AXE 2.1

Library Reference

Introduction

AXE is a C++ library that provides facilities to build recursive descent parsers. A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures (or classes) where each such procedure (or class) implements one of the production rules of the grammar.[1]

AXE library contains a set of classes and functions to define syntax rules and semantic actions. The library uses standard C++17 and doesn't require compiler/platform specific facilities.[2] It's been tested with Visual C++ 2017 and expected to work with any other reasonably compliant C++17 compiler.

AXE is a header only library, it doesn't require linking. You only need to add
`#include <axe.h>`
in your source files and set the include directory in your compiler environment to point to `axe/include`.

[1]Recursive descent parser: Wikipedia
[2] GCC literal operator template implementation (n3599) used as an optimization, the other compilers use the standard C++

Changes in v.2

AXE v.2 introduced some significant changes compared to v.1

Version 2	Explanation
`compiler`	AXE v.2 requires reasonably conforming C++17 compiler. AXE v.1 required minimum C++11 compiler.
`Composite rules`	In AXE v.2 all composite rules take sub-rules by value. Which means that l-value rules are copied, and r-value rules are moved. In AXE v.1 the l-value sub-rules were held by reference. That created a problem of dangling references when returning a rule from a function. AXE v.2 parsers no longer have that limitation and can be returned from the functions. User can still hold sub-rules by reference using `std::reference_wrapper` by explicitly calling `std::ref`. **This change can break the v.1 code.** As a result of this change the recursive use of `axe::r_rule` class requires reference wrapping and the old code that didn't use reference wrapping will not work correctly as it will use copies in every recursive call.
`Lambda functions`	In AXE v.2 lambda functions used as rules and extractors are recognized via corresponding traits and no longer require `r_ref` and `e_ref` wrappers. The wrapper classes are retained for backward compatibility. Lambda functions in most cases should be polymorphic (auto parameter types or C++20 template lambda) to be recognized by trait class.
`Parser functions`	Many forwarding functions were eliminated due to the use of template class deduction guides. The rules look the same as before, but they no longer call the forwarding functions. The class names were changed to correspond to the previous function names. The new parser `axe::r_regex` to parse regular expressions was added. The parser uses `std::basic_regex` in implementation.
`Variadic rules`	AXE v.2 uses variadic templates for the composite types like r_and<…>, r_or<…>, etc. That provides for natural parse tree bindings and avoids deep nesting of types.
`Parse tree functions`	AXE v.2 introduced `parse_tree` function that returns a `result<Data, Iterator>::data` field which contains extracted data. There are additional convenience functions in axe_util.h header file to simplify extraction of data and writing it in xml format.
`Bug fixes and performance improvements`	Several bugs were fixed in the code, as well as some shortcut rules. The extractors for integral types performance was improved.

Syntax Rules

AXE library uses overloaded C++ operators and functions to approximate EBNF syntax, providing additional facilities for convenience and improved performance. Most of AXE operator and function rules are iterator agnostic and can be used to parse text and binary data in files, containers, etc. Some rules will accept input iterators, most rules require at least forward iterators.
AXE functions and classes are defined in **`axe`** namespace, operators are in **`axe::operators`** inline namespace, and shortcuts are in **`axe::shortcuts`** namespace.

AXE operator rules

Syntax	Explanation
`A & B`	Match rule A and B in specified order
`A && B`	Match rule A and B in any order (same as A & B | B & A)
`A > B`	Match A & B atomically, once matched A must match B (same as A & (B | r_fail()))
`A | B`	Match rule A or B in specified order
`A || B`	Match longest sequence of A and/or B (same as A & B | A | B)
`A ^ B`	Match permutations of A and B (same as A & ~B | B & ~A)
`A - B`	Match rule A but not B (same as !B & A)
`A % B`	Match A separated by B one or more times (same as r_many(A, B))
`+A`	Match A one or more times (same as r_many(A))
`*A`	Match A zero or more times (same as r_many(A,0))
`~A`	Match A zero or one times (same as r_many(A, 0, 1))
`!A`	Match negation of A; this rule never advances iterators
`A * n` `n * A`	Match rule A exactly n times (same as r_many(A, n, n))
`A/C`	New in AXE 2.1 Rule A is constrained by constraint C. If rule A is matched, then the Boolean constraint C is tested. If this test fails, then the whole composite rule also fails.

AXE Shortcuts

The following parser rules are included for convenience in namespace shortcuts:

Syntax	Explanation
_	Any character: axe::r_any()
_a	Single letter or _: axe::r_alpha()
_d	Decimal digit: axe::r_num()
_double	Double: axe::r_double()
_hs	Horizontal space: r_any(“ \t”)
_ident	Identifier: r_ident()
_int	Decimal integer: axe::r_decimal()
_n	New line: axe::r_char('\n')
_o	Octal digit: axe;r_oct()
_r	Carriage Return: axe::r_char('\r')
_s	Space separator: axe::r_char(‘ ‘)
_t	Tab character: axe::r_char('\t')
_uint	Decimal unsigned integer: axe::r_udecimal()
_w	Word: axe::alnumstr()
_ws	White space: “ \t\n\r”
_x	Hex digit: axe::r_hex()
_z	End of range: axe::r_end()
_endl	End of line: *_hs & (_n | _z)

AXE operator””

These convenience operators are included in namespace shortcuts:

Syntax	Explanation
“str”_axe	Creates `axe::r_str(“str”)` rule. For GCC (implementing n3599) this operator creates `r_strlit<’s’,’t’,’r’>` type object, which is a faster implementation.
“abc”_any	Creates `axe::r_any(“abc”)` rule
“abc”_regex	Creates `axe::r_regex(“abc”)` rule

AXE functions and class constructors

In most cases the above operators and shortcuts are sufficient for creating general parsing rules. The functions and constructors described below provide additional facilities for creating less common rules or providing more flexibility to the above operators.

Syntax	Explanation
`r_expression(T& t)` **`parse_expression(string, def_value)`**	**`r_expression`** is a rule that evaluates expressions and stores result in provided variable t. If parameter **`t`** is a floating point value each operand of the expression is evaluated as a floating point. Otherwise the operands are evaluated as integer values. **`r_expression`** doesn't allow spaces in the expression string. **`parse_expression`** function accepts an expression string and returns either expression value if evaluation succeeded, or **`def_value`** if it failed. The white-space characters are skipped.
`r_many(A, B, min_occurrence = 1, max_occurrence = -1)`	Matches rule A separated by rule B from min_occurrence to max_occurrence times. Operators *, +, % create r_many rule.
`r_many(A, min_occurrence = 1, max_occurrence = -1)`	Matches rule A from min_occurrence to max_occurrence times
`r_select(A, B, C)`	Matches rule A, and if matched matches B, otherwise matches C (equivalent to `!A & C | A & B`, but A is evaluated only once)
`r_ref(A)`	Deprecated. Creates a reference rule wrapper for rule A. An std::ref() function can be used instead.
`r_find(A)`	Skip input until rule **`A`** matched. When matched it returns iterator range starting from the initial position to the end of matched rule. If you want to find the first match of rule **`A`** but not advance iterator, then you can use **`r_find(!!A);`**

`r_fail(F)`	Creates a rule to invoke function F on failure. Function receives upto three iterators: beginning of range, failed position, end of range. Allowed signatures: `void()`, `void(Iterator)`, `void(Iterator, Iterator)`, and `void(Iterator,Iterator,Iterator)`.
	
	E.g.
	
