals_t
{
    int foo;
    std::string bar;
};
auto const parser = /* ... */;
globals_t globals{42, "yay"};
auto result = boost::parser::parse("str", boost::parser::with_globals(parser, globals)
    );
```

Every semantic action within that call to `parse()` can access the same `globals_t` object using `_globals(ctx)`.

The default error handler is great for most needs, but if you want to change it, you can do so by creating a new parser with a call to `with_error_handler()`:

```
auto const parser = /* ... */;
my_error_handler error_handler;
auto result = boost::parser::parse("str", boost::parser::with_error_handler(parser,
    error_handler));
```

Tip

If your parsing environment does not allow you to report errors to a terminal, you may want to use `callback_error_handler` instead of the default error handler.

Important

Globals and the error handler are ignored, if present, on any parser except the top-level parser.

1.1.20 More About Rules

In the earlier page about `rules` (Rule Parsers), I described `rules` as being analogous to functions. `rules` are, at base, organizational. Here are the common use cases for `rules`. Use a `rule` if you want to:

- fix the attribute type produced by a parser to something other than the default;
- create a parser that produces useful diagnostic text;
- create a recursive rule (more on this below);
- create a set of mutually-recursive parsers;
- do callback parsing.

Let's look at the use cases in detail.

Fixing the attribute type

We saw in the previous section how `parse()` is flexible in what types it will accept as attribute out-parameters. Here's another example.

```
namespace bp = boost::parser;
auto result = bp::parse(input, bp::int % ',', result);
```

`result` can be one of many different types. It could be `std::vector<int>`. It could be `std::set<long long>`. It could be a lot of things. Often, this is a very useful property; if you had to rewrite all of your parser logic because you changed the desired container in some part of your attribute from a `std::vector` to a `std::deque`, that would be annoying. However, that flexibility comes at the cost of type checking. If you want to write a parser that **always** produces exactly a `std::vector<unsigned int>` **and no other type**, you also probably want a compilation error if you accidentally pass that parser a `std::set<unsigned int>` attribute instead. There is no way with a plain parser to enforce that its attribute type may only ever be a single, fixed type.

Fortunately, `rules` allow you to write a parser that has a fixed attribute type. Every rule has a specific attribute type, provided as a template parameter. If one is not specified, the rule has no attribute. The fact that the attribute is a specific type allows you to remove attribute flexibility. For instance, say we have a rule defined like this:

```
bp::rule<struct doubles, std::vector<double>> doubles = "doubles";
auto const doubles_def = bp::double_ % ',';
```

```
BOOST_PARSER_DEFINE_RULES(doubles);
```

You can then use it in a call to `parse()`, and `parse()` will return a `std::optional<std::vector<double>>`:

```
auto const result = bp::parse(input, doubles, bp::ws);
```

If you call `parse()` with an attribute out-parameter, it must be exactly `std::vector<double>`:

```
std::vector<double> vec_result;
bp::parse(input, doubles, bp::ws, vec_result); // Ok.
std::deque<double> deque_result;
bp::parse(input, doubles, bp::ws, deque_result); // Ill-formed!
```

If we wanted to use a `std::deque<double>` as the attribute type of our rule:

```
// Attribute changed to std::deque<double>.
bp::rule<struct doubles, std::deque<double>> doubles = "doubles";
auto const doubles_def = bp::double_ % ',';
BOOST_PARSER_DEFINE_RULES(doubles);

int main()
{
    std::deque<double> deque_result;
    bp::parse(input, doubles, bp::ws, deque_result); // Ok.
}
```

The take-away here is that the attribute flexibility is still available, but only **within** the rule — the parser `bp::double_ % ','` can parse into a `std::vector<double>` or a `std::deque<double>`, but the rule `doubles` must parse into only the exact attribute it was declared to generate.

The reason for this is that, inside the rule parsing implementation, there is code something like this:

```
using attr_t = ATTR(doubles_def);
attr_t attr;
parse(first, last, parser, attr);
attribute_out_param = std::move(attr);
```

Where `attribute_out_param` is the attribute out-parameter we pass to `parse()`. If that final move assignment is ill-formed, the call to `parse()` is too.

You can also use rules to exploit attribute flexibility. Even though a rule reduces the flexibility of attributes it can generate, the fact that it is so easy to write a new rule means that we can use rules themselves to get the attribute flexibility we want across our code:

```
namespace bp = boost::parser;

// We only need to write the definition once...
auto const generic_doubles_def = bp::double_ % ',';

bp::rule<struct vec_doubles, std::vector<double>> vec_doubles = "vec_doubles";
auto const & vec_doubles_def = generic_doubles_def; // ... and re-use it,
BOOST_PARSER_DEFINE_RULES(vec_doubles);
```

```
// Attribute changed to std::deque<double>.
bp::rule<struct deque_doubles, std::deque<double>> deque_doubles = "deque_doubles";
auto const & deque_doubles_def = generic_doubles_def; // ... and re-use it again.
BOOST_PARSER_DEFINE_RULES(deque_doubles);
```

Now we have one of each, and we did not have to copy any parsing logic that would have to be maintained in two places.

Sometimes, you need to create a rule to enforce a certain attribute type, but the rule's attribute is not constructible from its parser's attribute. When that happens, you'll need to write a semantic action.

```
struct type_t
{
    type_t() = default;
    explicit type_t(double x) : x_(x) {}
    // etc.

    double x_;
};

namespace bp = boost::parser;

auto doubles_to_type = [](auto & ctx) {
    using namespace bp::literals;
    _val(ctx) = type_t(_attr(ctx)[0_c] * _attr(ctx)[1_c]);
};

bp::rule<struct type_tag, type_t> type = "type";
auto const type_def = (bp::double_ >> bp::double_)[doubles_to_type];
BOOST_PARSER_DEFINE_RULES(type);
```

For a rule `R` and its parser `P`, we do not need to write such a semantic action if:

- *`ATTR(R)`* is an aggregate, and *`ATTR(P)`* is a compatible tuple;

- *`ATTR(R)`* is a tuple, and *`ATTR(P)`* is a compatible aggregate;

- *`ATTR(R)`* is a non-aggregate class type `C`, and *`ATTR(P)`* is a tuple whose elements can be used to construct `C`; or

- *`ATTR(R)`* and *`ATTR(P)`* are compatible types.

The notion of "compatible" is defined in The `parse()` API.

Creating a parser for better diagnostics

Each `rule` has associated diagnostic text that Boost.Parser can use for failures of that rule. This is useful when the parse reaches a parse failure at an expectation point (see Expectation points). Let's say you have the following code defined somewhere.

```
namespace bp = boost::parser;

bp::rule<struct value_tag> value =
    "an integer, or a list of integers in braces";

auto const ints = '{' > (value % ',') > '}';
auto const value_def = bp::int_ | ints;

BOOST_PARSER_DEFINE_RULES(value);
```

Notice the two expectation points. One before `(value % ',')`, one before the final `'}'`. Later, you call parse in some input:

```
bp::parse("{ 4, 5 a", value, bp::ws);
```

This runs should of the second expectation point, and produces output like this:

```
1:7: error: Expected '}' here:
{ 4, 5 a
       ^
```

That's a pretty good error message. Here's what it looks like if we violate the earlier expectation:

```
bp::parse("{ }", value, bp::ws);

1:2: error: Expected an integer, or a list of integers in braces % ',' here:
{ }
  ^
```

Not nearly as nice. The problem is that the expectation is on `(value % ',')`. So, even thought we gave `value` reasonable dianostic text, we put the text on the wrong thing. We can introduce a new rule to put the diagnstic text in the right place.

```
namespace bp = boost::parser;

bp::rule<struct value_tag> value =
    "an integer, or a list of integers in braces";
bp::rule<struct comma_values_tag> comma_values =
    "a comma-delimited list of integers";

auto const ints = '{' > comma_values > '}';
auto const value_def = bp::int_ | ints;
auto const comma_values_def = (value % ',');

BOOST_PARSER_DEFINE_RULES(value, comma_values);
```

Now when we call `bp::parse("{ }", value, bp::ws)` we get a much better message:

```
1:2: error: Expected a comma-delimited list of integers here:
{ }
  ^
```

The `rule value` might be useful elsewhere in our code, perhaps in another parser. It's diagnostic text is appropriate for those other potential uses.

Recursive rules

It's pretty common to see grammars that include recursive rules. Consider this EBNF rule for balanced parentheses:

```
<parens> ::= "" | ( "(" <parens> ")" )
```

We can try to write this using Boost.Parser like this:

```
namespace bp = boost::parser;
auto const parens = '(' >> parens >> ')' | bp::eps;
```

We had to put the `bp::eps` second, because Boost.Parser's parsing algorithm is greedy. Otherwise, it's just a straight transliteration. Unfortunately, it does not work. The code is ill-formed because you can't define a variable in terms of itself. Well you can, but nothing good comes of it. If we instead make the parser in terms of a forward-declared `rule`, it works.

```
namespace bp = boost::parser;
bp::rule<struct parens_tag> parens = "matched parentheses";
auto const parens_def = '(' >> parens > ')' | bp::eps;
BOOST_PARSER_DEFINE_RULES(parens);
```

Later, if we use it to parse, it does what we want.

```
assert(bp::parse("(((()))))", parens, bp::ws));
```

When it fails, it even produces nice diagnostics.

```
bp::parse("((((()))", parens, bp::ws);
```

```
1:7: error: Expected ')' here (end of input):
((((()))
       ^
```

Recursive `rules` work differently from other parsers in one way: when re-entering the rule recursively, only the attribute variable (`_attr(ctx)` in your semantic actions) is unique to that instance of the rule. All the other state of the uppermost instance of that rule is shared. This includes the value of the rule (`_val(ctx)`), and the locals and parameters to the rule. In other words, `_val(ctx)` returns a reference to the **same object** in every instance of a recursive `rule`. This is because each instance of the rule needs a place to put the attribute it generates from its parse. However, we only want a single return value for the uppermost rule; if each instance had a separate value in `_val(ctx)`, then it would be impossible to build up the result of a recursive rule step by step during in the evaluation of the recursive instantiations.

Also, consider this rule:

```
namespace bp = boost::parser;
bp::rule<struct ints_tag, std::vector<int>> ints = "ints";
auto const ints_def = bp::int_ >> ints | bp::eps;
```

What is the default attribute type for ints_def? It sure looks like `std::optional<std::vector<int>>`. Inside the evaluation of `ints`, Boost.Parser must evaluate `ints_def`, and then produce a `std::vector<int>` — the return type of `ints` — from it. How? How do you turn a `std::optional<std::vector<int>>` into a `std::vector<int>`? To a human, it seems obvious, but the metaprogramming that properly handles this simple example and the general case is certainly beyond me.

Boost.Parser has a specific semantic for what consitutes a recursive rule. Each rule has a tag type associated with it, and if Boost.Parser enters a rule with a certain tag `Tag`, and the currently-evaluating rule (if there is one) also has the tag `Tag`, then rule instance being entered is considered to be a recursion. No other situations are considered recursion. In particular, if you have rules `Ra` and `Rb`, and `Ra` uses `Rb`, which in turn used `Ra`, the second use of `Ra` is not considered recursion. `Ra` and `Rb` are of course mutually recursive, but neither is considered a "recursive rule" for purposes of getting a unique value, locals, and parameters.

Mutually-recursive rules

One of the advantages of using rules is that you can declare all your rules up front and then use them immediately afterward. This lets you make rules that use each other without introducing cycles:

```
namespace bp = boost::parser;

// Assume we have some polymorphic type that can be an object/dictionary,
// array, string, or int, called `value_type`.

bp::rule<class string, std::string> const string = "string";
bp::rule<class object_element, bp::tuple<std::string, value_type>> const
    object_element = "object-element";
bp::rule<class object, value_type> const object = "object";
bp::rule<class array, value_type> const array = "array";
bp::rule<class value_tag, value_type> const value = "value";

auto const string_def = bp::lexeme['"' >> *(bp::char_ - '"') > '"'];
auto const object_element_def = string > ':' > value;
auto const object_def = '{'_l >> -(object_element % ',') > '}';
auto const array_def = '['_l >> -(value % ',') > ']';
auto const value_def = bp::int_ | bp::bool_ | string | array | object;

BOOST_PARSER_DEFINE_RULES(string, object_element, object, array, value);
```

Here we have a parser for a Javascript-value-like type `value_type`. `value_type` may be an array, which itself may contain other arrays, objects, strings, etc. Since we need to be able to parse objects within arrays and vice versa, we need each of those two parsers to be able to refer to each other.

Callback parsing

Only `rules` can be callback parsers, so if you want to get attributes supplied to you via callbacks instead of somewhere in the middle of a giant attribute that represents the whole parse result, you need to use `rules`. See Parsing JSON With Callbacks for an extended example of callback parsing.

Accessors available in semantic actions on rules

_val()

Inside all of a rule's semantic actions, the expression `_val(ctx)` is a reference to the attribute that the rule generates. This can be useful when you want subparsers to build up the attribute in a specific way:

```
namespace bp = boost::parser;
using namespace bp::literals;

bp::rule<class ints, std::vector<int>> const ints = "ints";
auto twenty_zeros = [](auto & ctx) { _val(ctx).resize(20, 0); };
auto push_back = [](auto & ctx) { _val(ctx).push_back(_attr(ctx)); };
auto const ints_def = "20-zeros"_l[twenty_zeros] | +bp::int_[push_back];
BOOST_PARSER_DEFINE_RULES(ints);
```

Tip

That's just an example. It's almost always better to do things without using semantic actions. We could have instead written `ints_def` as `"20-zeros" >> bp::attr(std::vector<int>(20)) | +bp::int_`, which has the same semantics, is a lot easier to read, and is a lot less code.

Locals

The `rule` template takes another template parameter we have not discussed yet. You can pass a third parameter `LocalState` to `rule`, which will be defaulted csontructed by the `rule`, and made available within semantic actions used in the rule as `_locals(ctx)`. This gives your rule some local state, if it needs it. The type of `LocalState` can be anything regular. It could be a single value, a struct containing multiple values, or a tuple, among others.

