AXE 1.5

Library Reference

Table of Contents

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 C++11 facilities and requires C++11 compiler. It's been tested with Visual C++ 2010 and gcc 4.6.0

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`.

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.

AXE polymorphic rule

AXE library provides polymorphic type **`axe::r_rule<Iterator>`** for recursive rule definitions. It uses type erasure to assign any other parser rule to it.

1 Recursive descent parser: Wikipedia

AXE operator rules

<table>
<tr><th>Syntax</th><th>Explanation</th></tr>
<tr><td>A & B</td><td>Match rule A and B in specified order</td></tr>
<tr><td>A && B</td><td>Match rule A and B in any order (same as A & B | B & A)</td></tr>
<tr><td>A > B</td><td>Match A & B atomically (same as A & (B | r_fail()))</td></tr>
<tr><td>A | B</td><td>Match rule A or B in specified order</td></tr>
<tr><td>A || B</td><td>Match longest sequence of A and/or B (same as A & B | A | B)</td></tr>
<tr><td>A ^ B</td><td>Match permutations of A and B (same as A & ~B | B & ~A)</td></tr>
<tr><td>A - B</td><td>Match rule A but not B (same as !B & A)</td></tr>
<tr><td>A % B</td><td>Match A separated by B one or more times (same as r_many(A, B))</td></tr>
<tr><td>+A</td><td>Match A one or more times (same as r_many(A))</td></tr>
<tr><td>*A</td><td>Match A zero or more times (same as r_many(A,0))</td></tr>
<tr><td>~A</td><td>Match A zero or one times (same as r_many(A, 0, 1))</td></tr>
<tr><td>!A</td><td>Match negation of A; this rule never advances iterators</td></tr>
</table>

AXE function rules

<table>
<tr><th>Syntax</th><th>Explanation</th></tr>
<tr><td>r_expression(T& t)</td><td>Matches arithmetic expression, evaluates it, and stores the result in t</td></tr>
<tr><td>r_many(A, B,
min_occurrence = 1,
max_occurrence = -1)</td><td>Matches rule A separated by rule B from min_occurrence to max_occurrence times.</td></tr>
<tr><td>r_many(A,
min_occurrence = 1,
max_occurrence = -1)</td><td>Matches rule A from min_occurrence to max_occurrence times</td></tr>
<tr><td>r_select(A, B, C)</td><td>Matches rule A, and if matched matches B, otherwise matches C (equivalent to <code>!A & C | A & B</code>, but A is matched only once)</td></tr>
<tr><td>r_ref(A)</td><td>Creates a reference rule wrapper for rule A. Most often used as a wrapper for functions, functors, and lambda functions. For example,
<code>auto skip_char = r_ref([](Iterator i1, Iterator i2)</code>
<code>{ return make_result(i1 != i2, i1 + 1, i2); });</code></td></tr>
<tr><td>r_find(A)</td><td>Skip input until rule A matched. When matched it returns iterator range starting from the initial position to the end of matched rule.</td></tr>
<tr><td>r_fail(F)</td><td>Creates a rule to invoke function F on failure. Function receives iterators specifying failed range. E.g. <code>A | r_fail([](I i1, I i2)</code>
<code>{ cerr << "parsing failed: " << string(i1, i2); });</code></td></tr>
</table>

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

r_bool(b)	Creates rule that evaluates boolean b when matching. Most often used with lvalue references convertible to bool or lambda functions returning value convertible to bool. For example, the following rule matches only numbers from 1 to 999:
	`unsigned number = 0;`
	`auto rule = r_numstr() >> number & r_bool([&](){ return number > 0 && number < 1000;});`

r_empty()	An empty rule always evaluates to true
r_token(t)	Matches a single token, e.g. `auto rule = r_token("abc");`
r_char(c)	Matches a single character **c**
r_bin(t)	Matches a binary representation of value **t**
r_str(str)	Matches a character string **str (const charT* or basic_string<charT>)**
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 was consumed.
r_var(t)	Reads from input `sizeof(t)` bytes and assigns to variable **t**.
r_array(a)	Matches a static array of rules (**std::array**)
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
`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
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");`

AXE Iterator transformation rules

The following parser rules transform iterators:

Syntax	Explanation
r_skip(R, F) **r_skip(R, char)**	Rule creates skip iterator, which omits all sequence elements satisfying F (it can be a predicate or a rule) and passes iterator to rule `R`. When F denotes a predicate, it is applied to a single character from input and if return value is

r_skip(R, const char*)	true a 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_space());` or using a rule: `r_skip(R, axe::r_any(" \t\n"));` 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 likely to 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 space characters: `r_skip(R, " \t\n");`
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, 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, 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 Shortcuts

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

Syntax	Explanation
_	Any character: axe::r_any()
_d	Decimal digit: axe::r_num()
_double	Double: axe::r_double()
_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_pred(axe::is_space());
_t	Tab character: axe::r_char('\t')
_uint	Decimal unsigned integer: axe::r_udecimal()

`_w`	Word: axe::alnum() | '_'
`_ws`	White space: axe::r_char(' ')
`_x`	Hex digit: axe::r_hex()
`_z`	End of range: axe::r_end()

AXE Predicate Functions

Syntax	Explanation
`is_alpha()`	Returns true for single alpha character
`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 (' ', '\n', '\r', '\t')
`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(string)`	Returns true if input matches any character from the string
`is_any()`	Returns true if input matches any character

AXE Predicate Operators

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

Predicate functions and operators are usually used with `r_pred` and `r_predstr` function to create rules.

Creating custom predicates

Custom predicate is a class which defines `operator()`, taking a parameter of character type and returning `bool`.[2]

2 When using gcc compiler, the predicates must derive from axe::p_base classs. A macro `AXE_PREDICATE` is defined for all supported compilers.

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

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

```
auto is_num = [](char c)->bool { return c >= '0' && c <= '9'; }
```

Creating custom rules

Custom rule is a class which defines const `operator()`, taking a pair of iterators and returning `axe::result` class.[3]

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

E.g. custom rule matching new line:

```
class custom AXE_RULE
{
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::make_result(matched, i1 + 1, i1);
	}
};

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

Lambda functions can also be used as rules. They must be wrapped in `r_ref` in order to satisfy `is_rule<>` trait.

E.g.

3 When using gcc compiler, the rules must derive from axe::r_base classs. A macro `AXE_RULE` is defined for all supported compilers.

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

AXE being a recursive descent parser doesn't allow left recursion[4]. 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_t;

template<class T>
struct r_group_t AXE_RULE
{
   // skipped ...
    template<class It>
    result<It> operator() (It i1, It i2)
    {
        return (r_lit('(') & r_expression_t<T>(value_) & ')')(i1, i2);
    }
};

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

4 [Left recursion: Wikipedia]()

```
        return
        (r_term_t<T>(value_) & // skipped ...
        )(i1, i2);
    }
};
```

In case of non-template `operator()`, it must be implemented outside of rule class in cpp file after all other rule classes are defined.
Though lambda functions do not allow recursive definition, polymorphic functions can be used recursively. For example,

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

Polymorphic rule `axe::r_rule` can also be used to define recursive rules. For example,

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

Semantic Actions

AXE defines the following operators and functions to perform semantic actions.

AXE action operators and functions

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. Extractors can be chained, each one receiving the matched range of the rule. For example, the following rule will extract and print the value and length of an identifier.
	
	`std::string input = "an_identifier";`
	`std::string name;`
	`size_t length(0);`
	
	`auto rule = r_ident() >> name >> e_length(length);`
	
	`rule(input.begin(), input.end());`
	
	`std::cout << "parsed identifier: " <<  name`
	`<< " is " << length << " characters long";`

`e_ref(E)`	Extractor reference wrapper, can be used with function, functors, and lambda functions to create extractors. For example, `void print_match(...) { cout << "zip_rule matched" << endl; }`

	`void e_ref_example()`
	`{`
	`   auto zip_rule = r_numstr(5) >> e_ref(print_match);`
	`   // or using lambda`
	`   auto zip_rule1 = r_numstr(5)`
	`   >> e_ref([](...) {cout << "zip_rule matched" << endl; });`
	`}`

`e_length(t)`	Extractor assigns length of matched range to t
`e_push_back(c)`	Extractor calls **`c.push_back(v)`** for each extracted value

`r_fail(F)`	`r_fail(F)` is a wrapper for action F, which is called when the preceding rule has failed. It 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 a pair of iterators and returning void. For `custom` class to be considered an extractor the trait `axe::is_extractor<custom>::value` must be true.[5]

E.g. custom extractor counting new lines:

```
class custom AXE_EXTRACTOR
{
      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<custom>::value,
                        "Error: Custom is not an extractor");
}
```

Lambda functions can also be used as semantic actions. They must be wrapped in `e_ref` in order to

5 When using gcc compiler, the semantic actions must derive from axe::e_base classs. A macro `AXE_EXTRACTOR` is defined for all supported compilers.

satisfy `is_extractor<>` trait.

E.g.