	`A | r_fail([](auto i1, auto i2, auto i3)`
	`{`
	`  cerr << "parsing failed at position indicated by !\n"`
	`  << string(i1, i2) << '!' << string(i2, i3);`
	`});`

r_fail(string = "")	When rule fails `axe::failure<charT>` exception is thrown with specified string. The `charT` type depends on iterator type the rule was executed: `using charT = typename iterator_traits<Iterator>::value_type;`

r_bool(b)	Creates rule that evaluates boolean expression which becomes the result of match. The rule doesn't advance the iterators.
	Most often this rule used with lvalue references convertible to bool or lambda functions returning value convertible to bool. Since the argument is held by value, it often needs a reference wrapper to be evaluated at parse time.
	
	For example, the following rule matches only numbers from 1 to 999:
	`unsigned number = 0;`
	`auto rule = _int >> number & r_bool([&] { return number > 0 && number < 1000;});`

r_constrained(R,C)	AXE 2.1 and later. Creates a constrained rule. Rule R is evaluated and if matched constraint C is tested. Constrained rule is matched if and only if rule R is matched and constraint C is satisfied. To create a constrained rule, an operator/ can be used. A constrained rule in many situations supersedes `r_bool`. For example, `auto rule = _int /[](auto i1, auto i2) { auto number = get_as<int>(i1, i2); return number > 0 && number < 1000;};`
r_empty()	An empty rule always evaluates to true and doesn't advance the iterators
r_char(c)	Matches a single character **`c`**. In many composite rules characters are converted to **`r_char`**.
r_bin(t)	Matches a binary representation of standard layout type byte by byte
r_str(str)	Matches a character string **`str (const charT* or basic_string<charT>)`**. In many composite rules string literals are converted to **`r_str`**.
r_lit(a)	This rule is a synonym for **`r_char`**, **`r_str`**, or **`r_bin`** depending on type of parameter **`a`**. For example, `auto rule = r_lit('a') & r_lit("abc") & r_lit(123) & r_lit(1.23);`
r_test(A)	Matches rule A, but doesn't advance iterator (same as `!!A`)
r_end()	Matches end iterator, it is used to verify all input is consumed.
r_var(t)	Rule to assign to POD types. Reads from input `sizeof(t)` bytes and assigns to variable **`t`**
r_array(a)	Matches a static array of rules (**`std::array`**). Same as r_and<T…>.

r_range(iterator begin, iterator end)	Matches range specified by begin and end iterators
r_sequence(c, min_occurrence = 0, max_occurrence = -1)	Matches a sequence of rules specified number of times

r_advance(size)	Advances input by `size` elements. Fails if iterator range is shorter than `size` elements. This rule is usually used with negative rules to avoid matching the same rule multiple times. For example,
	`*(r_any() - R) & R; // matches R twice`
	`*(r_any() - (R >> e_length(s)) & r_advance(s); // matches R once`

r_ident()	Matches an identifier (`r_alpha() & r_alnumstr(0)`)
r_udecimal(t)	Matches unsigned decimal number, assigning value to t
r_decimal(t)	Matches signed decimal number, assigning value to t
r_ufixed(t)	Matches unsigned fixed point number, assigning value to t
r_fixed(t)	Matches signed fixed point number, assigning value to t
r_double(t)	Matches floating point number, assigning value to t
r_alpha()	Matches a single alpha character (‘A’-‘Z’,’a’-‘z’, _)
r_alphastr()	Matches a string of alpha characters
r_alphastr(occurrence)	Matches a string of alpha characters specified number of times
r_alphastr(min_occurrence, max_occurrence)	Matches a string of alphabetic characters specified number of times
r_num()	Matches a single digit character
r_numstr()	Matches a string of digits
r_numstr(occurrence)	Matches a string of digits specified number of times
r_numstr(min_occurrence, max_occurrence)	Matches a string of digits specified number of times
r_alnum()	Matches alpha-numeric character
r_alnumstr()	Matches a string of alpha-numeric characters
r_alnumstr(occurrence)	Matches a string of alpha-numeric characters specified number of times
r_alnumstr(min_occurrence, max_occurrence)	Matches a string of alpha-numeric characters specified number of times
r_oct()	Matched octodecimal character

`r_octstr()`	Matches a string of octodecimal characters
`r_octstr( occurrence)`	Matches a string of octodecimal characters specified number of times
`r_octstr( min_occurrence, max_occurrence)`	Matches a string of octodecimal characters specified number of times
`r_hex()`	Matches hexadecimal character
`r_hexstr()`	Matches a string of hexadecimal characters
`r_hexstr( occurrence)`	Matches a string of hexadecimal characters specified number of times
`r_hexstr( min_occurrence, max_occurrence)`	Matches a string of hexadecimal characters specified number of times
`r_printable()`	Matches a printable character
`r_printablestr()`	Matches a string of printable characters
`r_printablestr( occurrence)`	Matches a string of printable characters specified number of times
`r_printablestr( min_occurrence, max_occurrence)`	Matches a string of printable characters specified number of times
`r_any()`	Matches any character; often used with `operator-` to create “any char except” rule. E.g. *(_ - _s) matches any characters except space.
`r_any(c1, c2)`	Matches any character in the range [c1, c2]
`r_any(str)`	Matches any character found in the string str
`r_anystr(c1, c2)`	Matches a string of characters in the range [c1, c2]
`r_anystr(c1, c2, occurrence)`	Matches a string of characters in the range [c1, c2] specified number of times
`r_anystr(c1, c2, min_occurrence, max_occurrence)`	Matches a string of characters in the range [c1, c2] specified number of times
`r_anystr(str)`	Matches a string of characters found in the string str
`r_anystr(str, occurrence)`	Matches a string of characters found in the string str specified number of times
`r_anystr(str, min_occurrence, max_occurrence)`	Matches a string of characters found in the string str specified number of times
`r_pred(P)`	Matches a single character satisfying predicate P
`r_predstr(P)`	Matches a string of characters satisfying predicate P
`r_predstr(P, occurrence)`	Matches a string of characters satisfying predicate P specified number of times

r_predstr(P, min_occurrence, max_occurrence)	Matches a string of characters satisfying predicate P specified number of times
r_named(R, const char* name)	`r_named` rule assigns name to the rule `R`. This name is returned by `name()` function; `axe::operator>` uses this name to create an error message when the rule fails. E.g. `auto rule = r_lit("Hello") > r_named(r_lit("world"), "world rule");`
r_ref(R)	Rule reference wrapper is deprecated. The lambda functions can be automatically recognized as rules and no longer require wrapping. If you need to take a rule by reference (e.g. `r_rule<I>`) you can use `std::ref` function instead. This function is retained in this release for backward compatibility.
r_regex(Args&&…) **r_regex(const CharT*)** **r_regex(const basic_string&)**	A parser for regular expressions. Parser implemented using `std::basic_regex`, parameters in the constructor are forwarded to the `std::basic_regex` constructor. For this rule to return match, `std::regex_search` should return empty prefix, so regular expression should usually start with ^ (beginning of line).