```
struct foo_locals
{
    char first_value = 0;
};

namespace bp = boost::parser;

bp::rule<class foo, int, foo_locals> const foo = "foo";

auto record_first = [](auto & ctx) { _locals(ctx).first_value = _attr(ctx); }
auto check_against_first = [](auto & ctx) {
    char const first = _locals(ctx).first_value;
    char const attr = _attr(ctx);
    if (attr == first)
        _pass(ctx) = false;
    _val(ctx) = (int(first) << 8) | int(attr);
};
```

```
auto const foo_def = bp::cu[record_first] >> bp::cu[check_against_first];
BOOST_PARSER_DEFINE_RULES(foo);
```

`foo` matches the input if it can match two elements of the input in a row, but only if they are not the same value. Without locals, it's a lot harder to write parsers that have to track state as they parse.

Parameters

Sometimes, it is convenient to parameterize parsers. Consider these parsing rules from the YAML 1.2 spec:

```
[80]
s-separate(n,BLOCK-OUT) ::= s-separate-lines(n)
s-separate(n,BLOCK-IN)  ::= s-separate-lines(n)
s-separate(n,FLOW-OUT)  ::= s-separate-lines(n)
s-separate(n,FLOW-IN)   ::= s-separate-lines(n)
s-separate(n,BLOCK-KEY) ::= s-separate-in-line
s-separate(n,FLOW-KEY)  ::= s-separate-in-line

[136]
in-flow(n,FLOW-OUT)  ::= ns-s-flow-seq-entries(n,FLOW-IN)
in-flow(n,FLOW-IN)   ::= ns-s-flow-seq-entries(n,FLOW-IN)
in-flow(n,BLOCK-KEY) ::= ns-s-flow-seq-entries(n,FLOW-KEY)
in-flow(n,FLOW-KEY)  ::= ns-s-flow-seq-entries(n,FLOW-KEY)

[137]
c-flow-sequence(n,c) ::= “[” s-separate(n,c)? in-flow(c)? “]”
```

YAML [137] says that the parsing should proceed into two YAML subrules, both of which have these `n` and `c` parameters. It is certainly possible to transliterate these YAML parsing rules to something that uses unparameterized Boost.Parser `rules`, but it is quite painful to do so. It is better to use a parameterized rule.

You give parameters to a `rule` by calling its `with()` member. The values you pass to `with()` are used to create a `boost::parser::tuple` that is available in semantic actions attached to the rule, using `_params(ctx)`.

Passing parameters to `rules` like this allows you to easily write parsers that change the way they parse depending on contextual data that they have already parsed.

Here is an implementation of YAML [137]. It also implements the two YAML rules used directly by [137], rules [136] and [80]. The rules that **those** rules use are also represented below, but are implemented using only `eps`, so that I don't have to repeat too much of the (very large) YAML spec.

```
namespace bp = boost::parser;

// A type to represent the YAML parse context.
enum class context {
    block_in,
    block_out,
    block_key,
    flow_in,
```

```
    flow_out,
    flow_key
};

// A YAML value; no need to fill it in for this example.
struct value
{
    // ...
};

// YAML [66], just stubbed in here.
auto const s_separate_in_line = bp::eps;

// YAML [137].
bp::rule<struct c_flow_seq_tag, value> c_flow_sequence = "c-flow-sequence";
// YAML [80].
bp::rule<struct s_separate_tag> s_separate = "s-separate";
// YAML [136].
bp::rule<struct in_flow_tag, value> in_flow = "in-flow";
// YAML [138]; just eps below.
bp::rule<struct ns_s_flow_seq_entries_tag, value> ns_s_flow_seq_entries =
    "ns-s-flow-seq-entries";
// YAML [81]; just eps below.
bp::rule<struct s_separate_lines_tag> s_separate_lines = "s-separate-lines";

// Parser for YAML [137].
auto const c_flow_sequence_def =
    '[' >>
    -s_separate.with(bp::_p<0>, bp::_p<1>) >>
    -in_flow.with(bp::_p<0>, bp::_p<1>) >>
    ']';
// Parser for YAML [80].
auto const s_separate_def = bp::switch_(bp::_p<1>)
    (context::block_out, s_separate_lines.with(bp::_p<0>))
    (context::block_in, s_separate_lines.with(bp::_p<0>))
    (context::flow_out, s_separate_lines.with(bp::_p<0>))
    (context::flow_in, s_separate_lines.with(bp::_p<0>))
    (context::block_key, s_separate_in_line)
    (context::flow_key, s_separate_in_line);
// Parser for YAML [136].
auto const in_flow_def = bp::switch_(bp::_p<1>)
    (context::flow_out, ns_s_flow_seq_entries.with(bp::_p<0>, context::flow_in))
    (context::flow_in, ns_s_flow_seq_entries.with(bp::_p<0>, context::flow_in))
    (context::block_out, ns_s_flow_seq_entries.with(bp::_p<0>, context::flow_key))
    (context::flow_key, ns_s_flow_seq_entries.with(bp::_p<0>, context::flow_key));

auto const ns_s_flow_seq_entries_def = bp::eps;
auto const s_separate_lines_def = bp::eps;

BOOST_PARSER_DEFINE_RULES(
    c_flow_sequence,
```

```
    s_separate,
    in_flow,
    ns_s_flow_seq_entries,
    s_separate_lines);
```

YAML [137] (`c_flow_sequence`) parses a list. The list may be empty, and must be surrounded by brackets, as you see here. But, depending on the current YAML context (the `c` parameter to [137]), we may require certain spacing to be matched by `s-separate`, and how sub-parser `in-flow` behaves also depends on the current context.

In `s_separate` above, we parse differently based on the value of `c`. This is done above by using the value of the second parameter to `s_separate` in a switch-parser. The second parameter is looked up by using `_p` as a parse argument.

`in_flow` does something similar. Note that `in_flow` calls its subrule by passing its first parameter, but using a fixed value for the second value. `s_separate` only passes its `n` parameter conditionally. The point is that a rule can be used with and without `.with()`, and that you can pass constants or parse arguments to `.with()`.

With those rules defined, we could write a unit test for YAML [137] like this:

```
auto const test_parser = c_flow_sequence.with(4, context::block_out);
auto result = bp::parse("[]", test_parser);
assert(result);
```

You could extend this with tests for different values of `n` and `c`. Obviously, in real tests, you parse actual contents inside the `"[]"`, if the other rules were implemented, like [138].

The _p variable template

Getting at one of a rule's arguments and passing it as an argument to another parser can be very verbose. `_p` is a variable template that allows you to refer to the `n`th argument to the current rule, so that you can, in turn, pass it to one of the rule's subparsers. Using this, `foo_def` above can be rewritten as:

```
auto const foo_def = bp::repeat(bp::_p<0>)[' '_l];
```

Using `_p` can prevent you from having to write a bunch of lambdas that get each get an argument out of the parse context using `_params(ctx)[0_c]` or similar.

Note that `_p` is a parse argument (see The Parsers And Their Uses), meaning that it is an invocable that takes the context as its only parameter. If you want to use it inside a semantic action, you have to call it.

Special forms of semantic actions usable within a rule

Semantic actions in this tutorial are usually of the signature `void (auto & ctx)`. That is, they take a context by reference, and return nothing. If they were to return something, that something would just get dropped on the floor.

It is a pretty common pattern to create a rule in order to get a certain kind of value out of a parser, when you don't normally get it automatically. If I want to parse an `int`, `int_` does that, and the thing that I parsed is also

the desired attribute. If I parse an `int` followed by a `double`, I get a `boost::parser::tuple` containing one of each. But what if I don't want those two values, but some function of those two values? I probably write something like this.

```
struct obj_t { /* ... */ };
obj_t to_obj(int i, double d) { /* ... */ }

namespace bp = boost::parser;
bp::rule<struct obj_tag, obj_t> obj = "obj";
auto make_obj = [](auto & ctx) {
    using boost::hana::literals;
    _val(ctx) = to_obj(_attr(ctx)[0_c], _attr(ctx)[1_c]);
};
constexpr auto obj_def = (bp::int_ >> bp::double_)[make_obj];
```

That's fine, if a little verbose. However, you can also do this instead:

```
namespace bp = boost::parser;
bp::rule<struct obj_tag, obj_t> obj = "obj";
auto make_obj = [](auto & ctx) {
    using boost::hana::literals;
    return to_obj(_attr(ctx)[0_c], _attr(ctx)[1_c]);
};
constexpr auto obj_def = (bp::int_ >> bp::double_)[make_obj];
```

Above, we return the value from a semantic action, and the returned value gets assigned to `_val(ctx)`.

Finally, you can provide a function that takes the individual elements of the attribute (if it's a tuple), and returns the value to assign to `_val(ctx)`:

```
namespace bp = boost::parser;
bp::rule<struct obj_tag, obj_t> obj = "obj";
constexpr auto obj_def = (bp::int_ >> bp::double_)[to_obj];
```

More formally, within a rule, the use of a semantic action is determined as follows. Assume we have a function `APPLY` that calls a function with the elements of a tuple, like `std::apply`. For some context `ctx`, semantic action `action`, and attribute `attr`, `action` is used like this:

- `_val(ctx) = APPLY(action, std::move(attr))`, if that is well-formed, and `attr` is a tuple of size 2 or larger;

- otherwise, `_val(ctx) = action(ctx)`, if that is well-formed;

- otherwise, `action(ctx)`.

The first case does not pass the context to the action at all. The last case is the normal use of semantic actions outside of rules.

1.1.21 Algorithms and Views That Use Parsers

Unless otherwise noted, all the algorithms and views are constrained very much like the way the `parse()` overloads are. The kinds of ranges, parsers, etc., that they accept are the same.

boost::parser::search()

As shown in The `parse()` API, the two patterns of parsing in Boost.Parser are whole-parse and prefix-parse. When you want to find something in the middle of the range being parsed, there's no `parse` API for that. You can of course make a simple parser that skips everything before what you're looking for.

```
namespace bp = boost::parser;
constexpr auto parser = /* ... */;
constexpr auto middle_parser = bp::omit[*(bp::char_ - parser)] >> parser;
```

`middle_parser` will skip over everything, one `char_` at a time, as long as the next `char_` is not the beginning of a successful match of `parser`. After this, control passes to `parser` itself. Ok, so that's not too hard to write. If you need to parse something from the middle in order to generate attributes, this is what you should use.

However, it often turns out you only need to find some subrange in the parsed range. In these cases, it would be nice to turn this into a proper algorithm in the pattern of the ones in `std::ranges`, since that's more idiomatic. `boost::parser::search()` is that algorithm. It has very similar semantics to `std::ranges::search`, except that it searches not for a match to an exact subrange, but to a match with the given parser. Like `std::ranges::search()`, it returns a subrange (`boost::parser::subrange` in C++17, `std::ranges::subrange` in C++20 and later).

