DSL in C++ Template Metaprogram

Zoltán Porkoláb, Ábel Sinkovics, and István Siroki

Eötvös Loránd University, Faculty of Informatics
Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{gsd | abel | steve}@caesar.elte.hu

Abstract. We discuss a DSL integration technique for the C++ programming language. The solution is based on compile-time parsing of the DSL code. The parser generator is a C++ template metaprogram reimplementation of a runtime Haskell parser generator library. The full parsing phase is executed when the host program is compiled. The library uses only standard C++ language features, thus our solution is highly portable. As a demonstration of the power of this approach, we present a highly efficient and type-safe version of printf and the way it can be constructed using our library. Despite the well known syntactical difficulties of C++ template metaprograms, building embedded languages using our library leads to self-documenting C++ source code.

1 DSL integration

Two basic approaches exist when a DSL is about to be integrated into a host language. In one approach some external tool or framework can be used to identify, parse, syntactically and semantically check the domain specific language and generate the code integrated into the host language. As an alternative way one can use internal solutions, that do not require other tools than the infrastructure of the host language. Although many of the recent DSL integration approaches focus on the application of external tools [8][19][12][13][22], in the case of the C++ programming language there are vital examples for using the internal method. One of the primary candidates is template metaprogramming, in which one can define multi-staged compilation steps using only the C++ compiler.

2 Boost.Xpressive

The `Boost.Xpressive` is an advanced, object-oriented regular expression library for C++ [23]. Regular expressions are the most commonly used domain specific language among modern generic purpose programming languages. They are used for a very special purpose, text manipulation, and have a specific, usually implementation-independent syntax. Regular expressions are usually implemented as libraries. Classical regular expression libraries, like `stl::regex` from

the new C++ standard, are powerful and flexible. Patterns are represented as strings which can be specified at run-time. In this case a syntax error in the regular expression, such as unbalanced parenthesis, can be detected only at run-time. The Boost.Xpressive library, allows an alternative way: regular expressions can be defined using expression templates and thus they are checked at compilation time. Regular expressions from Boost.Xpressive can either be statically bound, hard-coded and syntax-checked by the compiler or dynamically bound and specified at run-time. These regular expressions can refer to each other recursively and match patterns in strings that ordinary regular expressions can not.

3 Improve Xpressive using Metaparse

`Xpressive` is a regular expression library from `Boost` which gives a portable internal solution, since it doesn't require any external tool. It follows the second approach we've mentioned in section 1.

Xpressive provides two ways for defining a regular expression (regex):

1. **Dynamic Regex** We can call the `sregex::compile` method in runtime to create a regular expression we specified as a string. This method doesn't provide compile time syntax check for the regex, although it retains the well-known syntax.
2. **Static Regex** We can build a regex from overloaded `C++` expressions, so called expression templates. This way we have compile time syntax checking, but we get high syntactical overhead. For example a regular expression in dynamic regex `"(\\w+) (\\w+)!"` looks like this in static regex:
   `(s1= +_w) >> ' ' >> (s2= +_w) >> '!'`

Our goal is to combine the best of these two approaches and give an interface where we can specify a regex as a string with compile time syntax checking. This way we can achieve seamless integration without syntactical overhead, thus we don't need to escape the domain of regular expressions when we want to use them in `C++`.

This can be done with the help of the `Metaparse` library by writing simpler parsers and combining them to build a more complex parser. To identify the needed parsers we will write a grammar for regular expressions. These individual parsers are metafunctions with a scope of parsing one element from the grammar. They can process different regex pieces like `[abc]` or `(xyz)`. We can combine them as described in section 6, thus we can parse `([abc]xyz)` too without writing a new parser for this expression. We will show a working library which can parse arbitrary complicated regular expressions and create the respective static `Xpressive` regular expressions for them using `C++` compiler only.

4 Grammar

To build a complex parser like this, we need to clearly describe how the possible regular expression pieces can be used together. The parser needs to know the

expected order of the pieces, and how they can be embedded within each other. The different expressions need to be categorised and ranked according to their precedence. A grammar can express these relations.

The `Xpressive` User's guide contains a table which shows line-by-line the `Perl` regular expression syntax and the corresponding static `Xpressive` expression. We can use this table as a starting point to determine the elements the parser needs to interpret and convert into static `Xpressive` objects. Note, the `POSIX` character classes like `[:alnum:]` should be written as `[[:alnum:]]`, if we want to comply with the Perl syntax.

It is challenging to identify the main grammatical elements which are based on the precedence of the expressions. We can try to draw syntax trees for concrete expressions to identify what kind of elements we would need in the grammar to build it up.

As we can see on figure 1, a regular expression (`reg_exp`) consists of `seq` elements. These sequences consist of arbitrary number of `unary_item` expressions which are `items` with their possible repetitions. It is necessary to differentiate `items` and `set_items`, because the latter has only those expressions of the grammar which can be used within square brackets. Arbitrary number of `set_item` can be joined with a `set` element which contains the closing `']'` character. A character group like `[abc]` can be called `group` in our grammar and its rule can be that it starts with `'['` and then it continues with a `set` or a range expression (`range_exp`).

We can also identify that anything can be put in a bracket expression, so we allow our head element to be used between `'('` and `')'`. With different examples we can gradually identify the use-cases of the elements, and deduce the suitable grammar elements. The final version of our grammar can be seen in appendix A.

5 Test case generation

The test cases play a very important role in template metaprogramming. Without them, after we've written the code and compiled it, we can fix the obvious syntax errors, but apart from that we haven't instantiated any template. To test every newly added element of the grammar we need a test case which makes use of that element. There are many combinations of regular expressions and we have a restriction on the format of the strings which we can add to our parsers. A `Perl` script can be written to generate these test cases. This way we can ensure that whatever we match with the built `Xpressive` object, it would match in `Perl` too. With the generated test cases, we can use the following template function for verification:

```
template <class Regexp>
bool search(const std::string& s, const std::vector<std::
    string>& m)
{
  const sregex re = apply_wrap1<regexp_parser, Regexp>::
      type::run();
```

```
  smatch w;

  const bool success = regex_search(s, w, re);
  if ( (success && m.size() == 0) || (!success && m.size
      () != 0) ) {
    return false;
  }

  if (w.size() != m.size()) { return false; }
  for (int i = 0; i < m.size(); ++i) {
    if (i >= w.size() || w[i].str() != m[i]) {
      return false;
    }
  }

  return true;
}
```

A reference implementation can be found in `xlxpressive`'s source files [21].