AXE Iterator transformation rules

Transformation rules are used to modify the iterators to achieve specific purposes. The following iterator transformation rules are defined in AXE:

Syntax	Explanation
`r_skip(R, F)` **`r_skip(R, char)`** **`r_skip(R, const char*)`**	Rule creates skip-iterator, which omits all sequence elements satisfying F (it can be a predicate or a rule) and passes skip-iterator to the rule `R`. When F denotes a predicate, it is applied to a single character from input and if it returns true the character is skipped. When F denotes a rule, all characters matching the rule are skipped. This rule is usually used to skip white spaces, e.g. using a predicate: `r_skip(R, axe::is_wspace());` or using a rule: `r_skip(R, axe::r_any(" \t\n\r"));` This is a convenience rule, that allows to avoid specifying characters, like spaces in rules explicitly, though specifying skip characters in rules explicitly is more flexible and may result in better performance. Overloads of r_skip function which accept characters and character strings create iterators skipping specified characters, e.g. the following rule skips all white-space characters: `r_skip(R, " \t\n\r");`
`r_ucase(string, locale = std::locale())`	This rule transforms iterator to upper case iterator, using specified locale (or default locale). It can be used for case insensitive match converting all characters to upper-case, e.g. `auto rule = _int & r_ucase("L");`
`r_lcase(string, locale = std::locale())`	Same as above, but uses lower case transformation.
`r_icase(string, locale = std::locale())`	Case insensitive match. It transforms both string and iterators to lower case. This rule is slower than `r_ucase` or `r_lcase`, because both input characters and string are converted.
`r_convert(R, F)`	Creates a rule applying function F to convert iterator and pass it to rule R.
`r_buffered(R)`	Creates a buffer for input iterator, e.g. `istream_iterator`. Input iterators cannot be rolled back, so if a buffered rule is not matched an `axe::r_failure` exception is thrown. To minimize the amount of copying and the size of the buffer, only wrap the minimum necessary number of atomic rules in `r_buffered`.

AXE Predicate Functions

Syntax	Explanation
`is_alpha()`	Returns true for single alpha character (a letter or '_').
`is_num()`	Returns true for single decimal digit
`is_alnum()`	Returns true for single alpha or digit character
`is_hex()`	Returns true for single hex character
`is_oct()`	Returns true for single oct character
`is_printable()`	Returns true for single printable character
`is_space()`	Returns true for single space character (' ')
`is_wspace()`	Returns true for a single white-space character (" \t\n\r")
`is_char(c)`	Returns true if input matches character c
`is_any(c1, c2)`	Returns true if input matches any character in the range [c1, c2]
`is_any(const char* str)`	Returns true if input matches any character from the string
`is_any()`	Always returns true.

AXE Predicate Operators

Operators called on predicates return the following composite predicates:

Syntax	Explanation
`P1 && P2`	Evaluates to true if P1 and P2 evaluate to true
`P1 || P2`	Evaluates to true if P1 or P2 evaluate to true
`P1 ^ P2`	Evaluates to true if P1 or P2 evaluate to true, but not both
`!P`	Evaluates to true if P evaluates to false

AXE polymorphic rule

AXE library provides polymorphic type `axe::r_rule<Iterator>` for recursive rule definitions and runtime parser creation. It uses type erasure to assign any other parser rule to it. Polymorphic rule is often used in recursive parser definitions. Since AXE v.2.0 (unlike previous versions) when creating composite rules the sub-rules are copied or moved, so you would need to create a reference wrapper for recursive parsers, e.g.

```
axe::r_rule<Iterator> r;
auto a = "abc"_axe & std::ref(r);
r = "[" & _int | a "]";
```

AXE Parse Tree bindings

When `axe::parse_tree` function is used for a composite rule, it returns in `axe::result<D, I>::data` field the combination of the following data types:

Syntax	Explanation
`A & B & …`	`std::tuple<DA, DB, …>` You can use `std::get<N>` function to extract individual data fields.
`A | B | …`	`std::variant<DA, DB,…>` You would normally use `std::visit` function to extract individual data.
`*A, +A, A % B, r_many(A,…)`	`std::vector<DA>` The separator rules are not extracted.
`A - …`	`std::tuple<…, DA>` The last element of the tuple is the extracted data from rule A.
`~A`	`std::optional<DA>`
`A > B > …`	`std::tuple<DA, DB, …>`
`r_find<A>, r_named<A>`	`DA` -- the same type as for rule A.
All other rules, including custom rules that do not provide parse tree extraction	`axe::it_pair<I>`, which holds begin and end iterators of parsed sequence.

AXE Utility functions

The following functions are defined in the axe namespace in axe_utility.h:

Syntax	Explanation
`template<class R, class Txt>` `auto parse(R&& r, Txt&& txt)`	Function applies the rule r to the container txt and returns `result<Iterator>.`
`template<class R, class I>` `auto parse(R&& r, I begin, I end)`	Same as the above, but the function takes the iterators begin and end. An alternative is to call operator() on the rule directly: `r(begin, end);`
`template<class R, class Txt>` `auto parse_tree(R&& r, Txt&& txt)`	Function applies the rule **`r`** to the container **`txt`** and returns `result<Data, Iterator>.` The field `result<Data, I>::data` contains the parse tree for this operation. See AXE Parse Tree bindings below.
`template<class R, class I>` `auto parse_tree(R&& r, I begin, I end)`	Same as the above, but the function takes the iterators begin and end.
`template<class D, class Data>` `auto get_as(const Data& data)`	Function uses AXE extractors to convert `result<Data, I>::data` field passed as an argument to specified type `D`. Function can convert tuple, variant, vector, optional, it_pair data types that comprise parse-tree data. AXE extractors provide converters to the arithmetic types, string types, and a stream based converter. If none of the AXE extractors can convert the parse-tree data to the requested type D a compilation error will be reported.
`template<class R>` `inline auto get_name(const R& r)`	Returns name if it's a named rule. Otherwise returns `type_info::name.` Named rules is a debugging convenience.
`template<class T, class C, class Traits, class Alloc>` `inline auto parse_expression(const std::basic_string<C, Traits, Alloc>& str, T def_value)`	Function parses the expression in string `str` using `axe::r_expression` parser. Returns parsed value of type **`T`** if successful. Otherwise returns `def_value`.
`auto write_xml(const` *`parse-tree-data-type`*`& v, std::ostream& os, size_t level = 0)`	Function writes `result<Data, I>::data` field passed as argument to the stream in xml format. Parameter `level` is used to create indentation. For example, `axe::write_xml(axe::parse_tree(*axe::r_find(".o."_re-gex), "The quick brown fox jumps over the lazy dog").data, std::cout);` the output:

Syntax	Explanation
	<vector size="4"> <![CDATA[row]]> <![CDATA[fox]]> <![CDATA[ov]]> <![CDATA[dog]]> </vector>

Creating custom predicates

Custom predicate is a class or lambda function which defines `operator()`, taking a parameter of character type and returning `bool`. Predicates are usually used with **`r_pred`** and **`r_predstr`** function to create corresponding rules.

For `custom` class to be considered a predicate the trait `axe::is_predicate<custom>::value` must be true.

For example, `axe::is_num` is a predicate that matches a single digit:

```
struct is_num
{
      bool operator()(char c) const { return c >= '0' && c <= '9'; }
};
```

Lambda functions can also be used to create a predicate. For example,

```
auto is_num = [](auto c) { return c >= '0' && c <= '9'; }
auto number_rule = axe::r_pred(is_num);
auto numbers_rule = axe::r_predstr(is_num);
```

Creating custom rules

AXE library provides many facilities to create parsing rules using shortcuts, operators, and lambda functions, so you almost never need to create a custom rule class. Nevertheless, if such need arises, this section explains how you can create a custom rule class.

Custom rule is a class or lambda function which defines const `operator()`. There are two supported overloads of this operator, one of which creates parse tree and the other doesn't.

The simplest form of operator(), which doesn't create parse tree, takes a pair of iterators and returns `axe::result<Iterator>` class.

If custom rule is extracting a parse tree then it must additionally define `operator()` which takes `axe::it_pair` parameter and returning `axe::result<Data, Iterator>` type. The data field in the return type should contain extracted value of one of supported types (see: AXE Parse Tree bindings).