```
namespace bp = boost::parser;
auto result = bp::search("aaXYZq", bp::lit("XYZ"), bp::ws);
assert(!result.empty());
assert(std::string_view(result.begin(), result.end() - result.begin()) == "XYZ");
```

Since `boost::parser::search()` returns a subrange, whatever parser you give it produces no attribute. I wrote `bp::lit("XYZ")` above; if I had written `bp::string("XYZ")` instead, the result (and lack of `std::string` construction) would not change.

As you can see above, one aspect of `boost::parser::search()` differs intentionally from the conventions of the `std::ranges` algorithms — it accepts C-style strings, treating them as if they were proper ranges.

Also, `boost::parser::search()` knows how to accommodate your iterator type. You can pass the C-style string `"aaXYZq"` as in the example above, or `"aaXYZq" | bp::as_utf32`, or `"aaXYZq" | bp::as_utf8`, or even `"aaXYZq" | bp::as_utf16`, and it will return a subrange whose iterators are the type that you passed as input, even though internally the iterator type might be something different (a UTF-8 -> UTF-32 transcoding iterator in Unicode parsing, as with all the `| bp::as_utfN` examples above). As long as you pass a range to be parsed whose value type is `char`, `char8_t`, `char32_t`, or that is adapted using some combination of `as_utfN` adaptors, this accommodation will operate correctly.

`boost::parser::search()` has multiple overloads. You can pass a range or an iterator/sentinel pair, and you can pass a skip parser or not. That's four overloads. Also, all four overloads take an optional `boost::parser::trace` parameter at the end. This is really handy for investigating why you're not finding something in the input that you expected to.

boost::parser::search_all

`boost::parser::search_all` creates `boost::parser::search_all_views`. `boost::parser::search_all_view` is a `std::views`-style view. It produces a range of subranges. Each subrange it produces is the next match of

the given parser in the parsed range.

```
namespace bp = boost::parser;
auto r = "XYZaaXYZbaabaXYZXYZ" | bp::search_all(bp::lit("XYZ"));
int count = 0;
// Prints XYZ XYZ XYZ XYZ.
for (auto subrange : r) {
    std::cout << std::string_view(subrange.begin(), subrange.end() - subrange.begin())
     << " ";
    ++count;
}
std::cout << "\n";
assert(count == 4);
```

All the details called out in the subsection on `boost::parser::search()` above apply to `boost::parser::search_all`: its parser produces no attributes; it accepts C-style strings as if they were ranges; and it knows how to get from the internally-used iterator type back to the given iterator type, in typical cases.

`boost::parser::search_all` can be called with, and `boost::parser::search_all_view` can be constructed with, a skip parser or not, and you can always pass `boost::parser::trace` at the end of any of their overloads.

boost::parser::split

`boost::parser::split` creates `boost::parser::split_views`. `boost::parser::split_view` is a `std::views`-style view. It produces a range of subranges of the parsed range split on matches of the given parser. You can think of `boost::parser::split_view` as being the complement of `boost::parser::search_all_view`, in that `boost::parser::split_view` produces the subranges between the subranges produced by `boost::parser::search_all`. `boost::parser::split_view` has very similar semantics to `std::views::split_view`. Just like `std::views::split_view`, `boost::parser::split_view` will produce empty ranges between the beginning/end of the parsed range and an adjacent match, or between adjacent matches.

```
namespace bp = boost::parser;
auto r = "XYZaaXYZbaabaXYZXYZ" | bp::split(bp::lit("XYZ"));
int count = 0;
// Prints '' 'aa' 'baaba' '' ''.
for (auto subrange : r) {
    std::cout << "'" << std::string_view(subrange.begin(), subrange.end() - subrange.
   begin()) << "' ";
    ++count;
}
std::cout << "\n";
assert(count == 5);
```

All the details called out in the subsection on `boost::parser::search()` above apply to `boost::parser::split`: its parser produces no attributes; it accepts C-style strings as if they were ranges; and it knows how to get from the internally-used iterator type back to the given iterator type, in typical cases.

`boost::parser::split` can be called with, and `boost::parser::split_view` can be constructed with, a skip parser or not, and you can always pass `boost::parser::trace` at the end of any of their overloads.

boost::parser::replace

Important

`boost::parser::replace` and `boost::parser::replace_view` are not available on MSVC in C++17 mode.

`boost::parser::replace` creates `boost::parser::replace_views`. `boost::parser::replace_view` is a `std::views`-style view. It produces a range of subranges from the parsed range `r` and the given replacement range `replacement`. Wherever in the parsed range a match to the given parser `parser` is found, `replacement` is the subrange produced. Each subrange of `r` that does not match `parser` is produced as a subrange as well. The subranges are produced in the order in which they occur in `r`. Unlike `boost::parser::split_view`, `boost::parser::replace_view` does not produce empty subranges, unless `replacement` is empty.

```
namespace bp = boost::parser;
auto card_number = bp::int_ >> bp::repeat(3)['-' >> bp::int_];
auto rng = "My credit card number is 1234-5678-9012-3456." | bp::replace(card_number,
    "XXXX-XXXX-XXXX-XXXX");
int count = 0;
// Prints My credit card number is XXXX-XXXX-XXXX-XXXX.
for (auto subrange : rng) {
    std::cout << std::string_view(subrange.begin(), subrange.end() - subrange.begin())
    ;
    ++count;
}
std::cout << "\n";
assert(count == 3);
```

If the iterator types `Ir` and `Ireplacement` for the `r` and `replacement` ranges passed are identical (as in the example above), the iterator type for the subranges produced is `Ir`. If they are different, an implementation-defined type is used for the iterator. This type is the moral equivalent of a `std::variant<Ir, Ireplacement>`. This works as long as `Ir` and `Ireplacement` are compatible. To be compatible, they must have common reference, value, and rvalue reference types, as determined by `std::common_type_t`. One advantage to this scheme is that the range of subranges represented by `boost::parser::replace_view` is easily joined back into a single range.

```
namespace bp = boost::parser;
auto card_number = bp::int_ >> bp::repeat(3)['-' >> bp::int_];
auto rng = "My credit card number is 1234-5678-9012-3456." | bp::replace(card_number,
    "XXXX-XXXX-XXXX-XXXX") | std::views::join;
std::string replace_result;
for (auto ch : rng) {
    replace_result.push_back(ch);
}
assert(replace_result == "My credit card number is XXXX-XXXX-XXXX-XXXX.");
```

Note that we could **not** have written `std::string replace_result(r.begin(), r.end())`. This is ill-formed because the `std::string` range constructor takes two iterators of the same type, but `decltype(rng.end())` is a sentinel type different from `decltype(rng.begin())`.

Though the ranges `r` and `replacement` can both be C-style strings, `boost::parser::replace_view` must know the end of `replacement` before it does any work. This is because the subranges produced are all common ranges, and so if `replacement` is not, a common range must be formed from it. If you expect to pass very long C-style strings to `boost::parser::replace` and not pay to see the end until the range is used, don't.

`ReplacementV` is constrained almost exactly the same as `V`. `V` must model `parsable_range` and `std::ranges::viewable_r` `ReplacementV` is the same, except that it can also be a `std::ranges::input_range`, whereas `V` must be a `std::ranges::forward_range`.

You may wonder what happens when you pass a UTF-N range for `r`, and a UTF-M range for `replacement`. What happens in this case is silent transcoding of `replacement` from UTF-M to UTF-N by the `boost::parser::replace` range adaptor. This doesn't require memory allocation; `boost::parser::replace` just slaps `| boost::parser::as_utfN` onto `replacement`. However, since Boost.Parser treats `char` ranges as unknown encoding, `boost::parser::replace` will not transcode from `char` ranges. So calls like this won't work:

```
char const str[] = "some text";
char const replacement_str[] = "some text";
using namespace bp = boost::parser;
auto r = empty_str | bp::replace(parser, replacement_str | bp::as_utf8); // Error: ill
    -formed!  Can't mix plain-char inputs and UTF replacements.
```

This does not work, even though `char` and UTF-8 are the same size. If `r` and `replacement` are both ranges of `char`, everything will work of course. It's just mixing `char` and UTF-encoded ranges that does not work.

All the details called out in the subsection on `boost::parser::search()` above apply to `boost::parser::replace`: its parser produces no attributes; it accepts C-style strings for the `r` and `replacement` parameters as if they were ranges; and it knows how to get from the internally-used iterator type back to the given iterator type, in typical cases.

`boost::parser::replace` can be called with, and `boost::parser::replace_view` can be constructed with, a skip parser or not, and you can always pass `boost::parser::trace` at the end of any of their overloads.

boost::parser::transform_replace

Important

`boost::parser::transform_replace` and `boost::parser::transform_replace_view` are not available on MSVC in C++17 mode.

Important

`boost::parser::transform_replace` and `boost::parser::transform_replace_view` are not available on GCC in C++20 mode before GCC 12.

`boost::parser::transform_replace` creates `boost::parser::transform_replace_views`. `boost::parser::transfor` is a `std::views`-style view. It produces a range of subranges from the parsed range `r` and the given invocable `f`. Wherever in the parsed range a match to the given parser `parser` is found, let `parser`'s attribute be `attr`; `f(std::move(attr))` is the subrange produced. Each subrange of `r` that does not match `parser` is

produced as a subrange as well. The subranges are produced in the order in which they occur in r. Unlike boost::parser::split_view, boost::parser::transform_replace_view does not produce empty subranges, unless f(std::move(attr)) is empty. Here is an example.

```
auto string_sum = [](std::vector<int> const & ints) {
    return std::to_string(std::accumulate(ints.begin(), ints.end(), 0));
};

auto rng = "There are groups of [1, 2, 3, 4, 5] in the set." |
           bp::transform_replace('[' >> bp::int_ % ',' >> ']', bp::ws, string_sum);
int count = 0;
// Prints "There are groups of 15 in the set".
for (auto subrange : rng) {
    for (auto ch : subrange) {
        std::cout << ch;
    }
    ++count;
}
std::cout << "\n";
assert(count == 3);
```

Let the type decltype(f(std::move(attr))) be Replacement. Replacement must be a range, and must be compatible with r. See the description of boost::parser::replace_view's iterator compatibility requirements in the section above for details.

As with boost::parser::replace, boost::parser::transform_replace can be flattened from a view of subranges into a view of elements by piping it to std::views::join. See the section on boost::parser::replace above for an example.

Just like boost::parser::replace and boost::parser::replace_view, boost::parser::transform_replace and boost::parser::transform_replace_view do silent transcoding of the result to the appropriate UTF, if applicable. If both r and f(std::move(attr)) are ranges of char, or are both the same UTF, no transcoding occurs. If one of r and f(std::move(attr)) is a range of char and the other is some UTF, the program is ill-formed.

boost::parser::transform_replace_view will move each attribute into f; f may move from the argument or copy it as desired. f may return an lvalue reference. If it does so, the address of the reference will be taken and stored within boost::parser::transform_replace_view. Otherwise, the value returned by f is moved into boost::parser::transform_replace_view. In either case, the value type of boost::parser::transform_replace_view is always a subrange.

boost::parser::transform_replace can be called with, and boost::parser::transform_replace_view can be constructed with, a skip parser or not, and you can always pass boost::parser::trace at the end of any of their overloads.

1.1.22 Unicode Support

Boost.Parser was designed from the start to be Unicode friendly. There are numerous references to the "Unicode code path" and the "non-Unicode code path" in the Boost.Parser documentation. Though there are in fact two code paths for Unicode and non-Unicode parsing, the code is not very different in the two code

paths, as they are written generically. The only difference is that the Unicode code path parses the input as a range of code points, and the non-Unicode path does not. In effect, this means that, in the Unicode code path, when you call `parse(r, p)` for some input range `r` and some parser `p`, the parse happens as if you called `parse(r | boost::parser::as_utf32, p)` instead. (Of course, it does not matter if `r` is a proper range, or an iterator/sentinel pair; those both work fine with `boost::parser::as_utf32`.)

Matching "characters" within Boost.Parser's parsers is assumed to be a code point match. In the Unicode path there is a code point from the input that is matched to each `char_` parser. In the non-Unicode path, the encoding is unknown, and so each element of the input is considered to be a whole "character" in the input encoding, analogous to a code point. From this point on, I will therefore refer to a single element of the input exclusively as a code point.

So, let's say we write this parser:

```
constexpr auto char8_parser = boost::parser::char_('\xcc');
```

For any `char_` parser that should match a value or values, the type of the value to match is retained. So `char8_parser` contains a `char` that it will use for matching. If we had written:

```
constexpr auto char32_parser = boost::parser::char_(U'\xcc');
```

`char32_parser` would instead contain a `char32_t` that it would use for matching.

So, at any point during the parse, if `char8_parser` were being used to match a code point `next_cp` from the input, we would see the moral equivalent of `next_cp == '\xcc'`, and if `char32_parser` were being used to match `next_cp`, we'd see the equivalent of `next_cp == U'\xcc'`. The take-away here is that you can write `char_` parsers that match specific values, without worrying if the input is Unicode or not because, under the covers, what takes place is a simple comparison of two integral values.

Note

Boost.Parser actually promotes any two values to a common type using `std::common_type` before comparing them. This is almost always works because the input and any parameter passed to `char_` must be character types.

Since matches are always done at a code point level (remember, a "code point" in the non-Unicode path is assumed to be a single `char`), you get different results trying to match UTF-8 input in the Unicode and non-Unicode code paths:

```
namespace bp = boost::parser;

{
    std::string str = (char const *)u8"\xcc\x80"; // encodes the code point U+0300
    auto first = str.begin();

    // Since we've done nothing to indicate that we want to do Unicode
    // parsing, and we've passed a range of char to parse(), this will do
    // non-Unicode parsing.
    std::string chars;
    assert(bp::parse(first, str.end(), *bp::char_('\xcc'), chars));
```

```
    // Finds one match of the *char* 0xcc, because the value in the parser
    // (0xcc) was matched against the two code points in the input (0xcc and
    // 0x80), and the first one was a match.
    assert(chars == "\xcc");
}
{
    std::u8string str = u8"\xcc\x80"; // encodes the code point U+0300
    auto first = str.begin();

    // Since the input is a range of char8_t, this will do Unicode
    // parsing.  The same thing would have happened if we passed
    // str | boost::parser::as_utf32 or even str | boost::parser::as_utf8.
    std::string chars;
    assert(bp::parse(first, str.end(), *bp::char_('\xcc'), chars));

    // Finds zero matches of the *code point* 0xcc, because the value in
    // the parser (0xcc) was matched against the single code point in the
    // input, 0x0300.
    assert(chars == "");
}
```

Implicit transcoding

Additionally, it is expected that most programs will use UTF-8 for the encoding of Unicode strings. Boost.Parser is written with this typical case in mind. This means that if you are parsing 32-bit code points (as you always are in the Unicode path), and you want to catch the result in a container `C` of `char` or `char8_t` values, Boost.Parser will silently transcode from UTF-32 to UTF-8 and write the attribute into `C`. This means that `std::string`, `std::u8string`, etc. are fine to use as attribute out-parameters for `*char_`, and the result will be UTF-8.

Note

UTF-16 strings as attributes are not supported directly. If you want to use UTF-16 strings as attributes, you may need to do so by transcoding a UTF-8 or UTF-32 attribute to UTF-16 within a semantic action. You can do this by using `boost::parser::as_utf16`.

The treatment of strings as UTF-8 is nearly ubiquitous within Boost.Parser. For instance, though the entire interface of `symbols` uses `std::string` or `std::string_view`, UTF-32 comparisons are used internally.

Explicit transcoding

I mentioned above that the use of `boost::parser::utf*_view` as the range to parse opts you in to Unicode parsing. Here's a bit more about these views and how best to use them.

If you want to do Unicode parsing, you're always going to be comparing code points at each step of the parse. As such, you're going to implicitly convert any parse input to UTF-32, if needed. This is what all the parse API functions do internally.

However, there are times when you have parse input that is a sequence of UTF-8-encoded `char`s, and you want to do Unicode-aware parsing. As mentioned previously, Boost.Parser has a special case for `char` inputs, and it will **not** assume that `char` sequences are UTF-8. If you want to tell the parse API to do Unicode processing on them anyway, you can use the `as_utf32` range adapter. (Note that you can use any of the `as_utf*` adaptors and the semantics will not differ from the semantics below.)

```
namespace bp = boost::parser;

auto const p = '"' >> *(bp::char_ - '"' - 0xb6) >> '"';
char const * str = "\"two wörds\""; // ö is two code units, 0xc3 0xb6

auto result_1 = bp::parse(str, p);                // Treat each char as a code point (
    typically ASCII).
assert(!result_1);
auto result_2 = bp::parse(str | bp::as_utf32, p); // Unicode-aware parsing on code
    points.
assert(result_2);
```

The first call to `parse()` treats each `char` as a code point, and since `"ö"` is the pair of code units `0xc3` `0xb6`, the parse matches the second code unit against the `- 0xb6` part of the parser above, causing the parse to fail. This happens because each code unit/`char` in `str` is treated as an independent code point.

The second call to `parse()` succeeds because, when the parse gets to the code point for `'ö'`, it is `0xf6` (U+00F6), which does not match the `- 0xb6` part of the parser.

The other adaptors `as_utf8` and `as_utf16` are also provided for completeness, if you want to use them. They each can transcode any sequence of character types.

Important

The `as_utfN` adaptors are optional, so they don't come with `parser.hpp`. To get access to them, `#include <boost/parser/transcode_view.hpp>`.

(Lack of) normalization

One thing that Boost.Parser does not handle for you is normalization; Boost.Parser is completely normalization-agnostic. Since all the parsers do their matching using equality comparisons of code points, you should make sure that your parsed range and your parsers all use the same normalization form.

1.1.23 Callback Parsing

In most parsing cases, being able to generate an attribute that represents the result of the parse, or being able to parse into such an attribute, is sufficient. Sometimes, it is not. If you need to parse a very large chunk of text, the generated attribute may be too large to fit in memory. In other cases, you may want to generate attributes

sometimes, and not others. `callback_rules` exist for these kinds of uses. A `callback_rule` is just like a rule, except that it allows the rule's attribute to be returned to the caller via a callback, as long as the parse is started with a call to `callback_parse()` instead of `parse()`. Within a call to `parse()`, a `callback_rule` is identical to a regular `rule`.

For a rule with no attribute, the signature of a callback function is `void (tag)`, where `tag` is the tag-type used when declaring the rule. For a rule with an attribute `attr`, the signature is `void (tag, attr)`. For instance, with this rule:

```
boost::parser::callback_rule<struct foo_tag> foo = "foo";
```

this would be an appropriate callback function:

```
void foo_callback(foo_tag)
{
    std::cout << "Parsed a 'foo'!\n";
}
```

For this rule:

```
boost::parser::callback_rule<struct bar_tag, std::string> bar = "bar";
```

this would be an appropriate callback function:

```
void bar_callback(bar_tag, std::string const & s)
{
    std::cout << "Parsed a 'bar' containing " << s << "!\n";
}
```

Important

In the case of `bar_callback()`, we don't need to do anything with `s` besides insert it into a stream, so we took it as a `const` lvalue reference. Boost.Parser moves all attributes into callbacks, so the signature could also have been `void bar_callback(bar_tag, std::string s)` or `void bar_callback(bar_tag, std::string && s)`.

You opt into callback parsing by parsing with a call to `callback_parse()` instead of `parse()`. If you use `callback_rules` with `parse()`, they're just regular `rules`. This allows you to choose whether to do "normal" attribute-generating/attribute-assigning parsing with `parse()`, or callback parsing with `callback_parse()`, without rewriting much parsing code, if any.

The only reason all `rules` are not `callback_rules` is that you may want to have some `rules` use callbacks within a parse, and have some that do not. For instance, if you want to report the attribute of `callback_rule` `r1` via callback, `r1`'s implementation may use some rule `r2` to generate some or all of its attribute.

See Parsing JSON With Callbacks for an extended example of callback parsing.

1.1.24 Error Handling and Debugging

Error handling

Boost.Parser has good error reporting built into it. Consider what happens when we fail to parse at an expectation point (created using `operator>`). If I feed the parser from the Parsing JSON With Callbacks example a file called sample.json containing this input (note the unmatched `'['`):

```
{
    "key": "value",
    "foo": [, "bar": []
}
```

This is the error message that is printed to the terminal:

```
sample.json:3:12: error: Expected ']' here:
    "foo": [, "bar": []
           ^
```

That message is formatted like the diagnostics produced by Clang and GCC. It quotes the line on which the failure occurred, and even puts a caret under the exact position at which the parse failed. This error message is suitable for many kinds of end-users, and interoperates well with anything that supports Clang and/or GCC diagnostics.

Most of Boost.Parser's error handlers format their diagnostics this way, though you are not bound by that. You can make an error handler type that does whatever you want, as long as it meets the error handler interface.

The Boost.Parser error handlers are:

- `default_error_handler`: Produces formatted diagnostics like the one above, and prints them to `std::cerr`. `default_error_handler` has no associated file name, and both errors and diagnostics are printed to `std::cerr`. This handler is `constexpr`-friendly.
- `stream_error_handler`: Produces formatted diagnostics. One or two streams may be used. If two are used, errors go to one stream and warnings go to the other. A file name can be associated with the parse; if it is, that file name will appear in all diagnostics.
- `callback_error_handler`: Produces formatted diagnostics. Calls a callback with the diagnostic message to report the diagnostic, rather than streaming out the diagnostic. A file name can be associated with the parse; if it is, that file name will appear in all diagnostics. This handler is useful for recording the diagnostics in memory.
- `rethrow_error_handler`: Does nothing but re-throw any exception that it is asked to handle. Its `diagnose()` member functions are no-ops.
- `vs_output_error_handler`: Directs all errors and warnings to the debugging output panel inside Visual Studio. Available on Windows only. Probably does nothing useful desirable when executed outside of Visual Studio.

You can set the error handler to any of these, or one of your own, using `with_error_handler()` (see The `parse()` API). If you do not set one, `default_error_handler` will be used.

How diagnostics are generated

Boost.Parser only generates error messages like the ones in this page at failed expectation points, like `a > b`, where you have successfully parsed `a`, but then cannot successfully parse `b`. This may seem limited to you. It's actually the best that we can do.

In order for error handling to happen other than at expectation points, we have to know that there is no further processing that might take place. This is true because Boost.Parser has `P1 | P2 | ... | Pn` parsers ("`or_parser`s"). If any one of these parsers `Pi` fails to match, it is not allowed to fail the parse — the next one (`Pi+1`) might match. If we get to the end of the alternatives of the or_parser and `Pn` fails, we still cannot fail the top-level parse, because the `or_parser` might be a subparser within a parent `or_parser`.

Ok, so what might we do? Perhaps we could at least indicate when we ran into end-of-input. But we cannot, for exactly the same reason already stated. For any parser `P`, reaching end-of-input is a failure for `P`, but not necessarily for the whole parse.

Perhaps we could record the farthest point ever reached during the parse, and report that at the top level, if the top level parser fails. That would be little help without knowing which parser was active when we reached that point. This would require some sort of repeated memory allocation, since in Boost.Parser the progress point of the parser is stored exclusively on the stack — by the time we fail the top-level parse, all those far-reaching stack frames are long gone. Not the best.

Worse still, knowing how far you got in the parse and which parser was active is not very useful. Consider this.

```
namespace bp = boost::parser;
auto a_b = bp::char_('a') >> bp::char_('b');
auto c_b = bp::char_('c') >> bp::char_('b');
auto result = bp::parse("acb", a_b | c_b);
```

If we reported the farthest-reaching parser and it's position, it would be the `a_b` parser, at position `"bc"` in the input. Is this really enlightening? Was the error in the input putting the `'a'` at the beginning or putting the `'c'` in the middle? If you point the user at `a_b` as the parser that failed, and never mention `c_b`, you are potentially just steering them in the wrong direction.

All error messages must come from failed expectation points. Consider parsing JSON. If you open a list with `'['`, you know that you're parsing a list, and if the list is ill-formed, you'll get an error message saying so. If you open an object with `'{'`, the same thing is possible — when missing the matching `'}'`, you can tell the user, "That's not an object", and this is useful feedback. The same thing with a partially parsed number, etc. If the JSON parser does not build in expectations like matched braces and brackets, how can Boost.Parser know that a missing `'}'` is really a problem, and that no later parser will match the input even without the `'}'`?

Important

The bottom line is that you should build expectation points into your parsers using `operator>` as much as possible.

Using error handlers in semantic actions

You can get access to the error handler within any semantic action by calling `_error_handler(ctx)` (see The Parse Context). Any error handler must have the following member functions:

```
template<typename Context, typename Iter>
void diagnose(
    diagnostic_kind kind,
    std::string_view message,
    Context const & context,
    Iter it) const;

template<typename Context>
void diagnose(
    diagnostic_kind kind,
    std::string_view message,
    Context const & context) const;
```

If you call the second one, the one without the iterator parameter, it will call the first with `_where(context).begin()` as the iterator parameter. The one without the iterator is the one you will use most often. The one with the explicit iterator parameter can be useful in situations where you have messages that are related to each other, associated with multiple locations. For instance, if you are parsing XML, you may want to report that a close-tag does not match its associated open-tag by showing the line where the open-tag was found. That may of course not be located anywhere near `_where(ctx).begin()`. (A description of `_globals()` is below.)