We can adapt the format of the `libs/xpressive/test/regress.txt` file, which contains `Xpressive`'s regression tests. Here is an example to show the structure of a test case:

```
[test37]
str=2001  foobar
pat=[1-9][0-9]+\\s*[abfor]+$
flg=
br0=2001  foobar
[end]
```

Each test case starts with its name in square brackets, then it has key-value pairs and is closed with the `[end]` tag. The key-value pairs have the below meaning:

- `str`
  The input string.
- `pat`
  The regular expression we're testing.
- `flg`
  The behaviour of the `Xpressive` regular expression algorithm can be modified with a couple of flags e.g. 'i' for case insensitive search.
- `br0-n`
  `br0` contains the whole matched string. Every other one comes from the back-referencing of bracket expressions. If a test case doesn't contain any `br` element it means that we expect that the test case will fail.

6 Making parsers based on the grammar

6.1 A basic parser

After we have the grammar, we can commence building up the parsers with the help of newer and newer test cases. Let's start with the less complicated ones, which are at the bottom of our syntax tree. The very first parser we'll build is for the caret character ('^'), which has a special meaning in regular expressions. We can put it as the first character of a sequence, which means that the following regular expression elements should match a string from its beginning. The code of this `bos` (beginning of sequence) parser is as follows:

```
struct build_bos
{
  typedef build_bos type;
  static xpressive::sregex run() {
    return xpressive::bos;
  }
};
typedef metaparse::always<
  metaparse::lit_c<'^'>, xlxpressive::build_bos
> bos;
```

The parser is built using a `typedef`. We can use `lit_c<'^'>` to identify a caret character. The `always` parser can be used to replace this character with our `build_bos` metafunction class. This class has a `type`, as every metafunction class should, and a static `run` method. This method can be used to return the desired static `Xpressive` object, which is in our case a `boost::xpressive::bos` object. The `eos` and `any` elements have been created the same way.

So far so good, but how can we use this as a regular expression? We have the '^' character identified by `lit_c`. Our `bos` parser returns the `build_bos` metafunction class, because we use the `always` parser. To get the static `Xpressive` object we can call the `run` method through `type`, since it is `static`.

We have the parser we'd like to test, so we just need to give it the input string from a test case. To do this easily `Metaparse` provides us a `build_parser` metafunction. What it does is exactly what we want: it wraps our parser with a metafunction class which expects an input string, gives it to our parser and returns its result. If our parser fails, a compilation error will be generated. We use the `entire_input` parser around our own parser to ensure that we process the entire input string. So this is how we can use our first simple parser:

```
typedef metaparse::build_parser<
  metaparse::entire_input<xlxpressive::bos>
> regexp_parser;
```

This can be tested with a simple test case like this:

```
[test1]
str=
pat=^
flg=
br0=
[end]
```

The final step is to create an `sregex` object. This can be done by applying our `regexp_parser` on a `boost::mpl::string`, or we can use the `MPLLIBS_STRING` macro, if we can leverage the C++11 standard. For our first simple example here's how we can do this:

```
const sregex re = mpl::apply_wrap1<regexp_parser,
    MPLLIBS_STRING("^")>::type::run();
```

To make the whole thing even more usable, we can wrap it with a macro like `REGEX`. This way we can simply create our `Xpressive` regular expression object (`sregex`) using this macro, and then we can use it the normal way:

```
#define REGEX(s) (mpl::apply_wrap1<regexp_parser,
    MPLLIBS_STRING(s)>::type::run())
const sregex re = REGEX("^");
```

As we build our solution further we can add new test cases and change the actual parser behind `regexp_parser` to the actual top element we have implemented so far from the grammar. This way we can test the added parser and ensure that we haven't broken anything.

6.2 How to build more complex parsers

After we have these basic elements we can go up a level on the syntax tree to see how they fit into the upper element, `item` in this case. It looks like this in our grammar:

```
item ::= set_item|bos|eos|any|bracket_exp|group
```

We just simply list the acceptable elements. It is important that these elements are in the order of precedence i.e. if the first clause matches, the others aren't evaluated at all. We can do the same using the `one_of` parser:

```
typedef metaparse::one_of<
  xlxpressive::set_item,
  xlxpressive::bos,
  xlxpressive::eos,
  xlxpressive::any,
  xlxpressive::bracket_exp,
  xlxpressive::group
>
item;
```

We used `typedef` again to create a new element of the grammar. This is a common technique to define our entities in template metaprogramming. As you can see we've listed all the elements we need for a complete item. We will examine some of them in details later on. The very similar `set_item` can be built this way too.

The `item` element with an optional `repetition` behind it construct the `unary_item`, our next target.

```
unary_item ::= item (('*'|'+'|'?'|repeat) '?'?)?
```

This complicated thing after `item` means that it can be followed by a '*', '+', '?' character or a 'repeat{n, m}' construct. This is the `repetition` of `item`. The '?' character is optional, just like this whole `repetition` thing itself. Let's see what we want to identify and what we want that to be transformed into. On the left side we have the regular expressions in their common format, how we use them in `Perl` and on the right side the respective static `Xpressive` form can be seen.

```
a       -> a
a*      -> *a
a+      -> +a
a?      -> !a
a{n,m}  -> repeat<n,m>(a)
a*?     -> -*a
a+?     -> -+a
a??     -> -!a
a{n,m}? -> -repeat<n,m>(a)
```

The main problem here is this: we can parse an `item` and a `repetition` separately in a `sequence`, but how should we give the result of `item` to the `repetition`? Let me show how we can identify these elements one-by-one, and after that how we can solve this issue. Using a top-to-bottom approach let's write what we want first, break the problem into smaller parts and solve these later on:

```
typedef metaparse::transform<
  metaparse::sequence<xlxpressive::item::type,
      xlxpressive::repetition>,
  xlxpressive::build_unary_item
>
unary_item;
```

With `sequence` we can specify an order between the sub-parsers. We accept a `unary_item` only, if it consists of an `item` and then a `repetition`. The `transform` parser will be used many times further on. It is very useful, because it calls the second template parameter with the result of the parser in its first template parameter, which is a `sequence` in our case. The second parameter is a metafunction class responsible for the transformation. We need a metafunction class

instead of a simple metafunction, because we need a type here. Metafunction classes are complete types, so we can pass them as metafunction arguments. They are wrappers around their publicly-accessible `apply` metafunction.

We have seen how the `item` parser can be built up. Let's create the `repetition` parser, so we're focusing on this part of the grammar now:

```
(('*'|'+'|'?'|repeat) '?'?)?
repeat ::= '{' (number ',' number|',' number| number ',') '}'
```

The pipe separated list is similar to what we have seen at the `item` element, so we can use the `lit_c` and `one_of` parser combinators again. To specify the order between this part and the question mark character, a `sequence` can be used. The numbers can be identified with the `digit_val` parser, if we want the `int` value to be returned. If we want to mark whether we've seen an optional element, like '?', or not, we can use the `return_` parser. It simply returns what is its argument without parsing anything. We can use it with `char` and `int` values in our case.