For `custom` class to be considered a rule the trait `axe::is_rule_v<custom>` must be true. Automatic rule detection is limited to a few predefined iterator types, so in general case `operator()` should be a template to be recognized as a rule. In some cases the return type of `operator()` must be explicit `axe::result<Iterator>` and not `auto` or `decltype(auto)`. The reason for that are the rule traits, which may not work with `auto` return type. Compiler forces instantiation of the functions with `auto` return type which can result in hard error instead of SFINAE.

E.g. custom rule matching new line:

```
class custom
{
public:
	template<class Iterator>
	axe::result<Iterator> operator()(Iterator i1, Iterator i2) const
	{
		// match new line
		bool matched = i1 != i2 && *i1 == '\n';
		// return result, advancing iterator if matched
		return axe::result(matched, matched ? std::next(i1): i1);
	}
};

void test()
{
	static_assert(axe::is_rule_v<custom>, "Error: Custom is not a rule");
}
```

Lambda functions can also be used as rules. They must take two auto parameters (or C++20 lambda template) to be recognized as rules.

E.g.

```
auto custom = [](auto i1, auto i2)
{
	return axe::result(i1 != i2 && *i1 == '\n', i1 + 1, i1);
};
```

Recursive rules

AXE being a recursive descent parser doesn't allow left recursion[3]. But the right recursion in custom rules is possible.
When writing template `operator()`, it can be inlined in the body of the rule class. The dependent names in `operator()` of the rule are looked up at the point of instantiation. After forward declaring a custom rule, the name can then appear in the template `operator()` of other rules before the type of that custom rule is complete.
For example, the AXE rule for `r_expression_t` is forward declared and the name is used in `r_group_t` rule and called recursively:

```
template<class T> struct r_expression;

template<class T>
struct r_group_t
{
   // skipped ...
    template<class It>
    result<It> operator() (It i1, It i2)
```

[3]Left recursion: Wikipedia

```
    {
        return ('(' & r_expression<T>(value_) & ')')(i1, i2);
    }
};

template<class T>
struct r_factor_t
{
   // skipped ...
    template<class It>
    result<It> operator() (It i1, It i2)
    {
        return (r_decimal(value_) | r_group_t<T>(value_))(i1, i2);
    }
};

template<class T>
struct r_term_t
{
   // skipped ...
    template<class It>
    result<It> operator() (It i1, It i2)
    {
        return (r_factor_t<T>(value_) & // skipped …
        ))(i1, i2);
    }
};

template<class T>
struct r_expression
{
   // skipped ...
    template<class It>
    result<It> operator() (It i1, It i2)
    {
        return
        (r_term_t<T>(value_) & // skipped ...
        )(i1, i2);
    }
};
```

For details of `r_expression` implementation see `axe_expression.h` file. There is also a convenience function defined in this file, which parses expression and returns value if successful or `def_value` if parsing failed:

```
template<class T, class C, class Traits, class Alloc>
inline auto parse_expression(const std::basic_string<C, Traits, Alloc>& str, T def_value)
```

Though lambda functions do not allow recursive definition[4], standard polymorphic functions (`std::function`) can be used recursively for rule definition. For example,

```
std::function<axe::result<const char*> (const char*, const char*)> block
    = [&](const char* i1, const char* i2)
{ return (';'_axe | '{' & std::ref(block) & '}')(i1,i2); };
```

AXE provide a polymorphic rule `axe::r_rule`, which can also be used to define recursive rules. One

[4] You can define lambda taking another lambda as a parameter, then proving the same lambda as the parameter, thus creating recursion. This usually would fail when creating parsing rules though, because compiler wouldn't be able to deduce the return type for such lambda causing rule traits to fail, and providing the return type explicitly is impossible in most situations.

thing to remember is that composite rules keep sub-rules by value, so `axe::r_rule` must be wrapped in `reference_wrapper` using `std::ref` function (`std::reference_wrapper` is recognized as a rule) in order to refer to the same function object.

For example,

```
axe::r_rule<I> json_value;
auto json_array = json_spaces & '['
    & axe::r_many(json_spaces & std::ref(json_value) & json_spaces, ',', 0)
    & json_spaces & ']';
json_value = json_string | axe::r_double(d) | json_object | json_array
    | "true" | "false" | "null";
```

Semantic Actions

There are two ways to extract data from a parser. One is to use parse_tree function and get the whole parse tree in `axe::result<Data, Iterator>::data` field. The other way is using an in-place extractor, which creates an extractor rule. Both approaches have their pluses and minuses.

AXE in-place extractors

Syntax	Explanation

`A >> E`	Operator >> creates an extractor rule.
	If rule A is matched, the iterator range for that rule is passed to extractor E. The extractor can accept zero, one, two, or three arguments of Iterator type. If the extractor accepts one argument then the iterator pointing to the beginning of matched rule will be passed to the function.
	If the extractor accepts two arguments, the beginning and the end of the matched rule are passed.
	If the extractor accepts three arguments then in addition to previous a third iterator to the end of the input sequence is passed.
	Extractors can be chained, each one receiving the matched range of the rule. They are invoked in the same order as the associativity of the operators, so a later invoked extractor can use the result of the previously executed one.
	For example, the following rule will extract and print the value and length of an identifier.
	
	`std::string name;`
	`size_t length(0);`
	
	`auto rule = r_ident() >> name >> e_length(length);`
	
	`parse(rule, "an_identifier"s);`
	`std::cout << "parsed identifier: " <<  name`
	`<< " is " << length << " characters long";`

`A >> e_ref(E)`	Extractor reference wrapper is deprecated. The lambda functions can be automatically recognized as extractors and no longer require wrapping. This function is retained for backward compatibility.
`A >> e_length(t)`	Extractor assigns length of matched range to t.
`A >> e_push_back(c)`	Extractor calls **`c.push_back(value)`** for each extracted value.
`r_fail(F)`	`r_fail(F)` is a wrapper for a function F, which is called when the preceding rule has failed. The function F can accept zero, one, two, or three arguments. If the function accepts one argument the iterator pointing to the failed position is passed to it. If the function accepts two arguments the iterator pointing to the failed position and the iterator to the end of the sequence are passed to it. If the function accepts three arguments the iterator pointing to the beginning of the sequence, the iterator pointing to the failed position, and the iterator to the end of the sequence are passed to it. This function is always used with `operator |` to create a rule. For example,

	auto on_fail = []
	{
	  std::cerr << "Error: expecting identifier";
	}
	
	auto rule = r_ident() | r_fail(on_fail);

Creating custom semantic actions

Custom semantic action is a class which defines `operator()`, taking zero, one, two, or three iterators and returning void. For `custom` class to be considered an extractor the trait `axe::is_extractor_v<custom>` must be true. In most cases the operator() must be a template to be recognized by the trait class.

E.g. custom extractor counting new lines:

```
class custom
{
      size_t& count_;
public:
      custom(size_t& count) : count_(count) {}
      template<class Iterator>
      void operator()(Iterator i1, Iterator i2)
      {
            if(i1 != i2 && *i1 == '\n') ++count_;
      }
};

void test()
{
      static_assert(axe::is_extractor_v<custom>,
                        "Error: Custom is not an extractor");
}
```

Lambda functions can also be used as semantic actions. They must take `auto` parameters to be recognized by `axe::is_extractor_v` trait.

E.g.

```
size_t count = 0;
auto custom = [&](auto i1, auto i2)
{
      if(i1 != i2 && *i1 == '\n') ++count;
};
using namespace axe::shortcuts;
parse(*(_ >> custom), "count\neol\nin this text\n");
```

Using parse tree

In addition to in-place extractors AXE can create a parse tree for parser.