```
[](auto & ctx) {
    // Assume we have a std::vector of open tags, and another
    // std::vector of iterators to where the open tags were parsed, in our
    // globals.
    if (_attr(ctx) != _globals(ctx).open_tags.back()) {
        std::string open_tag_msg =
            "Previous open-tag \"" + _globals(ctx).open_tags.back() + "\" here:";
        _error_handler(ctx).diagnose(
            boost::parser::diagnostic_kind::error,
            open_tag_msg,
            ctx,
            _globals(ctx).open_tags_position.back());
        std::string close_tag_msg =
            "does not match close-tag \"" + _attr(ctx) + "\" here:";
        _error_handler(ctx).diagnose(
            boost::parser::diagnostic_kind::error,
            close_tag_msg,
            ctx);

        // Explicitly fail the parse.  Diagnostics do not affect parse success.
        _pass(ctx) = false;
    }
}
```

_report_error() and _report_warning()

There are also some convenience functions that make the above code a little less verbose, `_report_error()` and `_report_warning()`:

```
[](auto & ctx) {
    // Assume we have a std::vector of open tags, and another
    // std::vector of iterators to where the open tags were parsed, in our
    // globals.
    if (_attr(ctx) != _globals(ctx).open_tags.back()) {
        std::string open_tag_msg =
            "Previous open-tag \"" + _globals(ctx).open_tags.back() + "\" here:";
        _report_error(ctx, open_tag_msg, _globals(ctx).open_tag_positions.back());
        std::string close_tag_msg =
            "does not match close-tag \"" + _attr(ctx) + "\" here:";
        _report_error(ctx, close_tag_msg);

        // Explicitly fail the parse.  Diagnostics do not affect parse success.
        _pass(ctx) = false;
    }
}
```

You should use these less verbose functions almost all the time. The only time you would want to use `_error_handler()` directly is when you are using a custom error handler, and you want access to some part of its interface besides `diagnose()`.

Though there is support for reporting warnings using the functions above, none of the error handlers supplied by Boost.Parser will ever report a warning. Warnings are strictly for user code.

For more information on the rest of the error handling and diagnostic API, see the header reference pages for `error_handling_fwd.hpp` and `error_handling.hpp`.

Creating your own error handler

Creating your own error handler is pretty easy; you just need to implement three member functions. Say you want an error handler that writes diagnostics to a file. Here's how you might do that.

```
struct logging_error_handler
{
    logging_error_handler() {}
    logging_error_handler(std::string_view filename) :
        filename_(filename), ofs_(filename_)
    {
        if (!ofs_)
            throw std::runtime_error("Could not open file.");
    }

    // This is the function called by Boost.Parser after a parser fails the
    // parse at an expectation point and throws a parse_error.  It is expected
    // to create a diagnostic message, and put it where it needs to go.  In
```

```
    // this case, we're writing it to a log file.  This function returns a
    // bp::error_handler_result, which is an enum with two enumerators -- fail
    // and rethrow.  Returning fail fails the top-level parse; returning
    // rethrow just re-throws the parse_error exception that got us here in
    // the first place.
    template<typename Iter, typename Sentinel>
    bp::error_handler_result
    operator()(Iter first, Sentinel last, bp::parse_error<Iter> const & e) const
    {
        bp::write_formatted_expectation_failure_error_message(
            ofs_, filename_, first, last, e);
        return bp::error_handler_result::fail;
    }

    // This function is for users to call within a semantic action to produce
    // a diagnostic.
    template<typename Context, typename Iter>
    void diagnose(
        bp::diagnostic_kind kind,
        std::string_view message,
        Context const & context,
        Iter it) const
    {
        bp::write_formatted_message(
            ofs_,
            filename_,
            bp::_begin(context),
            it,
            bp::_end(context),
            message);
    }

    // This is just like the other overload of diagnose(), except that it
    // determines the Iter parameter for the other overload by calling
    // _where(ctx).
    template<typename Context>
    void diagnose(
        bp::diagnostic_kind kind,
        std::string_view message,
        Context const & context) const
    {
        diagnose(kind, message, context, bp::_where(context).begin());
    }

    std::string filename_;
    mutable std::ofstream ofs_;
};
```

That's it. You just need to do the important work of the error handler in its call operator, and then implement the two overloads of `diagnose()` that it must provide for use inside semantic actions. The default implemen-

tation of these is even available as the free function `write_formatted_message()`, so you can just call that, as you see above. Here's how you might use it.

```
int main()
{
    std::cout << "Enter a list of integers, separated by commas. ";
    std::string input;
    std::getline(std::cin, input);

    constexpr auto parser = bp::int_ >> *(',' > bp::int_);
    logging_error_handler error_handler("parse.log");
    auto const result = bp::parse(input, bp::with_error_handler(parser, error_handler)
    );

    if (result) {
        std::cout << "It looks like you entered:\n";
        for (int x : *result) {
            std::cout << x << "\n";
        }
    }
}
```

We just define a `logging_error_handler`, and pass it by reference to `with_error_handler()`, which decorates the top-level parser with the error handler. We **could not** have written `bp::with_error_handler(parser, logging_error_handler("parse.log"))`, because `with_error_handler()` does not accept rvalues. This is becuse the error handler eventually goes into the parse context. The parse context only stores pointers and iterators, keeping it cheap to copy.

If we run the example and give it the input `"1,"`, this shows up in the log file:

```
parse.log:1:2: error: Expected int_ here (end of input):
1,
  ^
```

Fixing ill-formed code

Sometimes, during the writing of a parser, you make a simple mistake that is diagnosed horrifyingly, due to the high number of template instantiations between the line you just wrote and the point of use (usually, the call to `parse()`). By "sometimes", I mean "almost always and many, many times". Boost.Parser has a workaround for situations like this. The workaround is to make the ill-formed code well-formed in as many circumstances as possible, and then do a runtime assert instead.

Usually, C++ programmers try whenever they can to catch mistakes as early as they can. That usually means making as much bad code ill-formed as possible. Counter-intuitively, this does not work well in parser combinator situations. For an example of just how dramatically different these two debugging scenarios can be with Boost.Parser, please see the very long discussion in the `none` is weird section of Rationale.

If you are morally opposed to this approach, or just hate fun, good news: you can turn off the use of this technique entirely by defining `BOOST_PARSER_NO_RUNTIME_ASSERTIONS`.

Runtime debugging

Debugging parsers is hard. Any parser above a certain complexity level is nearly impossible to debug simply by looking at the parser's code. Stepping through the parse in a debugger is even worse. To provide a reasonable chance of debugging your parsers, Boost.Parser has a trace mode that you can turn on simply by providing an extra parameter to `parse()` or `callback_parse()`:

```
boost::parser::parse(input, parser, boost::parser::trace::on);
```

Every overload of `parse()` and `callback_parse()` takes this final parameter, which is defaulted to `boost::parser::trac`

If we trace a substantial parser, we will see a **lot** of output. Each code point of the input must be considered, one at a time, to see if a certain rule matches. An an example, let's trace a parse using the JSON parser from Parsing JSON. The input is `"null"`. `null` is one of the types that a Javascript value can have; the top-level parser in the JSON parser example is:

```
auto const value_p_def =
    number | bp::bool_ | null | string | array_p | object_p;
```

So, a JSON value can be a number, or a Boolean, a `null`, etc. During the parse, each alternative will be tried in turn, until one is matched. I picked `null` because it is relatively close to the beginning of the `value_p_def` alternative parser. Even so, the output is pretty huge. Let's break it down as we go:

```
[begin value; input="null"]
```

Each parser is traced as `[begin foo; ...]`, then the parsing operations themselves, and then `[end foo; ...]`. The name of a rule is used as its name in the `begin` and `end` parts of the trace. Non-rules have a name that is similar to the way the parser looked when you wrote it. Most lines will have the next few code points of the input quoted, as we have here (`input="null"`).

```
[begin number | bool_ | null | string | ...; input="null"]
```

This shows the beginning of the parser **inside** the rule `value` — the parser that actually does all the work. In the example code, this parser is called `value_p_def`. Since it isn't a rule, we have no name for it, so we show its implementation in terms of subparsers. Since it is a bit long, we don't print the entire thing. That's why that ellipsis is there.

```
[begin number; input="null"]
  [begin raw[lexeme[ >> ...]][<<action>>]; input="null"]
```

Now we're starting to see the real work being done. `number` is a somewhat complicated parser that does not match `"null"`, so there's a lot to wade through when following the trace of its attempt to do so. One thing to note is that, since we cannot print a name for an action, we just print `"<<action>>"`. Something similar happens when we come to an attribute that we cannot print, because it has no stream insertion operation. In that case, `"<<unprintable-value>>"` is printed.

```
    [begin raw[lexeme[ >> ...]]; input="null"]
      [begin lexeme[-char_('-') >> char_('1', '9') >> ... | ... >> ...]; input="null"]
        [begin -char_('-') >> char_('1', '9') >> *digit | char_('0') >> -(char_('.')
    >> ...) >> -( >> ...); input="null"]
          [begin -char_('-'); input="null"]
            [begin char_('-'); input="null"]
```

```
            no match
          [end char_('-'); input="null"]
          matched ""
          attribute: <<empty>>
        [end -char_('-'); input="null"]
        [begin char_('1', '9') >> *digit | char_('0'); input="null"]
          [begin char_('1', '9') >> *digit; input="null"]
            [begin char_('1', '9'); input="null"]
              no match
            [end char_('1', '9'); input="null"]
            no match
          [end char_('1', '9') >> *digit; input="null"]
          [begin char_('0'); input="null"]
            no match
          [end char_('0'); input="null"]
          no match
        [end char_('1', '9') >> *digit | char_('0'); input="null"]
        no match
      [end -char_('-') >> char_('1', '9') >> *digit | char_('0') >> -(char_('.') >>
...) >> -( >> ...); input="null"]
      no match
    [end lexeme[-char_('-') >> char_('1', '9') >> ... | ... >> ...]; input="null"]
    no match
  [end raw[lexeme[ >> ...]]; input="null"]
  no match
[end raw[lexeme[ >> ...]][<<action>>]; input="null"]
no match
[end number; input="null"]
[begin bool_; input="null"]
  no match
[end bool_; input="null"]
```

`number` and `boost::parser::bool_` did not match, but `null` will:

```
[begin null; input="null"]
  [begin "null" >> attr(null); input="null"]
    [begin "null"; input="null"]
      [begin string("null"); input="null"]
        matched "null"
        attribute:
      [end string("null"); input=""]
      matched "null"
      attribute: null
```

Finally, this parser actually matched, and the match generated the attribute `null`, which is a special value of the type `json::value`. Since we were matching a string literal `"null"`, earlier there was no attribute until we reached the `attr(null)` parser.