```
size_t count = 0;
auto custom = axe::e_ref([](const char* i1, const char* i2)
{
      if(i1 != i2 && *i1 == '\n') ++count;
});
```

Examples

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.[6]

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

```
auto zip = r_numstr(5) & ~('-' & r_numstr(4));
```

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

```
auto zip_extractor = e_ref([](const char* i1, const char* i2)
{
   std::cout << std::string(i1, i2) << std::endl;
});

auto zip_rule = *(*(r_any() - r_num()) & ~(zip >> zip_extractor | r_any()));
```

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.[7]

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

```
auto tel1 = '(' & r_numstr(3) & ')' & r_numstr(3) & '-' & r_numstr(4);
auto tel2 = r_numstr(3) & '-' & r_numstr(3) & '-' & r_numstr(4);
auto tel3 = "1-" & r_numstr(3) & '-' & r_numstr(3) & '-' & r_numstr(4);
auto tel4 = r_numstr(3) & '.' & r_numstr(3) & '.' & r_numstr(4);
```

The following function matches telephone number or prints error if not matched:

```
void print_tel_number(const char* begin, const char* end)
{
      auto tel = tel1 | tel2 | tel3 | tel4;
      auto tel_rule = tel >> e_ref([](const char* i1, const char* i2)
      {
            std::cout << "parsed number is: " << std::string(i1, i2);
```

6 ZIP code: Wikipedia
7 Local conventions for writing telephone numbers: Wikipedia

```
        })
        | r_fail([](...)
        {
              std::cout << "Error: telephone number was not matched";
        });
        tel_rule(begin, end);
}
```

CSV parser

This example shows how to create a parser for comma-separated values (csv) format[8]. 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>
```

```
#include <iostream>
#include <sstream>
#include <axe\axe.h>

#pragma warning(disable:4503)

template<class I>
void csv(I begin, I end)
{
    // define comma rule
    auto comma = axe::r_lit(',');
    // endl matches end of line symbol
    auto endl = axe::r_lit('\n');
    // space matches ' ' or '\t'
    auto space = axe::r_any(" \t");
    // trailing spaces
    auto trailing_spaces = *space & (comma | endl);
    std::string value;
    // create extractor to print matched value
    auto print_value = axe::e_ref([&value](I, I)
    {
        std::cout << "<" << value << ">";
    });
    // rule for comma separated value
    auto csv_str = *space & +(axe::r_printable() - trailing_spaces)
        >>  value & *space;
    // rule for single string of comma separated values
    auto line = *((csv_str & comma) >> print_value)
        & csv_str >> print_value
        & endl >> axe::e_ref([](I, I)
    {
        std::cout << "\n";
    });
    // file containing many csv lines
    auto csv_file = +line
```

8 Comma-separated values: Wikipedia

```
        | axe::r_fail([](I i1, I i2) {
            std::cout << "\nFailed: " << std::string(i1, i2);
    });
    csv_file(begin, end);
}

int main(int argc, const char* argv[])
{
    std::stringstream ss;
    ss << "Year, Make, Model, Trim, Length " << std::endl;
    ss << "2010,Ford,E350, Wagon Passenger Van ,212.0" << std::endl;
    ss << "2011 , Toyota, Tundra, CREWMAX, 228.7 " << std::endl;
    std::string str = ss.str();
    csv(str.begin(), str.end());
    std::cin.ignore();
    return 0;
}
```