In-place extractors are easy to use and produce faster parsers. But they become part of the rule, reducing the flexibility of using the rule in different contexts. They also practically always bind to l-values receiving the result. You must be careful not to return such parsers from (lambda) functions, because the references will become dangling when the l-values go out of scope.

The alternative way of extracting data from a parse result is using a parse tree. The parse tree is returned as a data field in `axe::result<Data, Iterator>` struct. This field is created based on the rules described in ***AXE Parse Tree bindings*** table.

When you call `axe::parse_tree` function the data field will contain the parse tree. That allows to separate the rules from data extraction, so the rules can be reused in different context requiring different extraction. It also simplifies returning rules from (lambda) functions because they don't have to deal with possibility of dangling references.

The cost you pay for using the parse tree is slower execution because during the parsing process the parse tree data keeps growing and is returned on the function stack. It's also more difficult to reason about the extractor as you must correctly recognize the bindings by looking at the rule. If you do it incorrectly compiler will likely stop you with a long error message. Another inconvenience of parse tree is that it makes the rule less flexible to changes, because parse tree extractor and the rule must be modified in sync.

To simplify the process of extraction from the parse tree, AXE provides `get_as` function template:

```
template<class D, class Data>
auto get_as(const Data& data);
```

- where `D` is the type you want to convert to, and data is the `result<Data, Iterator>::data` field returned from the `parse_tree` function.

This function is overloaded for all known parse tree data types and can convert to arithmetic, string, and other stream-able types using the same extractor functions that are used for in-place extraction.

You can also write an output in xml format for the whole parse tree by using the `write_xml` function:

```
auto write_xml(const parse-tree-data& data, std::ostream& os, size_t level = 0);
```

The xml tags used correspond to the data type of the node in the parse tree: `<tuple>`, `<variant>`, `<vector>`, `<optional>`, and the parsed text is written as `<![CDATA[parsed-text]]>`

See Examples section for ways of using parse tree extraction.

Examples

Examples below omit the include files and the using declarations:

```
using namespace axe;
using namespace axe::shortcuts;
```

Zip code parser

Zip codes are a system of postal codes used by the United States Postal Service (USPS) since 1963. The basic format consists of five decimal numerical digits. An extended ZIP+4 code, introduced in the 1980s, includes the five digits of the ZIP code, a hyphen, and four more digits that determine a more precise location than the ZIP code alone.[5]

Therefore, we can write the following rule to match a ZIP code:

```
auto zip_rule = _d * 5 & ~('-' & _d * 4) & !_d;
```

where:

`_d * 5` expression creates a rule that matches exactly 5 digits

`~('-' & _d * 4)` expression creates a rule that optionally matches '-' followed by 4 digits

`!_d` expression creates a rule that test the next character without advancing the iterator, and matches everything except a digit.

The following rule matches (and skips) any portion of the text that is not `zip_rule`: `*(_ - zip_rule)`

To skip any portion of the text that is not `zip_rule` and then extract string from `zip_rule`, we can write this rule: `*(_ - zip_rule) & zip_rule >> [](auto i1, auto i2) {…}`

The following function extracts and prints zip codes found in the text:

```
void print_zip(const std::string& text)
{
    // rule to match zip code
    auto zip_rule = _d * 5 & ~('-' & _d * 4) & !_d;

    // skip everything that is not zip_rule and extract zip_rule
    auto text_rule = *(*(_ - zip_rule) & zip_rule
        >> [](auto i1, auto i2)
    {
        std::cout << std::string(i1, i2) << "\n";
    });

    parse(text_rule, text);

    std::cout << "Now using the parse tree\n";

    // the same parse rule without the in-place extractor
    auto text_rule1 = *(*(_ - zip_rule) & zip_rule);

    // this is an example of using a parse tree to extract data
    // the rule above produces vector of tuple,
    // we are only interested in the second value in that tuple corresponding to zip_rule
```

[5]ZIP code: Wikipedia

```
    for (auto& d : parse_tree(text_rule1, text).data)
    {
        std::cout << axe::get_as<std::string>(std::get<1>(d)) << "\n";
    }
}

// a function to run a test of the parser rules created above

void test_zip()
{
    std::cout << "------------------------------------------------------------test_zip:\n";
    auto zips = std::string(R"*(94302 Zip Code
94303-1011 Zip Code
94309 Zip Code
94301 Zip Code
94304 Zip Code
94306 Zip Code
94307)*"
);
    print_zip(zips);
    std::cout << "------------------------------------------------------------------\n";
}
```

Telephone number parser

The traditional convention for phone numbers [in United States] is (AAA) BBB-BBBB, where AAA is the area code and BBB-BBBB is the subscriber number. The format AAA-BBB-BBBB or sometimes 1-AAA-BBB-BBBB is often seen. Sometimes the stylized format of AAA.BBB.BBBB is seen.[6]

We can write the following four rules corresponding to these conventions:

```
    // four common telephone formats
    auto tel1 = '(' & _d * 3 & ')' & _d * 3 & '-' & _d * 4;
    auto tel2 = _d * 3 & '-' & _d * 3 & '-' & _d * 4;
    auto tel3 = "1-" & _d * 3 & '-' & _d * 3 & '-' & _d * 4;
    auto tel4 = _d * 3 & '.' & _d * 3 & '.' & _d * 4;
```

The following function prints telephone numbers found in the text. As a demonstration of constrained rule, a list of numbers, which don't satisfy the specified constraint, is excluded. A constraint lambda function checks the set of excluded numbers and returns true if a number is not excluded.

```
//-----------------------------------------------------------------------------
void print_tel(const std::string& text)
{
    // four common telephone formats
    auto tel1 = '(' & _d * 3 & ')' & _d * 3 &'-' & _d * 4;
    auto tel2 = _d * 3 & '-' & _d * 3 & '-' & _d * 4;
    auto tel3 = "1-" & _d * 3 & '-' & _d * 3 & '-' & _d * 4;
    auto tel4 = _d * 3 & '.' & _d * 3 & '.' & _d * 4;

       // exclude known fax numbers
       const std::set<std::string> excluded_numbers{"408-278-7444", "650-617-3120"};
```

[6]Local conventions for writing telephone numbers: Wikipedia

```
        auto tel2_filtered = tel2 / [&](auto i1, auto i2)
        {
                return excluded_numbers.count(std::string(i1, i2)) == 0;
        };

        auto tel = tel1 | tel2_filtered | tel3 | tel4;
    // extracting rule
    auto text_rule = *(*(_ - tel) & tel
        >> [](auto i1, auto i2)
    {
        slog << std::string(i1, i2) << "\n";
    });

    parse(text_rule, text);

    slog << "Using parse_tree function:\n";

    // using parse tree for data extraction
    for (auto p : parse_tree(*(*(_ - tel) & tel), text).data)
    {
        slog << get_as<std::string>(std::get<1>(p)) << "\n";
    }
}

void test_tel()
{
    slog << "------------------------------------------------------------test_tel:\n";
    auto tels = std::string(R"*(call City Hall at (650)329-2100
Police Desk at 329-2406
Phone: 650-329-2258, Fax: 650-617-3120
1505 Meridian, San Jose, CA
Phone: 408-278-7400
Fax: 408-278-7444
94307)*");
    print_tel(tels);
    slog << "-----------------------------------------------------------------------\n";
}
```