We have the following structure to be parsed: `[*+?{](\d,\d})?'?'?`. To handle the optional parts we use the `one_of` parser this way:

```
typedef metaparse::one_of<
  metaparse::digit_val,
  metaparse::return_< mpl::int_<-1> >
>
maybe_digit;
```

It can accept a digit or return -1 otherwise. If we create these parsers, we can make the code more expressive and readable. We should do the same thing for the comma:

```
typedef metaparse::one_of<
  metaparse::lit_c<',',>,
  metaparse::return_< mpl::char_<'x'> >
>
maybe_comma;
```

We parse a `','` character or return an `'x'`. Of course, we can choose any other character instead of `'x'` to express that we've not matched the expected one. We can define `maybe_close_curly_bracket` and `maybe_questionmark` the same way. We should have a return value in case we don't have this repetition part at all. For this we can create a `dont_repeat` type, which can be defined using a `boost::mpl::vector` type sequence the following way:

```
typedef metaparse::return_<
  mpl::vector<
    mpl::char_<'x'>,
    mpl::int_<-1>,
```

```
    mpl::char_<'x'>,
    mpl::int_<-1>,
    mpl::char_<'x'>,
    mpl::char_<'x'>
  >
>
dont_repeat;
```

With these structures we can define `repetition` in a self-documenting manner like this:

```
typedef metaparse::transform <
  metaparse::one_of<
    metaparse::sequence<
      metaparse::one_of<
        metaparse::lit_c<'*'>,
        metaparse::lit_c<'+'>,
        metaparse::lit_c<'?'>,
        metaparse::lit_c<'{'>
      >,
      maybe_digit,
      maybe_comma,
      maybe_digit,
      maybe_close_curly_bracket,
      maybe_questionmark
    >,
    dont_repeat
  >,
  xlxpressive::eval_repetition
>
repetition;
```

We've wrapped the whole thing with the `transform` parser combinator again, because we have character and numeric values only and we need to evaluate them and return the corresponding repetition somehow.

With `eval_repetition` we want to give back a "metaprogramming data", which shows what kind of repetition the user has specified. These are the results of our repetition parsing and can be declared as `struct`s e.g.:

```
struct no_repeat { typedef no_repeat type; };   // when
    we haven't seen repetition
struct any_repeat { typedef any_repeat type; }; // when
    '*' has been identified
struct any_may_repeat { typedef any_may_repeat type; };
    // when both '*' and a following '?' have been
    identified
// etc...
```

Let's just concentrate on `any_repeat`. How can `eval_repetition` return it, while it has the `char` and `int` values only? We can use the template specialization here, because this way like with pattern matching in functional programming, we can uniquely choose and return the needed result. `eval_repetition` is a metafunction class. To do the pattern matching with specialization we can declare a `struct eval_repetition_impl` which has 6 template parameters. We can pass these parameters to the implementation by processing the sequence we've got from the `transform` parser. The `boost::mpl::at_c` metafunction can be used here, since we know the length of the received sequence and we can pass the index as a constant value:

```
struct eval_repetition
{
  template <class Seq>
  struct apply :
    eval_repetition_impl<
      mpl::at_c<Seq, 0>::type::value,
      mpl::at_c<Seq, 1>::type::value,
      mpl::at_c<Seq, 2>::type::value,
      mpl::at_c<Seq, 3>::type::value,
      mpl::at_c<Seq, 4>::type::value,
      mpl::at_c<Seq, 5>::type::value
    > {};
};
```

The implementation for `any_repeat` can be seen below. The listed constants in the template parameters of `eval_repetition_impl` show that we've specialized for the case when the user gave a `'*'` after the `item`. The inheritance can be used here as a technique to return our prepared `any_repeat` type.

```
template <char A, int N, char B, int M, char C, char D>
struct eval_repetition_impl;

template <> struct eval_repetition_impl<'*', -1, 'x', -1,
    'x', 'x'> : any_repeat {};
```

So this is how we can recognise the `item` and `repetition` elements. However, to combine them in `build_unary_item` we need to slightly modify the possible results of repetition:

```
//a* -> *a
struct any_repeat
{
  typedef any_repeat type;
  static xpressive::sregex run(xpressive::sregex base) {
    return *base;
  }
};
```

We added an `sregex` parameter to the `run` method, because the `repetition` needs to use the `item` it stands after. In case of `any_repeat` we return the `item`, which is called `base` here, with a `'*'` in front of it.

```
typedef metaparse::transform<
  metaparse::sequence<xlxpressive::item::type,
      xlxpressive::repetition>,
  xlxpressive::build_unary_item
>
unary_item;
```

The `transform` parser of `unary_item` passes the `item` result and the `repetition` result to `build_unary_item`. So, these wrapped by the `Seq` type sequence become the template parameter of the `apply` metafunction of `build_unary_item`. The first element of `Seq` is the result of the `item` parser, which is a sub-parser's builder metafunction class e.g. `build_bos`. The second element is the `repetition` data, which has a `run` method now. This method is the key to solve our previously described problem, namely how we can pass the previously used `item` parser as a parameter of `repetition`. The result of parsing a `repetition`, like `any_repeat`, has a `run` method now, after the above modification, which has an `sregex` argument.

So what the `build_unary_item` metafunction class should do is to call the `repetition`'s run method with the identified `sregex` object from `item`.

```
struct build_unary_item
{
  template <class Seq>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run()
    {
      return mpl::back<Seq>::type::run(mpl::front<Seq>::
          type::run());
    }
  };
};
```

It can call `item`'s static `run` method through its `type`, so this way within the `repetition`'s `run` method we'll have the actual, preceding `sregex` extracted from the `item` parser.

The idea behind building these separate parsers is that we can call the `run` method of our top element, which calls the lower level element's `run` method and so forth. Through these `run` method call-chains, we can build up more complex regular expressions from the individual parsers.

Let's see how this works through a simple example. Suppose we want to use the `".*"` regular expression. We can use the same macro we introduced earlier: `REGEX(".*")`. As you may remember this applies the `".*"` `boost::mpl::string` on the top level parser of our library, which can be the `unary_item` parser now.