INI File parser

INI file was once popular format for configuration files on the Windows and other platforms[9]. 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. In this example, a simple semantic action `e_cout` prints parsed comment, section, and property rules to `cout`.

```
template<class I>
void ini(I begin, I end)
{
    // semantic rule to print matched range
    auto e_cout = axe::e_ref([](I i1, I i2) {std::cout << "\n>" << std::string(i1, i2);});
    // endl rule to match end of line symbol or end of iterator range
    auto endl = axe::r_lit('\n') | axe::r_end();
    // space rule
    auto space = axe::r_any(" \t");
    // trailing spaces
    auto trailing_spaces = *space & endl;
    // section name is any alpha-numeric string
    auto sec_name = axe::r_alnumstr();
    // section rule, can end with traling spaces
    auto section = ('[' & sec_name & ']') >> e_cout & trailing_spaces;
    // key name rule, can contain any characters, except '=' and spaces
    auto key_name = +(axe::r_any() - '=' - endl - space);
    // unquoted raw key value
    auto raw_key_value = *(axe::r_any() - trailing_spaces);
    // quoted key value
    auto quoted_key_value = '"' & *("\\\"" | axe::r_any() - '"') & '"';
    // key value can either be unquoted or quoted
    auto key_value = quoted_key_value | raw_key_value;
    // rule for property line
    auto prop_line = (*space & key_name & *space & '=' & *space
        & key_value & trailing_spaces) >> e_cout;
    // rule for comment
    auto comment = (';' & *(axe::r_any() - endl) & endl) >> e_cout;
    // rule for INI file
    auto ini_file = *comment & *(section & *(prop_line | comment)) & axe::r_end()
        | axe::r_fail([](I i1, I i2)
    {
```

9 INI file: Wikipedia

```
        std::cerr << "\nIni file invalid. Parsed portion follows:\n"
            << std::string(i1, i2);
    });
    // perform matching
    ini_file(begin, end);
}
```

Now we can test this parser on a simple example:

```
void test_ini()
{
    std::cout << "\ntest_ini()\n";
    std::stringstream ss;
    ss << "; This is a test of ini file" << std::endl;
    ss << "[section1]\n";
    ss << "key1=value1\n";
    ss << "key2 = value2 \n";
    ss << "; this is comment\n";
    ss << "[section2]\n";
    ss << "key3 = \" v a l u e 3   \"";
    std::string str = ss.str();
    ini(str.begin(), str.end());
}
```

It should print the following:
test_ini()

>; This is a test of ini file

>[section1]
>key1=value1

>key2 = value2

>; this is comment

>[section2]
>key3 = " v a l u e 3   "

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:

```
#include <string>
#include <string.h>
#include <map>
#include <vector>
#pragma warning(disable:4503)
#include <axe.h>

void parse_cmd_line(const char* cmd)
{
    using namespace axe;
    std::string exe_name;
    std::map<std::string, std::string> key_map; // key-value pairs
    std::vector<std::string> parameters;

    auto executable = r_alnumstr() & *('.' & r_alnumstr());
    std::string key_name;
    auto key = r_alnumstr() >> key_name
       >> e_ref([&](const char*, const char*) { key_map[key_name]; });
    auto value = r_alnumstr() >> e_ref([&](const char* i1, const char* i2)
       { key_map[key_name] = std::string(i1, i2); });
    auto parameter = r_alnumstr() >> e_push_back(parameters);
    auto space = r_lit(' ');
    auto key_value = '-' & key & ~(*space & '=' & *space & value);

    auto command_line = executable  >> exe_name
        & *(+space & key_value)
        & *(+space & parameter)
        | r_fail([](const char* i1, const char* i2)
    { std::cout << "Failed to parse command line: " << std::string(i1, i2) << std::endl; });

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

int main(int argc, const char* argv[])
{
    parse_cmd_line("command.exe -t=123 -h -p = p_value one two three");
}
```

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 `"/?<>\\:*|\""`[10]

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

10 Naming Files, Paths, and Namespaces

```
template<class I>
void print_paths(I i1, I i2)
{
    using namespace axe;
    // spaces are allowed in quoted paths only
    auto space = r_any(" \t");
    // illegal path characters
    auto illegal = r_any("/?<>\\:*|\"");
    // end of line characters
    auto endl = r_any("\n\r");
    // define path characters
    auto path_chars = r_any() - illegal - space - 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) &
       *('\\' & +(space | 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;
    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[11]. 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.