```
      [end "null"; input=""]
      [begin attr(null); input=""]
        matched ""
```

```
          attribute: null
        [end attr(null); input=""]
        matched "null"
        attribute: null
      [end "null" >> attr(null); input=""]
      matched "null"
      attribute: null
    [end null; input=""]
    matched "null"
    attribute: null
  [end number | bool_ | null | string | ...; input=""]
  matched "null"
  attribute: null
[end value; input=""]
--------------------
parse succeeded
--------------------
```

At the very end of the parse, the trace code prints out whether the top-level parse succeeded or failed.

Some things to be aware of when looking at Boost.Parser trace output:

- There are some parsers you don't know about, because they are not directly documented. For instance, `p[a]` forms an `action_parser` containing the parser `p` and semantic action `a`. This is essentially an implementation detail, but unfortunately the trace output does not hide this from you.
- For a parser `p`, the trace-name may be intentionally different from the actual structure of `p`. For example, in the trace above, you see a parser called simply `"null"`. This parser is actually `boost::parser::omit[boost::par` but what you typically write is just `"null"`, so that's the name used. There are two special cases like this: the one described here for `omit[string]`, and another for `omit[char_]`.
- Since there are no other special cases for how parser names are printed, you may see parsers that are unlike what you wrote in your code. In the sections about the parsers and combining operations, you will sometimes see a parser or combining operation described in terms of an equivalent parser. For example, `if_(pred)[p]` is described as "Equivalent to `eps(pred) >> p`". In a trace, you will not see `if_`; you will see `eps` and `p` instead.
- The values of arguments passed to parsers is printed whenever possible. Sometimes, a parse argument is not a value itself, but a callable that produces that value. In these cases, you'll see the resolved value of the parse argument.

1.1.25 Memory Allocation

Boost.Parser seldom allocates memory. The exceptions to this are:

- `symbols` allocates memory for the symbol/attribute pairs it contains. If symbols are added during the parse, allocations must also occur then. The data structure used by `symbols` is also a trie, which is a node-based tree. So, lots of allocations are likely if you use `symbols`.
- The error handlers that can take a file name allocate memory for the file name, if one is provided.
- If trace is turned on by passing `boost::parser::trace::on` to a top-level parsing function, the names of parsers are allocated.

- When a failed expectation is encountered (using `operator>`), the name of the failed parser is placed into a `std::string`, which will usually cause an allocation.
- `string()`'s attribute is a `std::string`, the use of which implies allocation. You can avoid this allocation by explicitly using a different string type for the attribute that does not allocate.
- The attribute for `repeat(p)` in all its forms, including `operator*`, `operator+`, and `operator%`, is `std::vector<ATTR(p)>`, the use of which implies allocation. You can avoid this allocation by explicitly using a different sequence container for the attribute that does not allocate. `boost::container::static_vector` or C++26's `std::inplace_vector` may be useful as such replacements.

With the exception of allocating the name of the parser that was expected in a failed expectation situation, Boost.Parser does not does not allocate unless you tell it to, by using `symbols`, using a particular error_handler, turning on trace, or parsing into attributes that allocate.

1.1.26 Best Practices

Parse unicode from the start

If you want to parse ASCII, using the Unicode parsing API will not actually cost you anything. Your input will be parsed, `char` by `char`, and compared to values that are Unicode code points (which are `char32_t`s). One caveat is that there may be an extra branch on each char, if the input is UTF-8. If your performance requirements can tolerate this, your life will be much easier if you just start with Unicode and stick with it.

Starting with Unicode support and UTF-8 input will allow you to properly handle unexpected input, like non-ASCII languages (that's most of them), with no additional effort on your part.

Write rules, and test them in isolation

Treat rules as the unit of work in your parser. Write a rule, test its corners, and then use it to build larger rules or parsers. This allows you to get better coverage with less work, since exercising all the code paths of your rules, one by one, keeps the combinatorial number of paths through your code manageable.

Prefer auto-generated attributes to semantic actions

There are multiple ways to get attributes out of a parser. You can:

- use whatever attribute the parser generates;
- provide an attribute out-argument to `parse()` for the parser to fill in;
- use one or more semantic actions to assign attributes from the parser to variables outside the parser;
- use callback parsing to provide attributes via callback calls.

All of these are fairly similar in how much effort they require, except for the semantic action method. For the semantic action approach, you need to have values to fill in from your parser, and keep them in scope for the duration of the parse.

It is much more straight forward, and leads to more reusable parsers, to have the parsers produce the attributes of the parse directly as a result of the parse.

This does not mean that you should never use semantic actions. They are sometimes necessary. However, you should default to using the other non-semantic action methods, and only use semantic actions with a good reason.

If your parser takes end-user input, give rules names that you would want an end-user to see

A typical error message produced by Boost.Parser will say something like, "Expected FOO here", where FOO is some rule or parser. Give your rules names that will read well in error messages like this. For instance, the JSON examples have these rules:

```
bp::rule<class escape_seq, uint32_t> const escape_seq =
    "\\uXXXX hexadecimal escape sequence";
bp::rule<class escape_double_seq, uint32_t, double_escape_locals> const
    escape_double_seq = "\\uXXXX hexadecimal escape sequence";
bp::rule<class single_escaped_char, uint32_t> const single_escaped_char =
    "'\"', '\\', '/', 'b', 'f', 'n', 'r', or 't'";
```

Some things to note:

- `escape_seq` and `escape_double_seq` have the same name-string. To an end-user who is trying to figure out why their input failed to parse, it doesn't matter which kind of result a parser rule generates. They just want to know how to fix their input. For either rule, the fix is the same: put a hexadecimal escape sequence there.

- `single_escaped_char` has a terrible-looking name. However, it's not really used as a name anywhere per se. In error messages, it works nicely, though. The error will be "Expected '"', '', '/', 'b', 'f', 'n', 'r', or 't' here", which is pretty helpful.

Have a simple test that you can run to find ill-formed-code-as-asserts

Most of these errors are found at parser construction time, so no actual parsing is even necessary. For instance, a test case might look like this:

```
TEST(my_parser_tests, my_rule_test) {
    my_rule r;
}
```

1.1.27 Writing Your Own Parsers

You should probably never need to write your own low-level parser. You have primitives like `char_` from which to build up the parsers that you need. It is unlikely that you're going to need to do things on a lower level than a single character.

However. Some people are obsessed with writing everything for themselves. We call them C++ programmers. This section is for them. However, this section is not an in-depth tutorial. It is a basic orientation to get you familiar enough with all the moving parts of writing a parser that you can then learn by reading the Boost.Parser code.

Each parser must provide two overloads of a function `call()`. One overload parses, producing an attribute

(which may be the special no-attribute type `detail::nope`). The other one parses, filling in a given attribute. The type of the given attribute is a template parameter, so it can take any type that you can form a reference to.

Let's take a look at a Boost.Parser parser, `opt_parser`. This is the parser produced by use of `operator-`. First, here is the beginning of its definition.

```
template<typename Parser>
struct opt_parser
{
```

The end of its definition is:

```
    Parser parser_;
};
```

As you can see, `opt_parser`'s only data member is the parser it adapts, `parser_`. Here is its attribute-generating overload to `call()`.

```
template<
    typename Iter,
    typename Sentinel,
    typename Context,
    typename SkipParser>
auto call(
    Iter & first,
    Sentinel last,
    Context const & context,
    SkipParser const & skip,
    detail::flags flags,
    bool & success) const
{
    using attr_t = decltype(parser_.call(
        first, last, context, skip, flags, success));
    detail::optional_of<attr_t> retval;
    call(first, last, context, skip, flags, success, retval);
    return retval;
}
```

First, let's look at the template and function parameters.

- `Iter & first` is the iterator. It is taken as an out-param. It is the responsibility of `call()` to advance `first` if and only if the parse succeeds.
- `Sentinel last` is the sentinel. If the parse has not yet succeeded within `call()`, and `first == last` is `true`, `call()` must fail (by setting `bool & success` to `false`).
- `Context const & context` is the parse context. It will be some specialization of `detail::parse_context`. The context is used in any call to a subparser's `call()`, and in some cases a new context should be created, and the new context passed to a subparser instead; more on that below.
- `SkipParser const & skip` is the current skip parser. `skip` should be used at the beginning of the parse, and in between any two uses of any subparser(s).
- `detail::flags flags` are a collection of flags indicating various things about the current state of the

parse. `flags` is concerned with whether to produce attributes at all; whether to apply the skip parser `skip`; whether to produce a verbose trace (as when `boost::parser::trace::on` is passed at the top level); and whether we are currently inside the utility function `detail::apply_parser`.

- `bool & success` is the final function parameter. It should be set to `true` if the parse succeeds, and `false` otherwise.

Now the body of the function. Notice that it just dispatches to the other `call()` overload. This is really common, since both overloads need to to the same parsing; only the attribute may differ. The first line of the body defines `attr_t`, the default attribute type of our wrapped parser `parser_`. It does this by getting the `decltype()` of a use of `parser_.call()`. (This is the logic represented by *`ATTR()`* in the rest of the documentation.) Since `opt_parser` represents an optional value, the natural type for its attribute is `std::optional<`*`ATTR`*`(parser)>`. However, this does not work for all cases. In particular, it does not work for the "no-attribute" type `detail::nope`, nor for `std::optional<T>` — *`ATTR`*`(--p)` is just *`ATTR`*`(-p)`. So, the second line uses an alias that takes care of those details, `detail::optional_of<>`. The third line just calls the other overload of `call()`, passing `retval` as the out-param. Finally, `retval` is returned on the last line.

Now, on to the other overload.

```
template<
    typename Iter,
    typename Sentinel,
    typename Context,
    typename SkipParser,
    typename Attribute>
void call(
    Iter & first,
    Sentinel last,
    Context const & context,
    SkipParser const & skip,
    detail::flags flags,
    bool & success,
    Attribute & retval) const
{
    [[maybe_unused]] auto _ = detail::scoped_trace(
        *this, first, last, context, flags, retval);

    detail::skip(first, last, skip, flags);

    if (!detail::gen_attrs(flags)) {
        parser_.call(first, last, context, skip, flags, success);
        success = true;
        return;
    }

    parser_.call(first, last, context, skip, flags, success, retval);
    success = true;
}
```

The template and function parameters here are identical to the ones from the other overload, except that we have `Attribute & retval`, our out-param.

Let's look at the implementation a bit at a time.

```
[[maybe_unused]] auto _ = detail::scoped_trace(
    *this, first, last, context, flags, retval);
```

This defines a RAII trace object that will produce the verbose trace requested by the user if they passed `boost::parser::trace::on` to the top-level parse. It only has effect if `detail::enable_trace(flags)` is `true`. If trace is enabled, it will show the state of the parse at the point at which it is defined, and then again when it goes out of scope.

Important

For the tracing code to work, you must define an overload of `detail::print_parser` for your new parser type/template. See `<boost/parser/detail/printing.hpp>` for examples.

```
detail::skip(first, last, skip, flags);
```

This one is pretty simple; it just applies the skip parser. `opt_parser` only has one subparser, but if it had more than one, or if it had one that it applied more than once, it would need to repeat this line using `skip` between every pair of uses of any subparser.

```
if (!detail::gen_attrs(flags)) {
    parser_.call(first, last, context, skip, flags, success);
    success = true;
    return;
}
```

This path accounts for the case where we don't want to generate attributes at all, perhaps because this parser sits inside an `omit[]` directive.

```
parser_.call(first, last, context, skip, flags, success, retval);
success = true;
```

This is the other, typical, path. Here, we do want to generate attributes, and so we do the same call to `parser_.call()`, except that we also pass `retval`.

Note that we set `success` to `true` after the call to `parser_.call()` in both code paths. Since `opt_parser` is zero-or-one, if the subparser fails, `opt_parse` still succeeds.

When to make a new parse context

Sometimes, you need to change something about the parse context before calling a subparser. For instance, `rule_parser` sets up the value, locals, etc., that are available for that rule. `action_parser` adds the generated attribute to the context (available as `_attr(ctx)`). Contexts are immutable in Boost.Parser. To "modify" one for a subparser, you create a new one with the appropriate call to `detail::make_context()`.

`detail::apply_parser()`

Sometimes a parser needs to operate on an out-param that is not exactly the same as its default attribute, but that is compatible in some way. To do this, it's often useful for the parser to call itself, but with slightly different parameters. `detail::apply_parser()` helps with this. See the out-param overload of `repeat_parser::call()` for an example. Note that since this creates a new scope for the ersatz parser, the `scoped_trace` object needs to know whether we're inside `detail::apply_parser` or not.

That's a lot, I know. Again, this section is not meant to be an in-depth tutorial. You know enough now that the parsers in `parser.hpp` are at least readable.

1.2 Extended Examples

1.2.1 Parsing JSON

This is a conforming JSON parser. It passes all the required tests in the JSON Test Suite, and all but 5 of the optional ones. Notice that the actual parsing bits are only about 150 lines of code.