Here you can see the call-chain of the `run` methods of this example:

```
unary_item::type::run()
-> build_unary_item::apply<Seq>::run()
-> return repetition::type::run(item::type::run());
-> return any_repeat::type::run(build_any::type::run());
-> return any_repeat::type::run(~xpressive::_n);
-> return *~xpressive::_n;
```

Just one thing left here I still owe you, the repeat elements. These differ from the others, since they need two template parameters beside the `item` (`'a'`) argument:

```
a{n,m} -> repeat<n,m>(a)
a{n,m}? -> -repeat<n,m>(a)
```

This is why `eval_repetition_impl` has 6 template parameters instead of just 2. We can pass N and M to the corresponding result type, (`range_repeat` and `may_range_repeat`), using partial template specialization:

```
template <char A, int N, char B, int M, char C, char D>
struct eval_repetition_impl;

template <int N, int M> struct eval_repetition_impl<'{',
    N, ',', M, '}', '?'> : may_range_repeat<N, M> {};
```

With this solution, the `may_range_repeat` result type can directly pass its template parameters to a `boost::xpressive::repeat` instantiation:

```
//a{n, m}? -> -repeat<n, m>(a)
template <int n, int m>
struct may_range_repeat
{
  typedef may_range_repeat type;
  static xpressive::sregex run(xpressive::sregex base)
  {
    return -xpressive::repeat<n, m>(base);
  }
};
```

We can write `range_repeat` the same way.

`qexp` has been done in a similar way, but there the `always` parser has been used, because it has simpler sequences like "i:". Another similar parser is `bschar` which processes the non-printable characters we write starting with a backslash

like "\ n". We need the metaprogramming result expressions there too, but we don't need to write a complex evaluator like `eval_repetition` and a builder like `build_unary_item`.

Let's summarize what parts we have implemented so far from the grammar:

```
unary_item ::= item (('*'|'+'|'?'|repeat) '?'?)?
repeat ::= '{' (number ',' number|',' number| number ',') '}'
item ::= bos|eos|any
bos ::= '^'
eos ::= '$'
any ::= '.'
```

6.3 The top of the grammar

We continue to climb up the syntax tree, where the only thing left is the `seq` element, before we reach the top element `reg_exp`:

```
seq ::= unary_item*
```

This is the first expression which is built up from an arbitrary number of components. Metaparse gives us an `any` parser combinator which could be used in this case, but we need to build up something from the individually processed `unary_items`. The `foldl` parser combinator suits our needs better in this case. If you're familiar with functional programming, you might already have an idea what this parser could do. It tries to apply repeatedly its first template parameter, a parser. It uses the second parameter as a starting point and executes its third argument, which is a metafunction class by giving the existing result and the next element to that as template parameters. So, we can build something starting from the second argument and building it using the third argument, the metafunction which always has the last parsed element from the first argument, which is our repeated parser.

The parser what we want to apply 0 or more times is the `unary_item`. This is the first parameter, the repeatedly applied parser, which consumes the input string. The "neutral element" of our building process is `empty_seq`. It should give back a kind of regular expression which matches everything, so it doesn't have any effect on the rest of our built expression. `Xpressive` doesn't really support something like this, but we can use the below solution. It introduces a problem which I'm going to explain in details a bit later.

```
struct empty_seq
{
  typedef empty_seq type;
  static xpressive::sregex run()
  {
    return xpressive::as_xpr("");
  }
};
```

The `foldl` parser will take this element as its starting state. This means that its "builder" metafunction will append all the parsed elements to this, so that they will look like this: `as_xpr("")>>as_xpr('a')>>as_xpr('b')>>...`. We use `as_xpr` to create a static `Xpressive` object from the empty string. Otherwise, it would be a normal `string` and we would use the usual right shift operator with it, instead of the overloaded one. When we have an empty input string, our result will be this `empty_seq`.

Our third parameter is the `build_seq` metafunction class:

```
struct build_seq
{
  template <class Next, class State>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run()
    {
      xpressive::sregex s = State::type::run();
      return s >> Next::type::run();
    }
  };
};
```

It gets two template parameters from `foldl`:

- `Next`
  The next parsed element, which is the `apply` metafunction of `build_unary_item`, so we can call its `run` method.
- `State`
  The sequence we have built so far. At the beginning, it is the `empty_seq`. That's why each sequence starts with `as_xpr("")`.

The built simple parser always succeeds, even if `unary_item` rejects the input string the very first time:

```
typedef metaparse::foldl<
  xlxpressive::unary_item,
  xlxpressive::empty_seq,
  xlxpressive::build_seq
>
seq;
```

It iterates through our input string with `foldl` and processes it with the `unary_item` parser. After each successful parsing, it calls `build_seq` to assemble the separate `unary_items` with the "¿¿" operator. We need to use this operator for sequences, because in static `Xpressive` we must use valid, overloaded `C++` language constructs to build up our regular expression, that's the main idea behind it, hence

we cannot just put things like characters next to each other i.e. we need to convert our simple `ab` regular expression into `as_xpr(a)>>b`. `as_xpr` is needed to force the compiler to call the overloaded "¿¿" operator and not the one for `char` types.

By using `foldl` we've got a short and elegant solution, but with the `any` parser we'd need to solve where and how to aggregate the sequence, which is simply built by `foldl` in this case.

We can use the same technique we've seen at `seq` for our head element, `reg_exp`.

```
reg_exp ::= seq ('|' seq)*
```

I won't go into too much details here, but a few things worth mentioning. For example, we can use `foldlp` now, because we have at least one element in the arbitrary long sequence. `foldlp` does exactly what we need here: it executes its second argument first, a parser, and if it succeeds, it applies its first argument repeatedly, which is a parser too, and calls `build_reg_exp` to assemble the expression:

```
struct reg_exp : metaparse::transform<
  metaparse::foldlp<
    metaparse::last_of< metaparse::lit_c<'|'>,
        xlxpressive::seq >,
    xlxpressive::seq,
    xlxpressive::build_reg_exp
  >,
  xlxpressive::eval_reg_exp
>
{};
```

6.4 Bracket expressions

You might have observed that we've declared our top element as a `struct` and not used `typedef`. This is because we need to forward declare `reg_exp`, so that we can use it in the bracket expressions:

```
bracket_exp ::= '(' (reg_exp|qexp) ')'
qexp ::= '?' (no_back_ref|...) reg_exp
```

These expressions can have two forms: `(...)` and `(?...)`. The difference between them is that the prior is the "simple" bracket expression which saves a back-reference for its wrapped regular expression, while the latter with the starting '?' is for a couple special bracket wrapped constructs like `(?i...)` which does case-insensitive matching within this bracket expression.