```
    using namespace axe;
    unsigned result = 0;

    // thousands
    auto thousands = r_many(r_lit('M') >> e_ref([&result](...){ result+= 1000; }),
        0, 3);
    // hundreds
    auto value100 = r_lit('C') >> e_ref([&result](...){ result+= 100; });
    auto value400 = r_lit("CD") >> e_ref([&result](...){ result+= 400; });
    auto value500 = r_lit('D') >> e_ref([&result](...){ result+= 500; });
    auto value900 = r_lit("CM") >> e_ref([&result](...){ result+= 900; });
```

11 Roman numerals: Wikipedia

```
    auto hundreds = value400 | value900 | ~value500 & r_many(value100, 0, 3);
    // tens
    auto value10 = r_lit('X') >> e_ref([&result](...) { result += 10; });
    auto value40 = r_lit("XL") >> e_ref([&result](...) { result += 40; });
    auto value50 = r_lit("L") >> e_ref([&result](...) { result += 50; });
    auto value90 = r_lit("XC") >> e_ref([&result](...) { result += 90; });
    auto tens = value90 | value40 | ~value50 & r_many(value10, 0, 3);
    // ones
    auto value1 = r_lit('I') >> e_ref([&result](...) { result += 1; });
    auto value4 = r_lit("IV") >> e_ref([&result](...) { result += 4; });
    auto value5 = r_lit("V") >> e_ref([&result](...) { result += 5; });
    auto value9 = r_lit("IX") >> e_ref([&result](...) { result += 9; });
    auto ones = value9 | value4 |  ~value5 & r_many(value1, 0, 3);
    // a string of roman numerals separated by spacces
    auto spaces = +r_lit(' ');
    auto roman = ((thousands & ~hundreds & ~tens & ~ones)
        >> e_ref([&result](...) { std::cout << result << ' '; result = 0; }))
        % spaces;
    // test parser
    std::ostringstream text;
    text << "I MMCCCLVI MMMCXXIII XXIII LVI MMMCDLVII DCCLXXXVI DCCCXCIX  MMMDLXVII";
    std::string str = text.str();
    roman(str.begin(), str.end());
```

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.[12]

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)
{
    auto json_spaces = *axe::r_any(" \t\n\r");
    auto json_hex = axe::r_many(axe::r_hex(), 4);
    auto json_escaped = axe::r_lit('"') | '\\' | '/' | 'b' | 'f'
        | 'n' | 'r' | 't' | 'u' & json_hex;
    auto json_char =  axe::r_any() - '"' - '\\' | '\\' & json_escaped;
    auto json_string = '"' & *json_char & '"';
    // definition of json_value requires recursion
    // neither 'auto' declation, nor lambda functions allow recursion
    // instead one can create classes with recursive operator()
    // or use polymorphic r_rule class
    axe::r_rule<I> json_value;
    auto json_array = json_spaces & '['
        & axe::r_many(json_spaces & json_value & json_spaces, ',', 0)
        & json_spaces & ']';
    double d;
    auto json_object = json_spaces & '{'
        & axe::r_many(json_spaces & json_string & json_spaces
        & ':' & json_spaces & json_value & json_spaces, ',', 0)
        & json_spaces &'}';
    json_value = json_string | axe::r_double(d) | json_object | json_array
        | "true" | "false" | "null";

    (json_object >> axe::e_ref([](I i1, I i2) {
        std::cout << "JSON object parsed: " << std::string(i1, i2);
```

12 JSON: Wikipedia

```
    }) & axe::r_end())(begin, end);
}

int main(int argc, const char* argv[])
{
    std::ostringstream os;
    os << "{\n";
    os << "\"category\": 1,\n";
    os << "\"name\": \"inventory\",\n";
    os << "\"tags\": [\"warehouse\",\"inventory\"],\n";
    os << "\"vehicles\" :\n";
    os << "[\n";
    os << "\t{\n";
    os << "\t\"id\": 123456789,\n";
    os << "\t\"make\": \"Honda\",\n";
    os << "\t\"model\": \"Ridgeline\",\n";
    os << "\t\"trim\": \"RTL\",\n";
    os << "\t\"price\": 32616,\n";
    os << "\t\"tags\": [\"truck\",\"V6\",\"4WD\"]\n";
    os << "\t},\n";
    os << "\t{\n";
    os << "\t\"id\": 748201836,\n";
    os << "\t\"make\": \"Honda\",\n";
    os << "\t\"model\": \"Pilot\",\n";
    os << "\t\"trim\": \"Touring\",\n";
    os << "\t\"price\": 38042,\n";
    os << "\t\"tags\": [\"SUV\",\"V6\",\"4WD\"]\n";
    os << "\t}\n";
    os << "]\n";
    os << "}\n";
    std::string str = os.str();
    parse_json(str.begin(), str.end());
}
```

Running this program produces the following parsed text:

```
JSON object parsed: {
"category": 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.