```
// This header includes a type called json::value that acts as a
// Javascript-like polymorphic value type.
#include "json.hpp"

#include <boost/parser/parser.hpp>

#include <fstream>
#include <vector>
#include <climits>


namespace json {

    namespace bp = ::boost::parser;
    using namespace bp::literals;

    // The JSON spec imposes a limit on how deeply JSON data structures are
    // allowed to nest.  This exception is thrown when that limit is exceeded
    // during the parse.
    template<typename Iter>
    struct excessive_nesting : std::runtime_error
    {
        excessive_nesting(Iter it) :
            runtime_error("excessive_nesting"),
            iter(it)
        {}
        Iter iter;
    };
```

```
// The only globals we need to parse JSON are: "How many data structures
// deep are we?", and "What is the limit of open data structures
// allowed?".
struct global_state
{
    int recursive_open_count = 0;
    int max_recursive_open_count = 0;
};

// When matching paired UTF-16 surrogates, we need to track a bit of state
// between matching the first and second UTF-16 code units: namely, the
// value of the first code unit.
struct double_escape_locals
{
    int first_surrogate = 0;
};


// Here are all the rules declared.  I've given them names that are
// end-user friendly, so that if there is a parse error, you get a message
// like "expected four hexadecimal digits here:", instead of "expected
// hex_4 here:".

bp::rule<class ws> const ws = "whitespace";

bp::rule<class string_char, uint32_t> const string_char =
    "code point (code points <= U+001F must be escaped)";
bp::rule<class four_hex_digits, uint32_t> const hex_4 =
    "four hexadecimal digits";
bp::rule<class escape_seq, uint32_t> const escape_seq =
    "\\uXXXX hexadecimal escape sequence";
bp::rule<class escape_double_seq, uint32_t, double_escape_locals> const
    escape_double_seq = "\\uXXXX hexadecimal escape sequence";
bp::rule<class single_escaped_char, uint32_t> const single_escaped_char =
    "'\"', '\\', '/', 'b', 'f', 'n', 'r', or 't'";

bp::rule<class null, value> const null = "null";
bp::rule<class string, std::string> const string = "string";
bp::rule<class number, double> const number = "number";
bp::rule<class object_element, boost::parser::tuple<std::string, value>> const
    object_element = "object-element";
bp::rule<class object_tag, value> const object_p = "object";
bp::rule<class array_tag, value> const array_p = "array";

bp::rule<class value_tag, value> const value_p = "value";



// JSON limits whitespace to just these four characters.
auto const ws_def = '\x09'_l | '\x0a' | '\x0d' | '\x20';
```

```
    // Since our json object representation, json::value, is polymorphic, and
    // since its default-constructed state represents the JSON value "null",
    // we need to tell a json::value that it is an object (similar to a map)
    // before we start inserting values into it.  That's why we need
    // object_init.
    auto object_init = [](auto & ctx) {
        auto & globals = _globals(ctx);
        if (globals.max_recursive_open_count < ++globals.recursive_open_count)
            throw excessive_nesting(_where(ctx).begin());
        _val(ctx) = object();
    };

    // We need object_insert because we can't just insert into the json::value
    // itself.  The json::value does not have an insert() member, because if
    // it is currently holding a number, that makes no sense.  So, for a
    // json::value x, we need to call get<object>(x) to get the object
    // interface.
    auto object_insert = [](auto & ctx) {
        value & v = _val(ctx);
        get<object>(v).insert(std::make_pair(
            std::move(_attr(ctx))[0_c], std::move(_attr(ctx)[1_c])));
    };

    // These are the array analogues of the object semantic actions above.
    auto array_init = [](auto & ctx) {
        auto & globals = _globals(ctx);
        if (globals.max_recursive_open_count < ++globals.recursive_open_count)
            throw excessive_nesting(_where(ctx).begin());
        _val(ctx) = array();
    };
    auto array_append = [](auto & ctx) {
        value & v = _val(ctx);
        get<array>(v).push_back(std::move(_attr(ctx)));
    };

    // escape_double_seq is used to match pairs of UTF-16 surrogates that form
    // a single code point.  So, after matching one UTF-16 code unit c, we
    // only want to keep going if c is a lead/high surrogate.
    auto first_hex_escape = [](auto & ctx) {
        auto & locals = _locals(ctx);
        uint32_t const cu = _attr(ctx);
        if (!boost::parser::detail::text::high_surrogate(cu))
            _pass(ctx) = false; // Not a high surrogate; explicitly fail the parse.
        else
            locals.first_surrogate = cu; // Save this initial code unit for later.
    };
    // This is also used in escape_double_seq.  When we get to this action, we
    // know we've already matched a high surrogate, and so this one had better
    // be a low surrogate, or we have a (local) parse failure.
    auto second_hex_escape = [](auto & ctx) {
        auto & locals = _locals(ctx);
```

```
        uint32_t const cu = _attr(ctx);
        if (!boost::parser::detail::text::low_surrogate(cu)) {
            _pass(ctx) = false; // Not a low surrogate; explicitly fail the parse.
        } else {
            // Success!  Write to the rule's attribute the code point that the
            // first and second code points form.
            uint32_t const high_surrogate_min = 0xd800;
            uint32_t const low_surrogate_min = 0xdc00;
            uint32_t const surrogate_offset =
                0x10000 - (high_surrogate_min << 10) - low_surrogate_min;
            uint32_t const first_cu = locals.first_surrogate;
            _val(ctx) = (first_cu << 10) + cu + surrogate_offset;
        }
    };

    // This is the verbose form of declaration for the integer and unsigned
    // integer parsers int_parser and uint_parser.  In this case, we don't
    // want to use boost::parser::hex directly, since it has a variable number
    // of digits.  We want to match exactly 4 digits, and this is how we
    // declare a hexadecimal parser that matches exactly 4.
    bp::parser_interface<bp::uint_parser<uint32_t, 16, 4, 4>> const hex_4_def;

    // We use > here instead of >>, because once we see \u, we know that
    // exactly four hex digits must follow -- no other production rule starts
    // with \u.
    auto const escape_seq_def = "\\u" > hex_4;

    // This uses the actions above and the simpler rule escape_seq to find
    // matched UTF-16 surrogate pairs.
    auto const escape_double_seq_def =
        escape_seq[first_hex_escape] >> escape_seq[second_hex_escape];

    // This symbol table recognizes each character that can appear right after
    // an escaping backslash, and, if it finds one, produces the associated
    // code point as its attribute.
    bp::symbols<uint32_t> const single_escaped_char_def = {
        {"\"", 0x0022u},
        {"\\", 0x005cu},
        {"/", 0x002fu},
        {"b", 0x0008u},
        {"f", 0x000cu},
        {"n", 0x000au},
        {"r", 0x000du},
        {"t", 0x0009u}};

    // A string may be a matched UTF-16 escaped surrogate pair, a single
    // escaped UTF-16 code unit treated as a whole code point, a single
    // escaped character like \f, or any other code point outside the range
    // [0x0000u, 0x001fu].  Note that we had to put escape_double_seq before
    // escape_seq.  Otherwise, escape_seq would eat all the escape sequences
    // before escape_double_seq could try to match them.
```

```
    auto const string_char_def = escape_double_seq | escape_seq |
                                 ('\\'_l > single_escaped_char) |
                                 (bp::cp - bp::char_(0x0000u, 0x001fu));

    // If we see the special token null, treat that as a default-constructed
    // json::value.  Note that we could have done this with a semantic action,
    // but it is best to do everything you can without semantic actions;
    // they're a lot of code.
    auto const null_def = "null" >> bp::attr(value());

    auto const string_def = bp::lexeme['"' >> *(string_char - '"') > '"'];

    // Since the JSON format for numbers is not exactly what
    // boost::parser::double_ accepts (double_ accepts too much), we need to
    // parse a JSON number as a sequence of characters, and then pass the
    // result to double_ to actually get the numeric value.  This action does
    // that.  The parser uses boost::parser::raw to produce the subrange of
    // the input that covers the number as an attribute, which is used here.
    auto parse_double = [](auto & ctx) {
        auto const cp_range = _attr(ctx);
        auto cp_first = cp_range.begin();
        auto const cp_last = cp_range.end();

        auto const result = bp::prefix_parse(cp_first, cp_last, bp::double_);
        if (result) {
            _val(ctx) = *result;
        } else {
            // This would be more efficient if we used
            // boost::container::small_vector, or std::inplace_vector from
            // C++26.
            std::vector<char> chars(cp_first, cp_last);
            auto const chars_first = &*chars.begin();
            auto chars_last = chars_first + chars.size();
            _val(ctx) = std::strtod(chars_first, &chars_last);
        }
    };

    // As indicated above, we want to match the specific formats JSON allows,
    // and then re-parse the resulting matched range within the semantic
    // action.
    auto const number_def =
        bp::raw[bp::lexeme
                    [-bp::char_('-') >>
                     (bp::char_('1', '9') >> *bp::digit | bp::char_('0')) >>
                     -(bp::char_('.') >> +bp::digit) >>
                     -(bp::char_("eE") >> -bp::char_("+-") >> +bp::digit)]]
                [parse_double];

    // Note how, in the next three parsers, we turn off backtracking by using
    // > instead of >>, once we know that there is no backtracking alternative
    // that might match if we fail to match the next element.  This produces
```

```
    // much better error messages than if you always use >>.

    auto const object_element_def = string > ':' > value_p;

    auto const object_p_def = '{'_l[object_init] >>
                              -(object_element[object_insert] % ',') > '}';

    auto const array_p_def = '['_l[array_init] >>
                             -(value_p[array_append] % ',') > ']';

    // This is the top-level parser.
    auto const value_p_def =
        number | bp::bool_ | null | string | array_p | object_p;

    // Here, we define all the rules we've declared above, which also connects
    // each rule to its _def-suffixed parser.
    BOOST_PARSER_DEFINE_RULES(
        ws,
        hex_4,
        escape_seq,
        escape_double_seq,
        single_escaped_char,
        string_char,
        null,
        string,
        number,
        object_element,
        object_p,
        array_p,
        value_p);

    // json::parse() takes a string_view as input.  It takes an optional
    // callback to use for error reporting, which defaults to a no-op that
    // ignores all errors.  It also takes an optional max recursion depth
    // limit, which defaults to the one from the JSON spec, 512.
    std::optional<value> parse(
        std::string_view str,
        diagnostic_function errors_callback = diagnostic_function(),
        int max_recursion = 512)
    {
        // Turn the input range into a UTF-32 range, so that we can be sure
        // that we fall into the Unicode-aware parsing path inside parse()
        // below.
        auto const range = boost::parser::as_utf32(str);
        using iter_t = decltype(range.begin());

        if (max_recursion <= 0)
            max_recursion = INT_MAX;

        // Initialize our globals to the current depth (0), and the max depth
        // (max_recursion).
```

```
        global_state globals{0, max_recursion};
        bp::callback_error_handler error_handler(errors_callback);
        // Make a new parser that includes the globals and error handler.
        auto const parser = bp::with_error_handler(
            bp::with_globals(value_p, globals), error_handler);

        try {
            // Parse.  If no exception is thrown, due to: a failed expectation
            // (such as foo > bar, where foo matches the input, but then bar
            // cannot); or because the nesting depth is exceeded; we simply
            // return the result of the parse.  The result will contextually
            // convert to false if the parse failed.  Note that the
            // failed-expectation exception is caught internally, and used to
            // generate an error message.
            return bp::parse(range, parser, ws);
        } catch (excessive_nesting<iter_t> const & e) {
            // If we catch an excessive_nesting exception, just report it
            // and return an empty/failure result.
            if (errors_callback) {
                std::string const message = "error: Exceeded maximum number (" +
                                            std::to_string(max_recursion) +
                                            ") of open arrays and/or objects";
                std::stringstream ss;
                bp::write_formatted_message(
                    ss, "", range.begin(), e.iter, range.end(), message);
                errors_callback(ss.str());
            }
        }

        return {};
    }

}

std::string file_slurp(std::ifstream & ifs)
{
    std::string retval;
    while (ifs) {
        char const c = ifs.get();
        retval += c;
    }
    if (!retval.empty() && retval.back() == -1)
        retval.pop_back();
    return retval;
}

int main(int argc, char * argv[])
{
    if (argc < 2) {
        std::cerr << "A filename to parse is required.\n";
        exit(1);
```

```
    }

    std::ifstream ifs(argv[1]);
    if (!ifs) {
        std::cerr << "Unable to read file '" << argv[1] << "'.\n";
        exit(1);
    }

    // Read in the entire file.
    std::string const file_contents = file_slurp(ifs);
    // Parse the contents.  If there is an error, just stream it to cerr.
    auto json = json::parse(
        file_contents, [](std::string const & msg) { std::cerr << msg; });
    if (!json) {
        std::cerr << "Parse failure.\n";
        exit(1);
    }

    std::cout << "Parse successful; contents:\n" << *json << "\n";

    return 0;
}
```

1.2.2 Parsing JSON With Callbacks

This is just like the previous extended JSON parser example, except that it drops all the code that defines a JSON value, array, object, etc. It communicates events within the parse, and the value associated with each event. For instance, when a string is parsed, a callback is called that indicates this, along with the resulting `std::string`.