Here are a couple of these question mark prefixed expressions. On the right side of the arrow we've listed what we want to generate for them:

```
(?i:regex) -> icase(regex)
(?>regex)  -> keep(regex)
(?=regex)  -> before(regex)
(?!regex)  -> ~before(regex)
```

To handle these two types, we transform them with separate builder metafunction classes and choose between them with `one_of`. To recognize a bracket expression we need to identify a `qexp` or `reg_exp` between opening and closing brackets. Metaparse gives us a parser called `middle_of` which can be used for sequences with 3 elements to parse all of them, but return the result of the second only.

```
typedef metaparse::transform<
  metaparse::middle_of<
    metaparse::lit_c<'('>,
    metaparse::one_of<
      metaparse::transform< xlxpressive::qexp,
          xlxpressive::build_qexp_based_bracket_exp >,
      metaparse::transform< xlxpressive::reg_exp,
          xlxpressive::build_reg_exp_based_bracket_exp >
    >,
    metaparse::lit_c<')'>
  >,
  xlxpressive::eval_bracket_exp
>
bracket_exp;
```

The `build_qexp_based_bracket_exp` and `eval_bracket_exp` metafunctions simply call the passed type's static `run` method.

However, the `build_reg_exp_based_bracket_exp` metafunction is a bit more interesting one. It does exactly what it says on the tin, but to build a simple bracket expression in static `Xpressive`, where we should give back back-references after each bracketed expression, we need to express this intention explicitly. We cannot simply wrap our `reg_exp` in brackets and return it, because brackets cannot be overloaded this way. To solve this, `Xpressive` uses the `s1...s9` so called sub-match placeholders. So, for example to write a regular expression like `(a)` in the static `Xpressive` world, we should do this: `(s1=a)`.

Since we're translating "normal" string-based regular expressions into static `Xpressive`, we don't have the actual number of the next sub-match placeholder which we should return. So, we need to count it ourselves.

The easiest way of doing this, if we introduce a global variable; let's call it for example `bracket_counter`. We can increase its value every time when the `run` method is called, before we'd return the bracket-wrapped expression. However, it's not a good idea to use global variables in a library, so we should come up with a solution where our counter has local scope.

We could use the counter as a local variable in the `run` method, if it's passed as a parameter. If we take this idea as a starting-point, we'll find that this can

actually solve our problem, if we pass the counter through the `run` method call-chain as a reference. The very first `run` method which we call when we start the evaluation of our regular expression is the one in `eval_reg_exp`. We introduce this counter here as a local variable of the parameterless `run` method which calls the overloaded one. This way we ensure that the counter will be initialized at the start of the evaluation and that `reg_exp` can pass an existing value forward, if we use it in a bracket expression:

```
struct eval_reg_exp
{
  template <class Re>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run()
    {
      int bracket_counter = 0;

      return Re::type::run(bracket_counter);
    }
    static xpressive::sregex run(int &bracket_counter)
    {
      return Re::type::run(bracket_counter);
    }
  };
};
```

To make it work, we need to modify all the previously created parsers, so that the`run` method of their build and/or evaluate metafunctions will get the `bracket_counter` as a reference. The only thing they should do with that is passing it forward to the underlying parser(s), if they have any at all.

For example here's how `build_seq` can be modified:

```
struct build_seq
{
  template <class Next, class State>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run(int &bracket_counter)
    {
      xpressive::sregex s=State::type::run(
          bracket_counter);
      return s >> Next::type::run(bracket_counter);
    }
  };
};
```

So every parser will just get and pass the counter without modifying it, except `build_reg_exp_based_bracket_exp`. We can simply increase it here, evaluate the `reg_exp` and then choose the right sub-match placeholder with a `switch-case` construct:

```
struct build_reg_exp_based_bracket_exp
{
  template <class E>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run(int &bracket_counter)
    {
      ++bracket_counter;
      xpressive::sregex a = E::type::run(bracket_counter)
          ;
      xpressive::sregex ret;
      switch(bracket_counter) {
        case 1: ret = (xpressive::s1= a); break;
        case 2: ret = (xpressive::s2= a); break;
        // case 3 .. 9 are similar
      }
      return ret;
    }
  };
};
```

As I mentioned earlier, our solution for `seq` -where we've used the `as_xpr("")` as a neutral element in `empty_seq`- introduced an issue. This bracket expression is the parser where we can meet with this. Let me explain it through an example: let's assume that we want to use the "(foo)" regular expression. We can do it with our new library this way: `const sregex = REGEX("(foo)")`. It looks OK, but when we try it out with our `search.hpp` test utility, which is used to process each test cases we generate with `gen_test.pl`, we get an interesting result:

```
success: 1
matching expected success: 1
size_check: 0 | 1 ~ 2
'foo' VS 'foo'
sub_match: 0
RESULT: 0
```

It shows that our regular expression matched, and that it has the expected behaviour. However, the `sub_match` check failed, and this part `[0 | 1 ~ 2]` shows that our solution has one less sub-match, than the expected. To find the root-cause behind this, the best way is to debug the library, level-by-level to see where it goes wrong. If we do this in our case we can find that at the level of

the `seq` parser our library doesn't create what we would manually. This isn't a real problem however, but a speciality, caused by the way how we build up our regular expressions. We should be aware of this when we use the library.

Let's see what we generate for "(foo)" to understand this:

```
sregex(as_xpr("")) >> sregex(s1= as_xpr('f') >>
                               as_xpr('o') >>
                               as_xpr('o'))
```

We have an "empty" regular expression at the front of what the user has specified. Why this makes a difference can be found in the documentation of `Xpressive` [24]. Each `sregex` need to have its own back-tracking scope. Regarding our solution it means that we cannot iterate through the `smatch` object after a `regex_search`, if we want to see what are the sub-matches. Instead of this, we should call the `nested_results` method on the `smatch` object which we've used for the search.

Here's an example how we can do it.

```
// If sub_match and size_check failed, try to match with
    nested results
if (!sub_match && !size_check && w.nested_results().size
    () > 0) {
    sub_match = true;
    nested_result_analyzer nra(m, sub_match);
    nra = std::for_each(
      w.nested_results().begin(),
      w.nested_results().end(),
      nra);

    size_check = nra.size() == m.size();
  }
}
```

A reference implementation can be found in `xlxpressive`'s source files.