```
#include <axe.h>
#include <string>
#include <tuple>

using namespace axe;
using namespace axe::shortcuts;

template<class Container, class Rule>
std::tuple<Container, size_t> replace(const Container& source, Rule r, const Container& rep)
{
    Container result;
    size_t counter = 0;

    // the type of source iterator
    typedef decltype(source.begin()) I;

    // copy_rule matches all characters except r and copies them in result
    auto copy_rule = *(_ - r) >> axe::e_ref([&](I i1, I i2)
    {
        result.insert(result.end(), i1, i2);
    });

    // subst_rule matches r and copies replacement in result
    auto subst_rule = r >> axe::e_ref([&](I i1, I 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(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.

```
std::tuple<std::string, size_t> replace(const std::string& source,
    const std::string& target, const std::string& rep)
{
    return replace(source, axe::r_str(target), rep);
}
```

Formatted output

This example demonstrates how to create a parser for a format string to output data to **`ostream`** object. We would like to separate format string from data like the following:

```
std::cout << format("name = %1, value = %2") << "integer" << 123;
```

and see the output: `name = integer, value = 123`

We want the arguments in format string starting with % and followed by a natural number to be changed to actual values at the corresponding position. We also want to interpret `%%` in format string as literal `%`. Additionally, in case there are fewer arguments supplied than in format string we want to preserve all unmatched %N in the output. In case there are more arguments supplied than specified in the format string we want to append them to the output.

The rules for this parser can look like this:

```
auto percent = axe::r_lit("%%");
unsigned i;
auto arg = axe::r_lit('%') & axe::r_udecimal(i);
auto ignore = *(axe::r_any() - percent - arg);
auto format = *(ignore & *percent & *arg) & axe::r_end();
```

In this example we created what is called an island grammar, we wrote two rules to recognize %% and %N and another rule to ignore everything else.

In order to implement the desired functionality we also need to overload `operator<<` for format class and for any argument type used with format. The full code supporting narrow and wide strings can look like this:

```
template<class CharT>
class format_t
{
        typedef std::basic_string<CharT> StringT;
        std::basic_ostream<CharT>* os;
        std::vector<StringT> strings;
        unsigned index;

    template<class T>
    static StringT to_string(const T& t)
    {
                std::basic_ostringstream<CharT> tmp;
                tmp << t;
                return tmp.str();
    }
    format_t(const format_t&); // = delete;
public:
        format_t(format_t&& f)
          : os(f.os), strings(std::move(f.strings)), index(f.index)
        {
                f.os = nullptr;
                f.index = 0;
        }
        format_t(const StringT& format_string) : os(nullptr), index(0)
        {
                auto percent = axe::r_lit("%%");
                auto e_percent = axe::e_ref(
                            [this](StringT::const_iterator i1, StringT::const_iterator i2)
```

```
                        {
                                // adding a single % character
                                // VC10 bug - requires fully qualifed name
                                strings.push_back(format_t<CharT>::StringT(1, *i1));
                        });
                unsigned i;
                auto arg = axe::r_lit('%') & axe::r_udecimal(i);
                auto ignore = *(axe::r_any() - percent - arg);
                auto format = *(ignore >> axe::e_push_back(strings)
                        & *(percent >> e_percent)
                        & *(arg >> axe::e_push_back(strings)))
                        & axe::r_end()
               | axe::r_fail([](StringT::const_iterator i1, StringT::const_iterator i2)
                  {
                        // VC10 bug - requires fully qualifed name
                        throw format_t<CharT>::StringT(i1, i2);
                  });
                format(format_string.begin(), format_string.end());
        }
        ~format_t()
        {
         flush();
        }

      void flush()
      {
           if(os)
           {
                   std::for_each(strings.begin(), strings.end(),
                          [this](const StringT& str) { *os << str; });
           }
           os = nullptr;
      }

      template<class T>
       format_t<CharT>& operator<< (const T& t)
       {
                std::basic_ostringstream<CharT> ss;
                ss << (CharT)('%') << ++index;
                StringT index_str = ss.str();

                bool found = false; // at least one format found

                std::for_each(strings.begin(), strings.end(),
                        [&index_str, &t, &found](StringT& str)
                {
                        if(str == index_str)
                        {
                                str = format_t<CharT>::to_string(t);
                                found = true;
                        }
                });

                if(!found)
                { // no format found for this argument, include it verbatim
                        strings.push_back(to_string(t));
                }
                return *this;
       }
private:
       template<class C>
       friend format_t<C> operator<< (std::basic_ostream<C>& os, format_t<C>&& f);
};
```

```
    template<class C>
    format_t<C> operator<< (std::basic_ostream<C>& os, format_t<C>&& f)
    {
        f.os = &os;
        return std::move(f); // VC10 bug
    }

    format_t<char> format(std::string str)
    {
        return format_t<char>(std::move(str));
    }

    format_t<wchar_t> format(std::wstring str)
    {
        return format_t<wchar_t>(std::move(str));
    }

int main(int argc, char* argv[])
{
    // using lvalue
    auto fmt = format("Example 1\nCatalog #%1.\nBook Title: %2, Author: %3\n");
    fmt << 12345 << "Alice's Adventures in Wonderland" << "Charles Lutwidge Dodgson";
    std::cout << std::move(fmt); // operator<< is overloaded for rvalue references
    // using rvalue
    std::wcout << format(L"Example 2\nSymbol: %1, ROI: %2%%\n") << L"AAPL" << 380;
}
```