CSV parser

This example shows how to create a parser for comma-separated values (csv) format[7]. In this example we allow any printable characters (except comma character) inside the values. Additionally, we allow spaces in the beginning and the end of each value, which are removed during parsing. The spaces inside string values are preserved. The program below prints each extracted value in angle brackets:

```
<Year><Make><Model><Trim><Length>
<2010><Ford><E350><Wagon Passenger Van><212.0>
<2011><Toyota><Tundra><CREWMAX><228.7>
```

```
auto csv(const std::string& text)
{
    // comma separator including trailing spaces
    auto cvs_separator = *_hs & ',';
    // rule for comma separated value
    auto csv_value = *_hs & +(_ - cvs_separator - _endl)
```

[7]Comma-separated values: Wikipedia

```
        >> [](auto i1, auto i2)
    {
        std::cout << "<" << std::string(i1, i2) << ">";
    };

    // rule for single string of comma separated values
    auto line = csv_value % cvs_separator
        & _endl >> [] { std::cout << "\n"; };

    // file contaning multiple csv lines
    auto csv_file = +line & _z
        | axe::r_fail([](auto i1, auto i2)
        {
            std::cout << "\nFailed: " << std::string(i1, i2);
        });

    parse(csv_file, text);
}

void test_cvs()
{
    std::cout << "------------------------------------------------------------test_cvs:\n";
    auto text = std::string(R"*(Year, Make, Model, Trim, Length
2017,Ford,E350, Wagon Passenger Van ,212.0
2018 , Toyota, Tundra, CREWMAX, 228.7 )*");
    csv(text);
    std::cout << "-----------------------------------------------------------------\n";
}
```

INI File parser

INI file was once popular format for configuration files on the Windows and other platforms[8].

This example demonstrates how one can create a parser for INI file, which parses properties, sections, and comments.

Simplified EBNF rule for INI file is this:
ini_file ::= {comment}* {section {property | comment}*}*

The following function parses INI record, specified by a pair of iterators, and outputs INI records converted to xml.

```
void ini(I begin, I end)
{
    using namespace axe::shortcuts;

    auto close_section = [] (const std::string& section)
    {
        if (!section.empty())
        {
            std::cout << "</" << section << ">\n";
        }
    };

    std::string open_section;
    std::string section_name;
```

[8]INI file: Wikipedia

```
    // section rule, can end with traling spaces
    auto section = *_hs & ('[' & _ident >> section_name & ']')  >> [&]
    {
        close_section(open_section);
        open_section = section_name;
        std::cout << "<" << open_section << ">\n";
    } & _endl;

    // key name rule, can contain any characters, except '=' and spaces
    std::string key;
    auto key_rule = *_ws & +(_ - '=' - _endl - _hs) >> key;

    // value rules
    std::string value;

    // unquoted simple key value
    auto simple_value = *_hs & *(_ - _endl) >> value;

    // quoted value may have escaped quote chars
    auto quoted_value = *_hs & '"' & (*(_ - "\\\"" - '"') % +"\\\""_axe) >> value & '"';

    // key value can either be quoted or unquoted
    auto value_rule = quoted_value | simple_value;

    // rule for property line
    auto prop_line = key_rule & *_hs & '=' & value_rule & _endl >> [&]
    {
        std::cout << "\t<property key=\"" << key << "\" value=\"" << value << "\" />\n";
    };

    // rule for comment
    auto comment = *_ws & ';' & *(_ - _endl) & _endl;

    // rule for INI file
    auto ini_file = *comment & *(section & *(prop_line | comment)) & *_ws & _z
        >> [&] { close_section(open_section); }
        | axe::r_fail([](auto i1, auto i2, auto i3)
    {
        std::cerr << "\nIni file contains errors, indicated by !\n";
        std::cerr << std::string(i1, i2) << "!" << std::string(i2, i3);
    });

    // do parsing
    ini_file(begin, end);
}
```

Now we can test this parser on a simple example:

```
void test_ini()
{
    std::string text{ R"**(
    ; This is a test of ini file
    [section1]
    key1 = 5
    key2 = value
    ; this is comment
    [section2]
    key = " quoted value: \"3\" "
```

```
)**" };
    ini(text.begin(), text.end());
}
```

It should print the following:

<section1>
    <property key="key1" value="5" />
    <property key="key2" value="value" />
</section1>
<section2>
    <property key="key" value=" quoted value: \"3\" " />
</section2>

Command line parser

In this example we create a parser for command line and create semantic actions to extract keys and parameters.

This command line contains executable name, followed by optional key-value pairs, followed by optional parameters. In EBNF the grammar would look like this:

```
command_line ::= executable {"-" key [ = value]}* {parameter}*
```

The following function parses such command line and prints key-value pairs and parameters:

```
void parse_cmd_line(const std::string& cmd)
{
    using namespace axe;
    using namespace axe::shortcuts;

    // data to be gathered
    std::string exe_name;
    std::map<std::string, std::string> key_map;
    std::vector<std::string> parameters;

    // executable rule: file.extension
    auto executable = r_alnumstr() & *('.' & r_alnumstr());

    // key rule
    std::string key_name;
    auto key = r_alnumstr() >> key_name >> [&]() { key_map[key_name]; };

    // value rule
    auto value = r_alnumstr() >> [&](auto i1, auto i2)
    { key_map[key_name] = std::string(i1, i2); };

    // parameter rule
    auto parameter = r_alnumstr() >> e_push_back(parameters);

    // rule for key-value pairs
    auto key_value = '-' & key & ~(*_s & '=' & *_s & value);
    auto command_line = executable >> exe_name
        & *(+_s & key_value)
        & *(+_s & parameter) & _endl
        | r_fail([](auto, auto i2, auto i3)
    {
```

```
        std::cout << "Failed to parse portion of command line: " << std::string(i2, i3)
            << std::endl;
    });

    auto result = command_line(cmd.begin(), cmd.end());

    if (result.matched)
    {
        std::cout << "Matched string: " << std::string(cmd.begin(), result.position) << std::endl;
        std::cout << "Executable: " << exe_name << std::endl;
        std::cout << "Key-value pairs:" << std::endl;
        for (auto& i : key_map)
            std::cout << "\t" << i.first << " = " << i.second << std::endl;
        std::cout << "Parameters:" << std::endl;
        for (auto& p : parameters)
            std::cout << "\t" << p << std::endl;
    }
}

void test_cmd()
{
    std::cout << "------------------------------------------------------------test_cmd:\n";
    parse_cmd_line("command.exe -t=123 -h -p = p_value one two three");
    std::cout << "-------------------------------------------------------------------\n";
}
```

Windows Path Parser

This example shows how to create a parser for windows path format. Windows path starts with a letter followed by ':' or with "\\" followed by server name. Any characters are allowed, except `"/?<>\\:*|\""`[9]

The path can be enclosed in double quotes, in which case it can contain spaces. The following function extracts and prints all paths found in text.

```
template<class I>
void print_paths(I i1, I i2)
{
    using namespace axe;
    using namespace axe::shortcuts;
    // spaces are allowed in quoted paths only
    // illegal path characters
    auto illegal = r_any("/?<>\\:*|\"");
    // end of line characters
    auto endl = r_any("\n\r");
    // define path characters
    auto path_chars = _ - illegal - _hs - endl;
    // windows path can start with a server name or letter
    auto start_server = "\\\\" & +path_chars - '\\';
    auto start_drive = r_alpha() & ':';
    auto simple_path = (start_server | start_drive) & *('\\' & +path_chars);
    auto quoted_path = '"' & (start_server | start_drive) &
        *('\\' & +(_hs | path_chars)) & '"';
    // path can be either simple or quoted
    auto path = simple_path | quoted_path;
    // rule to extract all paths
    std::vector<std::wstring> paths;
    size_t length = 0;
```