6.5 Character groups

In regular expressions we can use character groups like `[a-z]` or `[abc]`, if we want to exactly list which characters we accept in a character place. However, we can't list any kind of grammar elements between the square brackets e.g. we cannot use a bracket expression there. That's why in the grammar we've allowed `char_group`, `range_exp` and `set_item` only. To make it clear let me give you an example of each, so that we can see the difference between them. On the left side we listed the grammar elements, while in the middle we can see an example for these elements as we would write them in a `Perl` regular expression. The last column shows the static `Xpressive` form of the examples from the second column.

```
char_group      [[:alnum:]]     alnum
range_exp       [0-9]           range('0','9')
set_item        [aB%\w7]        set['a' | 'B' | _w | '7']
```

As you can see, we differentiate them, because they correspond to different kind of `Xpressive` objects. They can be separated by their structure too. Each of them starts with a '[' character, so we first try to parse that. If we succeed, then according to our grammar, we have two choices: `char_group` or `set`:

```
group       ::= '[' (char_group|'^'? set)
set         ::= (range_exp|set_item)+ ']'
range_exp   ::= number set_num|letter set_abc|set_item
set_num     ::= '-' num_range|set_item
num_range   ::= number
set_abc     ::= '-' abc_range|set_item
abc_range   ::= letter
```

The `char_group` parser is the easier one. We just need to specify the sequences of the expected letters of the character group names and choose between them with a `one_of` and `always` construct. Here's `alnum` for example. All the others have been done the same way:

```
typedef metaparse::transform<
  metaparse::sequence<
    metaparse::lit_c<'['>,
    metaparse::lit_c<':'>,
    metaparse::one_of<
      keyword<MPLLIBS_STRING("alnum"), xlxpressive::alnum
          >,
      ...
    >,
    metaparse::lit_c<':'>,
    metaparse::lit_c<']'>,
    metaparse::lit_c<']'>
  >,
  xlxpressive::eval_char_group
>
char_group;
```

As you can see we use a new parser called `keyword` here. A `boost::mpl::string` can be parsed with that, which we specify as its first argument. It returns the optional second argument, if it succeeds. We use `alnum` as this second argument, which is the returned result of our parsing. It looks very much the same as the ones for `unary_item` we have seen previously, but we don't need the extra parameter for the `run` method.

```
//[:alnum:] -> alnum
struct alnum {
  typedef alnum type;
  static xpressive::sregex run() {
    return xpressive::alnum;
  }
};
```

The `run` method of the `eval_char_group` metafunction returns whatever the `::type::run()` method of the 3rd element of the passed `Seq` sequence returns, which is the result of the `one_of` parser. In our case, `::type::run()` is called on the `alnum` result, so `eval_char_group` returns `boost::xpressive::alnum`.

```
struct eval_char_group
{
  template <class Seq>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run(int &bracket_counter)
    {
      return mpl::at_c<Seq, 2>::type::run();
    }
  };
};
```

We get the `bracket_counter` argument in this `run` method, as we introduced them in section 6.4, but we don't pass it forward. This is because the things like `alnum` are leaf elements in our syntax tree, so they return the needed objects and don't need to do anything with a passed extra argument.

So far so good, let's continue with `set`. This grammar element introduces the most complicated problem we cover in this thesis and needs an advanced technique to solve it. Let me describe the problem with `set` first and then show you what kind of solution we can find in a case like this.

We've named this element of the grammar after the construct of `Xpressive` we want to generate. We can define a `set` in multiple ways in static `Xpressive`:

```
(set= 'a','b','c')
set[ range('0','9') |(set= 'a','b','c') ]
set[ range('0','9') | 'a' | 'b' | 'c' ]
```

The last form suits our needs best, because it can hold characters and `range_exp` elements too. The problem here is that we cannot put the objects of type `boost::xpressive::sregex` returned by our parsers directly into this object. The first thing we might think that, OK, we can try to find out what the exact type of e.g. `range` and return that in its `run` method. Unfortunately, we

can't, because it is generated with the help of `Proto` [10]. This means that we should return a type we cannot know, as it's known by the compiler only, when it generates it.

In `C++11` we could use the redefined `auto` keyword [9] to let the constructor automatically deduce the return type from the expression we return. However, we'd like to build a C++98 compliant library as much as possible to foster wider usability.

There is a technique we can use to solve this problem without leaving the frames of `C++98`. It is called Continuation-Passing Style (CPS) [11, 15]. When we write functions in CPS we give back the result of the function in its extra "continuation" argument instead of in the return statement. It's like we turn the expression "inside-out", as the innermost part will be evaluated first. This technique is useful in our case, because this extra argument of the function can have a template type, so that we can let the compiler maintain the type of the intermediate temporary elements. These are those types, like the type of the `range` element, which we couldn't know in advance of the compilation, because these are generated by the compiler.

We can build up our `set` expression by passing the already built part as an argument and always just appending that to the newly parsed object. To make it work we always need to have the previous element, so that we can call its `run` method with our newly constructed part. Our current parsers don't have this kind of functionality, so we will amend them. We need to modify only a subset of our parsers however, since not every kind of regular expression is grammatical within `set`. Based on our grammar, we need to write the `range_exp` parser this way and modify `set_item` (its sub-parsers actually):

```
set ::= (range_exp|set_item)+ ']'
range_exp ::= number set_num|letter set_abc|set_item
set_num     ::= '-' num_range|set_item
num_range   ::= number
set_abc     ::= '-' abc_range|set_item
abc_range   ::= letter
```

Each parser which can occur in `set` will have a common extra "interface" called `add_set_item` and a new `run` method, which is overloaded with the `(T after)` argument. These methods make us able to iterate through the parsed elements and call the previous element's overloaded `run` method.

```
template <class T>
static xpressive::sregex run(T after) {
  return after;
}

template <class Before>
struct add_set_item {
  typedef add_set_item type;
```

```
  template <class T>
  static xpressive::sregex run(T after) {
    return Before::type::run( [current_element] | after )
        ;
  }
};
```

`add_set_item` is a metafunction getting the previous parsed element as its template parameter. In the `after` parameter its `run` method gets what has been constructed after our current element. As we have what's before `[current_element]` and what's after it, we can put it right in the middle of them with the `run` method call on `Before`. We'll put these extra methods in the sub-parsers of `set_item`: `bschar`, `number`, `letter` and `non_alphabet`. `range_exp` won't have an argument-less `run` method, as it cannot be used outside `set`.

Let's see how the original `letter` parser looks like and add the modifications one-by-one:

```
struct build_letter
{
  template <class ch> struct apply
  {
    typedef apply type;
    static xpressive::sregex run(int &bracket_counter) {
      return xpressive::as_xpr( char_value() );
    }
    static char char_value() {
      return ch::value;
    }
  };
};
typedef metaparse::transform< metaparse::letter ,
    xlxpressive::build_letter
> letter;
```

We simply return the `char` value wrapped as a static `Xpressive` object. We use a separate method called `char_value()` here, because we'll need it later on for the `range_exp` parser. First we add the overloaded `run` method after our original one.

```
    template <class T>
    static xpressive::sregex run(T after)
    {
      return after;
    }
```

After we added this, we should add the `add_set_item` structure, which can call it through the `Before` parameter.

```
    template <class Before>
    struct add_set_item {
      typedef add_set_item type;

      template <class T>
      static xpressive::sregex run(T after)
      {
        return Before::type::run( char_value() | after );
      }
    };
```

We've used the overloaded `run` method, and the `add_set_item`, just like how we've sketched. We've replaced the placeholder `[current_element]` with method call `char_value()`, thus we return what the normal `run` method returns, but without wrapping it as an `Xpressive` object. This is how we can add the actually parsed element in the form we want. In the case of `letter` we simply pass the `char` value.