The output this program is this:

```
Example 1
Catalog #12345.
Book Title: Alice's Adventures in Wonderland, Author: Charles Lutwidge Dodgson
Example 2
Symbol: AAPL, ROI: 380%
```

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:

```
auto endl = r_char('\n'); // rule matching end-line
auto word = r_alnumstr(); // rule matching single word
auto number = r_numstr(); // rule matching single number
auto other = *(r_any() - endl - word - number); // 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:

```
number >> // extractor to count numbers

    e_ref([this](I i1, I i2)
    {
        ++nnumber_;
        numbers_.insert(token_t(i1, i2));
    })
```

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

The full code for this program is shown below.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <set>
#include <numeric>
#include <axe\axe.h>

using namespace axe;
typedef std::vector<unsigned char> token_t;

class wc
{
    // counts
    unsigned nline_, nword_, nnumber_, nother_;
    // 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:
    wc() : nline_(0), nword_(0), nnumber_(0), nother_(0)
    {
    }

    template<class I>
    wc& operator() (I begin, I end)
    {
        auto endl = r_char('\n'); // rule matching end-line
        auto word = r_alnumstr(); // rule matching single word
        auto number = r_numstr(); // rule matching single number
        auto other = *(r_any() - endl - word - number); // rule matching everything else

        auto file = // rule for parsing input file
            *(other >> // extractor to count other symbols
                e_ref([this](I i1, I i2)
                {
                    if(i1 != i2)
                    {
                        ++nother_;
                        other_.insert(token_t(i1, i2));
                    }
                })
            & *(endl >> e_ref([this](...) { ++nline_; }))
            & *(number >> // extractor to count numbers
                e_ref([this](I i1, I i2)
                {
                    ++nnumber_;
                    numbers_.insert(token_t(i1, i2));
                }))
            & *(word >> // extractor to count words
                e_ref([this](I i1, I i2)
                {
                    ++nword_;
                    words_.insert(token_t(i1, i2));
                })))
            & r_end();

        file(begin, end); // do the actual parsing

        return *this;
    }
};
```

```cpp
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(), 0u,
        [](unsigned 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_);

    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]
                << wc()(vec.begin(), vec.end()) << "\n";
        }
        else
        {
            std::cerr << "ERROR: failed to open file: " << argv[1] << "\n";
        }
    }
    else
    {
        std::cerr << "\nusage: wc <filename>\n";
    }

       return 0;
}
```

Doxygenated sources

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