[9]Naming Files, Paths, and Namespaces

```
    auto extract_paths = *(*(r_any() - (path >> e_push_back(paths) >> e_length(length)))
        & r_advance(length));
    // perform extraction
    extract_paths(i1, i2);
    // print extracted paths
    std::wcout << L"\nExtracted paths:\n";
    std::for_each(paths.begin(), paths.end(),
        [](const std::wstring& s)
    {
        std::wcout << s << L'\n';
    });
}
```

Roman Numerals

Roman numerals stem from the numeral system of ancient Rome. They are based on certain letters of the alphabet which are combined to signify the sum (or, in some cases, the difference) of their values[10]. Parsing roman numerals can be done in different ways. This example shows a simple parser, which takes a string of roman numerals separated by spaces, converts and prints them.

```
void parse_roman(const std::string& txt)
{
    auto result = 0u;
    // thousands
    auto thousands = r_many("M"_axe >> [&result]{ result += 1000; }, 0, 3);
    // hundreds
    auto value100 = "C"_axe >> [&result] { result += 100; };
    auto value400 = "CD"_axe >> [&result] { result += 400; };
    auto value500 = "D"_axe >> [&result] { result += 500; };
    auto value900 = "CM"_axe >> [&result] { result += 900; };
    auto hundreds = value400 | value900 | ~value500 & r_many(value100, 0, 3);
    // tens
    auto value10 = "X"_axe >> [&result] { result += 10; };
    auto value40 = "XL"_axe >> [&result] { result += 40; };
    auto value50 = "L"_axe >> [&result] { result += 50; };
    auto value90 = "XC"_axe >> [&result] { result += 90; };
    auto tens = value90 | value40 | ~value50 & r_many(value10, 0, 3);
    // ones
    auto value1 = "I"_axe >> [&result] { result += 1; };
    auto value4 = "IV"_axe >> [&result] { result += 4; };
    auto value5 = "V"_axe >> [&result] { result += 5; };
    auto value9 = "IX"_axe >> [&result] { result += 9; };
    auto ones = value9 | value4 | ~value5 & r_many(value1, 0, 3);
    // a string of roman numerals separated by spacces
    auto roman = ((thousands & ~hundreds & ~tens & ~ones)
        >> [&result] { std::cout << result << ' '; result = 0; })
        % _s;

    parse(roman, txt);
}

void test_roman()
{
    // test parser
    std::cout << "------------------------------------------------------------test_roman:\n";
    std::string txt("I MMCCCLVI MMMCXXIII XXIII LVI MMMCDLVII DCCLXXXVI DCCCXCIX MMMDLXVII");
    std::cout << txt << "\n";
    parse_roman(txt);
```

[10]Roman numerals: Wikipedia

```
    std::cout << "\n-------------------------------------------------------------------\n";
}
```

```
Output:
-------------------------------------------------------------test_roman:
I MMCCCLVI MMMCXXIII XXIII LVI MMMCDLVII DCCLXXXVI DCCCXCIX MMMDLXVII
1 2356 3123 23 56 3457 786 899 3567
-------------------------------------------------------------------
```

JSON Parser

JSON is a lightweight text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for most languages.[11]

This example demonstrates how to use polymorphic parser rule `axe::r_rule` to define recursive rules.

```
template<class I>
void parse_json(I begin, I end)
{
    using namespace axe::shortcuts;

    auto json_hex = axe::r_many(axe::r_hex(), 4);
    auto json_escaped = "\""_axe | '\\' | '/' | 'b' | 'f'
        | 'n' | 'r' | 't' | 'u' & json_hex;
    auto json_char = _ - '"' - '\\' | '\\' & json_escaped;
    auto json_string = '"' & *json_char & '"';
    // definition of json_value requires recursion
    // neither 'auto' declaration, nor lambda functions allow recursion
    // instead one can create function object with recursion in operator()
    // or use polymorphic r_rule class, which performs type erasure
    axe::r_rule<I> json_value;

    // json_value must be wrapped with std::ref because rules are taken by value
    auto json_array = *_ws & '['
        & (*_ws & std::ref(json_value) & *_ws) % ','
        & *_ws & ']';

    auto json_record = *_ws & json_string & *_ws
        & ':' & *_ws & std::ref(json_value) & *_ws;

    auto json_object = *_ws & '{' & json_record % ',' & *_ws & '}';

    json_value = json_string | _double | json_object | json_array
        | "true" | "false" | "null";

    parse(json_object >> [](auto i1, auto i2)
    {
        std::cout << "JSON object parsed: " << std::string(i1, i2);
    } & _z
        | axe::r_fail([](auto i1, auto i2, auto i3)
    {
        std::cout << "parsing failed at place pointed by !\n"
            << std::string(i1, i2) << '!' << std::string(i2, i3);
    }), begin, end);
}
```

[11] JSON: Wikipedia

```
void test_json()
{
    std::cout << "------------------------------------------------------------test_json:\n";
    std::string str(R"*(
{
"category": 1,
"sub-category": 1.1,
"name": "inventory",
"tags": ["warehouse","inventory"],
"vehicles" :
[
        {
        "id": 123456789,
        "make": "Honda",
        "model": "Ridgeline",
        "trim": "RTL",
        "price": 32616,
        "tags": ["truck","V6","4WD"]
        },
        {
        "id": 748201836,
        "make": "Honda",
        "model": "Pilot",
        "trim": "Touring",
        "price": 38042,
        "tags": ["SUV","V6","4WD"]
        }
]
})*");

    parse_json(str.begin(), str.end());
    std::cout << "\n-----------------------------------------------------------------------\n";
}
```

Running this program produces the following parsed text:

```
------------------------------------------------------------test_json:
JSON object parsed:
{
"category": 1,
"sub-category": 1.1,
"name": "inventory",
"tags": ["warehouse","inventory"],
"vehicles" :
[
        {
        "id": 123456789,
        "make": "Honda",
        "model": "Ridgeline",
        "trim": "RTL",
        "price": 32616,
        "tags": ["truck","V6","4WD"]
        },
        {
        "id": 748201836,
        "make": "Honda",
        "model": "Pilot",
        "trim": "Touring",
        "price": 38042,
        "tags": ["SUV","V6","4WD"]
        }
]
}
```

Replacement Parser

This example demonstrates how to create a parser to replace the the portions of a container matching specified rule with the content of another container. The function returns a pair, consisting of a new container and number of occurrences of target rule in the source container.

```
template<class Container, class Rule>
auto replace(const Container& source, Rule r, const Container& rep)
{
    using namespace axe;
    using namespace axe::shortcuts;

    Container result;
    size_t counter = 0;

    // copy_rule matches all characters except r and copies them in result
    auto copy_rule = *(_ - r) >> [&](auto i1, auto i2)
    {
        result.insert(result.end(), i1, i2);
    };
    // subst_rule matches r and copies replacement in result
    auto subst_rule = r >> [&](auto i1, auto i2)
    {
        result.insert(result.end(), rep.begin(), rep.end());
        ++counter;
    };
    // replace_rule is the rule to process source
    auto replace_rule = *subst_rule & *(copy_rule & subst_rule) & _z;
    // perform parsing and replacement
    replace_rule(source.begin(), source.end());
    // return result and number of occurrences of r replaced
    return std::make_tuple(std::move(result), counter);
}
```

Now using this function, we can write a function to replace all occurrences of a target string in the source string with replacement string.

```
auto replace(const std::string& source,
    const std::string& target,
    const std::string& replacement)
{
    return replace(source, axe::r_str(target), replacement);
}

void test_replacement()
{
    std::cout << "------------------------------------------------------------test_replacement:\n";
    auto source = R"*(
    This example shows how to replace all occurencies of abracadabra
    with something good. You should only see abracadabra in the output string.
    )*";

    std::cout << source;
    auto replacement = replace(source, "abracadabra", "something good");
    std::cout << "\nreplaced " << std::get<1>(replacement) << " occurences\n";
    std::cout << std::get<0>(replacement);
    std::cout << "\n-----------------------------------------------------------------\n";
}
```

A word counting program

This example demonstrates how to create a parser for word counting program. An island parser for this program consists of three rules: parsing words, numbers, and all other symbols. The parsers for those are _w, +_d, and

```
auto other = +(_ - _n - _w - +_d); // rule matching everything else
```

To make this program more useful, we also collect the dictionaries matching these rules. To do that we use extractors with lambda functions, like this:

```
+_d >> // extractor to count numbers
       [this](auto i1, auto i2)
       {
           ++nnumber_;
           numbers_.insert(token_t(i1, i2));
       }
```

When rule is matched, extractor with specified lambda function is called inserting token into the dictionary and incrementing the number.