Let's build up `set`. With CPS this is a two way process:

1. We parse the elements left-to-right and creating types which have the previous state and need an `after` parameter.
2. After we've parsed the last `set_item` or `range_exp` we have a temporary construct, which can be evaluated backwards by calling its `run` method with a neutral start value. This can be `range('4', '2')` as it's an empty range.

For the iteration we can use `foldl1`, because we don't accept an empty set. This part of the parser is similar to the `seq` and `reg_exp` parsers in section 6.3.

```
typedef metaparse::transform<
  metaparse::first_of<
    metaparse::transform<
      metaparse::foldl1<
        metaparse::one_of< xlxpressive::range_exp,
            xlxpressive::set_item >,
        xlxpressive::empty_set,
        xlxpressive::build_set
      >,
      xlxpressive::start_building_set
    >,
    metaparse::lit_c<']'>
  >,
  xlxpressive::eval_set
>
set;
```

We've seen `last_of` and `middle_of` previously, `first_of` is the one which returns the result of the first parser after accepting a sequence. This way we can get rid

of the ']' easily. Let's see what the first phase of parsing does. With `foldl1` we parse with one of the `range_exp` and `set_item` parsers. We'll specify `range_exp` later on, so let's just concentrate on `set_item` now, moreover we can just use `letter` instead of that.

We start building up the `set` with the `empty_set` metafunction class. On the backward way this will construct our final object we return, but in this stage this just has a metafunction which can be called with an `after` parameter.

```
struct empty_set
{
  typedef empty_set type;

  template <class T>
  static xpressive::sregex run(T after)
  {
    return xpressive::set[ after ];
  }
};
```

The `build_set` metafunction class uses the `add_set_item` interface of the actual `Next` element and passes the current state (`State`) as its template parameter. The `run` method of `add_set_item` is called with the `after` parameter. It's again just a metafunction, like `empty_set`, which has to be evaluated with an `after` parameter. It's like we're unrolling these generated metafunctions as a "wick" after us and when we reach the last element, we'll light it, that is, we work it up backward.

```
struct build_set
{
  template <class Next, class State>
  struct apply
  {
    typedef apply type;
    template <class T>
    static xpressive::sregex run(T after)
    {
      return Next::template add_set_item< State >::type::
          run(after);
    }
  };
};
```

The "lighter" for this string of metafunctions, to close this metaphor, is the `start_building_set` metafunction class. The `transform` parser around `foldl1` calls it with the result of our folding, hence we can open it up. We start the second phase of the parsing with the empty range (e.g. `range(4,2)`). This is

where the CPS starts working and the method calls build up our expression in the after argument.

```
struct start_building_set
{
  template <class RealSetBuilder>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run()
    {
      return RealSetBuilder::type::run(xpressive::range('
          4', '2'));
    }
  };
};
```

The eval_set metafunction class is a simple evaluator, which we've seen a couple before. It can call the run method of start_building_set, because that doesn't need any parameter.

```
struct eval_set
{
  template <class Set>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run(int &bracket_counter)
    {
      return Set::type::run();
    }
  };
};
```

The last element of the second phase is the empty_set which wraps the built object and returns our final expression:
`return boost::xpressive::set[ after ];`

To see how the backward way works, let's go through the evaluation of a simple character group: [abc]

On figure 2 you can see how start_building_set commences the backward build process by calling the run method of the last build_set with the empty range. In each build_set the State stores the previously processed build_set from the first phase, the Next parameter is what the current function can generate, and in the after parameter you can see how we build up the full expression. With the `return Next::template add_set_item< State >::type::run(after);` line, we call the run method from the add_set_item of the current builder e.g. build_letter.

In `build_letter`, `Before` is the passed `State` from `build_set`. By calling the `run` method on `Before` (e.g. `return Before::type::run( char_value() | after );`) we jump to the previous `build_set`, but with the extended `after` parameter. In the last `build_letter` we have `empty_set` in `Before`, as we started the folding with that.

We have seen how `char_group` and `set` work and how the sub-parsers of `set_item` should be modified. Let's see how the `range_exp` parser looks like to have everything to build the missing `group` parser.

As mentioned earlier, `range_exp` needs the `add_set_item` interface only to be implemented, because it cannot occur outside `set`. To parse a range we should accept two kind of sequences: `number - number` and `letter - letter`.

With `one_of` and `sequence` we can write this easily.

```
typedef metaparse::transform<
  metaparse::one_of<
    metaparse::sequence< xlxpressive::number, metaparse::
        lit_c<'-'>, xlxpressive::number>,
    metaparse::sequence< xlxpressive::letter, metaparse::
        lit_c<'-'>, xlxpressive::letter>
  >,
  xlxpressive::build_range
>
range_exp;
```

What `build_range` should do is to put together the received results and return the initialized `range` object. To extract the results from the `sequence` we can use the `boost::mpl::at_c` method again, hence we get the elements on the 0 and 2 indices. As we only use `range` within a `set` we only define its `add_set_item` interface.

```
struct build_range
{
  template <class Seq> struct apply {
    typedef apply type;
    template <class T> static sregex run(T after) {
      return after;
    }
    template <class Before> struct add_set_item {
      typedef add_set_item type;
      template <class T> static sregex run(T after) {
        return Before::type::run( range( at_c<Seq, 0>::
            type::char_value(), at_c<Seq, 2>::type::
            char_value() ) | after );
      }
    };
  };
};
```

The newly introduced `char_value` method of `build_letter` and `build_number` have been used here. We need to do this, because the `range Xpressive` object needs two `char` arguments.