```
#include <boost/parser/parser.hpp>
#include <boost/parser/transcode_view.hpp>

#include <fstream>
#include <vector>
#include <climits>


namespace json {

    namespace bp = ::boost::parser;
    using namespace bp::literals;

    template<typename Iter>
    struct excessive_nesting : std::runtime_error
    {
        excessive_nesting(Iter it) :
            runtime_error("excessive_nesting"), iter(it)
        {}
```

```
    Iter iter;
};


struct global_state
{
    int recursive_open_count = 0;
    int max_recursive_open_count = 0;
};

struct double_escape_locals
{
    int first_surrogate = 0;
};


bp::rule<class ws> const ws = "whitespace";

bp::rule<class string_char, uint32_t> const string_char =
    "code point (code points <= U+001F must be escaped)";
bp::rule<class four_hex_digits, uint32_t> const hex_4 =
    "four hexadecimal digits";
bp::rule<class escape_seq, uint32_t> const escape_seq =
    "\\uXXXX hexadecimal escape sequence";
bp::rule<class escape_double_seq, uint32_t, double_escape_locals> const
    escape_double_seq = "\\uXXXX hexadecimal escape sequence";
bp::rule<class single_escaped_char, uint32_t> const single_escaped_char =
    "'\"', '\\', '/', 'b', 'f', 'n', 'r', or 't'";

bp::callback_rule<class null_tag> const null = "null";

// Since we don't create polymorphic values in this parse, we need to be
// able to report that we parsed a bool, so we need a callback rule for
// this.
bp::callback_rule<class bool_tag, bool> const bool_p = "boolean";

bp::callback_rule<class string_tag, std::string> const string = "string";
bp::callback_rule<class number_tag, double> const number = "number";

// object_element is broken up into the key (object_element_key) and the
// whole thing (object_element).  This was done because the value after
// the ':' may have many parts.  It may be an array, for example.  This
// implies that we need to report that we have the string part of the
// object-element, and that the rest -- the value -- is coming.
bp::callback_rule<class object_element_key_tag, std::string> const
    object_element_key = "string";
bp::rule<class object_element_tag> const object_element = "object-element";

// object gets broken up too, to enable the reporting of the beginning and
// end of the object when '{' or '}' is parsed, respectively.  The same
// thing is done for array, below.
```

```
bp::callback_rule<class object_open_tag> const object_open = "'{'";
bp::callback_rule<class object_close_tag> const object_close = "'}'";
bp::rule<class object_tag> const object = "object";

bp::callback_rule<class array_open_tag> const array_open = "'['";
bp::callback_rule<class array_close_tag> const array_close = "']'";
bp::rule<class array_tag> const array = "array";

// value no longer produces an attribute, and it has no callback either.
// Each individual possible kind of value (string, array, etc.) gets
// reported separately.
bp::rule<class value_tag> const value = "value";


// Since we use these tag types as function parameters in the callbacks,
// they need to be complete types.
class null_tag {};
class bool_tag {};
class string_tag {};
class number_tag {};
class object_element_key_tag {};
class object_open_tag {};
class object_close_tag {};
class array_open_tag {};
class array_close_tag {};


auto const ws_def = '\x09'_l | '\x0a' | '\x0d' | '\x20';

auto first_hex_escape = [](auto & ctx) {
    auto & locals = _locals(ctx);
    uint32_t const cu = _attr(ctx);
    if (!boost::parser::detail::text::high_surrogate(cu))
        _pass(ctx) = false;
    else
        locals.first_surrogate = cu;
};
auto second_hex_escape = [](auto & ctx) {
    auto & locals = _locals(ctx);
    uint32_t const cu = _attr(ctx);
    if (!boost::parser::detail::text::low_surrogate(cu)) {
        _pass(ctx) = false;
    } else {
        uint32_t const high_surrogate_min = 0xd800;
        uint32_t const low_surrogate_min = 0xdc00;
        uint32_t const surrogate_offset =
            0x10000 - (high_surrogate_min << 10) - low_surrogate_min;
        uint32_t const first_cu = locals.first_surrogate;
        _val(ctx) = (first_cu << 10) + cu + surrogate_offset;
    }
};
```

```
bp::parser_interface<bp::uint_parser<uint32_t, 16, 4, 4>> const hex_4_def;

auto const escape_seq_def = "\\u" > hex_4;

auto const escape_double_seq_def =
    escape_seq[first_hex_escape] >> escape_seq[second_hex_escape];

bp::symbols<uint32_t> const single_escaped_char_def = {
    {"\"", 0x0022u},
    {"\\", 0x005cu},
    {"/", 0x002fu},
    {"b", 0x0008u},
    {"f", 0x000cu},
    {"n", 0x000au},
    {"r", 0x000du},
    {"t", 0x0009u}};

auto const string_char_def = escape_double_seq | escape_seq |
                             ('\\'_l > single_escaped_char) |
                             (bp::cp - bp::char_(0x0000u, 0x001fu));

auto const null_def = "null"_l;

auto const bool_p_def = bp::bool_;

auto const string_def = bp::lexeme['"' >> *(string_char - '"') > '"'];

auto parse_double = [](auto & ctx) {
    auto const cp_range = _attr(ctx);
    auto cp_first = cp_range.begin();
    auto const cp_last = cp_range.end();

    auto const result = bp::prefix_parse(cp_first, cp_last, bp::double_);
    if (result) {
        _val(ctx) = *result;
    } else {
        // This would be more efficient if we used
        // boost::container::small_vector, or std::inplace_vector from
        // C++26.
        std::vector<char> chars(cp_first, cp_last);
        auto const chars_first = &*chars.begin();
        auto chars_last = chars_first + chars.size();
        _val(ctx) = std::strtod(chars_first, &chars_last);
    }
};

auto const number_def =
    bp::raw[bp::lexeme
                [-bp::char_('-') >>
                 (bp::char_('1', '9') >> *bp::digit | bp::char_('0')) >>
```

```
                -(bp::char_('.') >> +bp::digit) >>
                -(bp::char_("eE") >> -bp::char_("+-") >> +bp::digit)]]
            [parse_double];

    // The object_element_key parser is exactly the same as the string parser.
    // Note that we did *not* use string here, though; we used string_def.  If
    // we had used string, its callback would have been called first, and
    // worse still, since it moves its attribute, the callback for
    // object_element_key would always report the empty string, because the
    // string callback would have consumed it first.
    auto const object_element_key_def = string_def;

    auto const object_element_def = object_element_key > ':' > value;

    // This is a very straightforward way to write object_def when we know we
    // don't care about attribute-generating (non-callback) parsing.  If we
    // wanted to support both modes in one parser definition, we could have
    // written:
    //    auto const object_open_def = eps;
    //    auto const object_close_def = eps;
    //    auto const object_def = '{' >> object_open >>
    //                             -(object_element % ',') >
    //                            '}' >> object_close;
    auto const object_open_def = '{'_l;
    auto const object_close_def = '}'_l;
    auto const object_def = object_open >>
                            -(object_element % ',') > object_close;

    auto const array_open_def = '['_l;
    auto const array_close_def = ']'_l;
    auto const array_def = array_open >> -(value % ',') > array_close;

    auto const value_def = number | bool_p | null | string | array | object;

    BOOST_PARSER_DEFINE_RULES(
        ws,
        hex_4,
        escape_seq,
        escape_double_seq,
        single_escaped_char,
        string_char,
        null,
        bool_p,
        string,
        number,
        object_element_key,
        object_element,
        object_open,
        object_close,
        object,
        array_open,
```

```
        array_close,
        array,
        value);

    // The parse function loses its attribute from the return type; now the
    // return type is just bool.
    template<typename Callbacks>
    bool parse(
        std::string_view str,
        std::string_view filename,
        Callbacks const & callbacks,
        int max_recursion = 512)
    {
        auto const range = boost::parser::as_utf32(str);
        using iter_t = decltype(range.begin());

        if (max_recursion <= 0)
            max_recursion = INT_MAX;

        global_state globals{0, max_recursion};
        // This is a different error handler from the json.cpp example, just
        // to show different options.
        bp::stream_error_handler error_handler(filename);
        auto const parser = bp::with_error_handler(
            bp::with_globals(value, globals), error_handler);

        try {
            // This is identical to the parse() call in json.cpp, except that
            // it is callback_parse() instead, and it takes the callbacks
            // parameter.
            return bp::callback_parse(range, parser, ws, callbacks);
        } catch (excessive_nesting<iter_t> const & e) {
            std::string const message = "error: Exceeded maximum number (" +
                                        std::to_string(max_recursion) +
                                        ") of open arrays and/or objects";
            bp::write_formatted_message(
                std::cout,
                filename,
                range.begin(),
                e.iter,
                range.end(),
                message);
        }

        return {};
    }

}

std::string file_slurp(std::ifstream & ifs)
{
```

```
    std::string retval;
    while (ifs) {
        char const c = ifs.get();
        retval += c;
    }
    if (!retval.empty() && retval.back() == -1)
        retval.pop_back();
    return retval;
}

// This is our callbacks-struct.  It has a callback for each of the kinds of
// callback rules in our parser.  If one were missing, you'd get a pretty
// nasty template instantiation error.  Note that these are all const members;
// callback_parse() takes the callbacks object by constant reference.
struct json_callbacks
{
    void operator()(json::null_tag) const { std::cout << "JSON null value\n"; }
    void operator()(json::bool_tag, bool b) const
    {
        indent();
        std::cout << "JSON bool " << (b ? "true" : "false") << "\n";
    }
    void operator()(json::string_tag, std::string s) const
    {
        indent();
        std::cout << "JSON string \"" << s << "\"\n";
    }
    void operator()(json::number_tag, double d) const
    {
        indent();
        std::cout << "JSON number " << d << "\n";
    }
    void operator()(json::object_element_key_tag, std::string key) const
    {
        indent();
        std::cout << "JSON object element with key \"" << key
                  << "\" and value...\n";
    }
    void operator()(json::object_open_tag) const
    {
        indent(1);
        std::cout << "Beginning of JSON object.\n";
    }
    void operator()(json::object_close_tag) const
    {
        indent(-1);
        std::cout << "End of JSON object.\n";
    }
    void operator()(json::array_open_tag) const
    {
        indent(1);
```

```
        std::cout << "Beginning of JSON array.\n";
    }
    void operator()(json::array_close_tag) const
    {
        indent(-1);
        std::cout << "End of JSON array.\n";
    }

    void indent(int level_bump = 0) const
    {
        if (level_bump < 0)
            indent_.resize(indent_.size() - 2);
        std::cout << indent_;
        if (0 < level_bump)
            indent_ += "  ";
    }
    mutable std::string indent_;
};

int main(int argc, char * argv[])
{
    if (argc < 2) {
        std::cerr << "A filename to parse is required.\n";
        exit(1);
    }

    std::ifstream ifs(argv[1]);
    if (!ifs) {
        std::cerr << "Unable to read file '" << argv[1] << "'.\n";
        exit(1);
    }

    std::string const file_contents = file_slurp(ifs);
    bool success = json::parse(file_contents, argv[1], json_callbacks{});
    if (success) {
        std::cout << "Parse successful!\n";
    } else {
        std::cerr << "Parse failure.\n";
        exit(1);
    }

    return 0;
}
```

Note that here, I was keeping things simple to stay close to the previous parser. If you want to do callback parsing, you might want that because you're limited in how much memory you can allocate, or because the JSON you're parsing is really huge, and you only need to retain certain parts of it.

If this is the case, one possible change that might be appealing would be to reduce the memory allocations. The only memory allocation that the parser does is the one we told it to do — it allocates `std::strings`. If we instead used `boost::container::small_vector<char, 1024>`, it would only ever allocate if it encountered a

string larger than 1024 bytes. We would also want to change the callbacks to take `const &` parameters instead of using pass-by-value.