The full code for this program is shown below.

```
using token_t = std::vector<unsigned char>;

class wc
{
    // counts
    unsigned nline_{ 0 }, nword_{ 0 }, nnumber_{ 0 }, nother_{ 0 };
    // dictionaries
    std::set<token_t> words_;
    std::set<token_t> numbers_;
    std::set<token_t> other_;

    friend std::ostream& operator<< (std::ostream& os, const wc& w);

public:
    template<class I>
    wc& operator() (I begin, I end)
    {
        using namespace axe::shortcuts;

        auto other = +(_ - _n - _w - +_d); // rule matching everything else

        auto file = // rule for parsing input file
            *(
                other >> // extractor to count other symbols
                [this](auto i1, auto i2)
                {
                    if(i1 != i2)
                    {
                        ++nother_;
                        other_.insert(token_t(i1, i2));
                    }
                }
                | _n >> [this](auto...) { ++nline_; }
                | +_d >> // extractor to count numbers
                    [this](auto i1, auto i2)
                    {
                        ++nnumber_;
                        numbers_.insert(token_t(i1, i2));
```

```
                    }
                | _w >> // extractor to count words
                    [this](auto i1, auto i2)
                    {
                        ++nword_;
                        words_.insert(token_t(i1, i2));
                    }
                        )
            & ~_ws
            & _z;

        auto p = axe::parse(file, begin, end); // do the actual parsing

        return *this;
    }
};

std::ostream& operator<< (std::ostream& os, const wc& w)
{
    // calculate dictionary size
    auto get_size = [](const std::set<token_t>& s)
    {
        return std::accumulate(s.begin(), s.end(), 0ull,
        [](auto v, const token_t& str)
        {
            return v + str.size();
        });
    };
    // report statistics
    os  << "\nlines:\t " << w.nline_
        << "\nwords:\t " << w.nword_ << ", unique: " << w.words_.size()
        << ", dictionary size: " << get_size(w.words_)
        << "\nnumbers: " << w.nnumber_  << ", unique: " << w.numbers_.size()
        << ", dictionary size: " << get_size(w.numbers_)
        << "\nother:\t " << w.nother_ << ", unique: " << w.other_.size()
        << ", dictionary size: " << get_size(w.other_)
        << "\n";
#ifndef NDEBUG
    os << "Unique words:";
    for(auto&& w : w.words_)
        os << "\n\t" << std::string(w.begin(), w.end());
    os << "\nUnique numbers:";
    for(auto&& w : w.numbers_)
        os << "\n\t" << std::string(w.begin(), w.end());
#endif
    return os;
}

int main(int argc, char* argv[])
{
    if(argc == 2)
    {
        std::ifstream s(argv[1], std::ios_base::binary);
        if(s)
        {
            std::istreambuf_iterator<char> begin(s.rdbuf()), end;
            token_t vec(begin, end);

            std::cout << "Read " << vec.size() << " characters from " << argv[1];
            std::cout.flush();
            std::cout << wc()(vec.begin(), vec.end()) << "\n";
        }
        else
        {
```

```
            std::cerr << "ERROR: failed to open file: " << argv[1] << "\n";
        }
    }
    else
    {
        std::cerr << "\nusage: " << argv[0] << " <filename>\n";
    }

    return 0;
}
```

Wildcard parser

This example demonstrates how you can create a parser dynamically. The `wildcard` function takes a string, which contain wildcard characters (asterisks)[12] and creates an AXE parser corresponding to it, and then it returns the parser from the function. In order to create the parser dynamically we can use a polymorphic rule `axe::r_rule<I>` and while parsing the user input use extractors to build the final rule.

Here is one way it can be implemented:

```
auto wildcard(const std::string& target)
{
    using namespace axe::shortcuts;
    using I = decltype(target.begin());
    axe::r_rule<I> result;

    // match any number of chars except '*'
    auto match_string = +(_ - '*');

    // make a rule to match an extracted string
    auto make_str_rule = [](auto i1, auto i2)
    {
        return axe::r_str(std::string(i1, i2));
    };

    // match wildcard starting with '*'
    auto first_rule = match_string >> [&](auto i1, auto i2)
    {
        result = make_str_rule(i1, i2);
    };

    // match wilcard starting with '*' followed by a string
    auto second_rule = "*"_axe & match_string >> [&](auto i1, auto i2)
    {
        result = result & axe::r_find(make_str_rule(i1, i2));
    };

    // match wildcard ending with '*'
    auto third_rule = "*"_axe >> [&]
    {
        result = result & *_;
    };

    auto wc_rule = first_rule & *second_rule & ~third_rule
        | +second_rule & ~third_rule
        | third_rule;
```

[12] Wikipedia: Wildcard character

```
    if (!parse(wc_rule & _z, target).matched)
        throw std::runtime_error("invalid wildcard: " + target);

    return result;
}
```

The above code parses wildcard target string and replaces normal characters with AXE `r_str` rules and asterisks with AXE `r_find` rules, which skip over the sequence of characters that do not match the substring.

Now we can write a test finding all substrings matching a given wildcard, similar to grep utility:

```
void test_wildcard()
{
    std::cout << "------------------------------------------------------------test_wildcard:\n";
    auto grep = [](auto& wc, auto& text)
    {
        std::cout << "\n> grep \"" << wc << "\"\n";
        auto print = [](auto i1, auto i2)
        {
            std::cout << std::string(i1, i2) << "\n";
        };

        // find all substrings matching the wildcards
        auto res = axe::parse(+axe::r_find(wildcard(wc) >> print), text);
        if (!res.matched)
            std::cout << wc << " -- not found\n";
    };

    const std::string text(R"**(The asterisk in a wildcard matches any character zero or more
times.
For example, "comp*" matches anything beginning with "comp" which means
"comp", "complete", and "computer" are all matched.)**");

    std::cout << text << "\n";

    grep("asterisk*wildcard", text);
    grep("*asterisk*times", text);
    grep("\"comp*\"", text);
    grep("abracadabra", text);
    std::cout << "\n-----------------------------------------------------------------\n";
}
```

The `test_wildcard` function will print what the following results:

--test_wildcard:
The asterisk in a wildcard matches any character zero or more times.
For example, "comp*" matches anything beginning with "comp" which means
"comp", "complete", and "computer" are all matched.

> grep "asterisk*wildcard"
asterisk in a **wildcard**

> grep "*asterisk*times"
The **asterisk** in a wildcard matches any character zero or more **times**

```
> grep ""comp*""
"comp*"
"comp"
"comp"
"complete"
"computer"

> grep "abracadabra"
abracadabra -- not found
```

Doxygenated sources

http://www.gbresearch.com/axe/dox/