We have written everything now to be able to parse a `group`. As we did so far we follow the grammar to compose the parser.

```
group ::= '[' (char_group|'^'? set)
```

We'll need a `build_group` metafunction which simply calls the `run` method of its parameter.

```
struct build_group
{
  template <class G>
  struct apply
  {
    typedef apply type;
    static xpressive::sregex run(int &bracket_counter)
    {
      return G::type::run(bracket_counter);
    }
  };
};
```

To write the `group` parser itself we need a `sequence` with the opening square bracket as first element to be parsed and then a `one_of` to parse either a `char_group` or a `set`. The "`'^'?`" part expresses that we might negate the `set` construct, e.g. `[^a]` to match any character except 'a'. To handle this we can use the same technique we used for `unary_item` in section 6.2: with `one_of` and `lit_c` we parse the '^' character or we just return an 'x' with the `return_` parser to express we shouldn't negate the `set`. We can name it `may_negate` with an implementation like this:

```
typedef metaparse::one_of<
  metaparse::lit_c<'^'>,
  metaparse::return_< mpl::char_<'x'> >
>
may_negate;
```

It makes the definition of the `group` parser more readable.

```
typedef metaparse::transform<
  metaparse::last_of<
    metaparse::lit_c<'['>,
    metaparse::one_of<
      xlxpressive::char_group,
      metaparse::transform<
```

```
        metaparse::sequence<
          may_negate,
          xlxpressive::set
        >,
        xlxpressive::eval_set_sign
      >
    >
  >,
  xlxpressive::build_group
>
group;
```

With `eval_set_sign` we can return a positive or a negative `set`. It makes the decision based on the passed character ('^' or 'x'). We can negate a `set` with the `'~'` operator in `Xpressive`. We only show the implementation of `positive_set`, because `negative_set` can be done the same way with the previously described differences.

```
//[...] -> set[...]
template <class S>
struct positive_set
{
  typedef positive_set type;
  static xpressive::sregex run(int &bracket_counter)
  {
    return S::type::run(bracket_counter);
  }
};
```

We simply call the `static run` method of the received `eval_set` result (`S`). It can be this simple, because `eval_set_sign` evaluates whether we should return a `set` or a negated `set`.

Let's see how it can be implemented:

```
template <char A, class Set> struct eval_set_sign_impl;
template <class Set> struct eval_set_sign_impl<'x', Set>
    : positive_set<Set> {};

struct eval_set_sign
{
  template <class Seq>
  struct apply :
    eval_set_sign_impl<
      mpl::front<Seq>::type::value,
      mpl::back<Seq>
    > {};
};
```

We've used the same kind of pattern matching with template specialization in the case of `eval_set_sign_impl` what we've seen at `unary_item`. In `eval_set_sign`, we call type::value on the first argument of `eval_set_sign_impl`, because we want to pass the `char` value of `may_negate`.

In this chapter we've seen how we can implement the parsers for all the main elements of our grammar. These parsers have been created gradually following our grammar using a bottom-to-top approach. A few examples have been shown how these parsers can be used to generate static `Xpressive` objects through `run` method call-chains. We've covered some advanced topics too e.g. how the continuous-passing style can be used in template metaprogramming `DSL` integration to build grammar elements with generated types.

7 Conclusion

Our goal was to show how we can use template metaprogramming to provide DSL embedding. The DSL we've chosen was `Boost.Xpressive` whom domain is regular expression. It provides an approach called "static regex" where we can write our regex with `C++` expressions. The problem is that its syntax is very different than the original regular expression syntax. Our aim was to provide a new interface for static `Xpressive` which enables us to specify compile-time checked regular expressions as strings.

First, we've written a grammar for these regular expressions. Then we've used this grammar to walk through the process how we can build compile time parsers with the help of the `Metaparse` library. We've successfully built all the parsers while we've encountered and solved more and more advanced problems. The parsers created following this approach can parse separate grammar elements. We've shown how the `run` method chain can construct a static `Xpressive` object.

We have created a working implementation of this library as an open source project available on `github` [21]. It contains more than 40 generated test cases showing what our solution can do at this stage.

As we've introduced the `REGEX` macro earlier, we can do a comparison to demonstrate what our approach is capable for. The below lines show grammatically equivalent regular expressions. The first line of a block is the static `Xpressive` example built by hand and the second line shows how it looks like as an input of our new interface. The \\ characters show extra line-breaks.

```
(s1= +_w) >> ' ' >> (s2= +_w) >> '!';
REGEX("(\\w+) (\\w+)!");

'$' >> +_d >> '.' >> _d >> _d;
REGEX("\\$\\d+\\.\\d\\d")));

bos >> set[as_xpr('a')|'b'|'c'|'d'] >> range('3','8') >> \\
  '.' >> 'f' >> 'o' >> 'o'  >> eos;
REGEX("^[abcd][3-8]\\.foo$");
```

```
bos>>(s1=+range('0','9')>>!(s2='.'>>*range('0','9')))>> \\
  (s3=set[as_xpr('C')|'F'])>>eos;
REGEX("^([0-9]+(\\.[0-9]*)?)([CF])$");
```

Our new interface is more readable and natural, while it still ensures compile time validation. Someone who understands regular expressions should be able to use our solution easily at the very first time, while the original interface needs to be studied first.

As a summary we can say that using C++ template metaprogrammimg is a good approach for embedding domain specific languages, if we construct our own parsers with the help of the `Metaparse` parser combinators. These have been created in such a way that we can easily combine them to parse our grammar elements. With their combinations we can tackle complex problems, like the smooth integration of regular expressions. We've also seen that a grammar should be created first, if we start embedding a `DSL` and that how important the test cases are, when we work on template metaprograms.

It can be subject of future works how this approach can be used for embedding other `DSL`s like `SQL` expressions or `Spirit parsers`.

References

1. D. Abrahams, A. Gurtovoy, C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley, 2004, [400], ISBN-0321-22725-6
2. Y. Gil, K. Lenz, Simple and Safe SQL queries with C++ templates In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.
3. J. Siek, A. Lumsdaine, Essential Language Support for Generic Programming, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73–84.
4. C. Simonyi, M. Christerson, S. Clifford, Intentional software, In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, October 22-26, 2006, Portland, Oregon, USA, pp. 451465.
5. Á. Sinkovics, Z. Porkoláb, Domain-Specific Language Integration with C++ Template Metaprogramming, In Marjan Mernik (Ed): Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. Published by Information Science Reference (an imprint of IGI Global), Hershey PA, USA. ISBN 978-1-4666-2092-6 (hardcover) - ISBN 978-1-4666-2093-3 (ebook) - ISBN 978-1-4666-2094-0 pp. 33-56.
6. Á. Sinkovics, Z. Porkoláb, Expressing C++ Template Metaprograms as Lambda Expressions, In Zoltán Horváth, Viktória Zsók, Peter Achten, Pieter Koopman (eds): Proceedings of Tenth Symposium on Trends in Functional Programming, Komrno, Slovakia, 2-4 June 2009, pp. 97-111.
7. E. Unruh, Prime number computation, 1994, ANSI X3J16-94-0075/ISO WG21-462
8. Visser, E., Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9., In C. Lengauer et al., editors, Domain-Specific Program Generation, vol. 3016 of Lecture Notes in Computer Science, 2004, pp. 216–238. Spinger-Verlag, June 2004.

9. The auto specified in C++11
http://en.cppreference.com/w/cpp/language/auto

10. Boost.Proto
http://www.boost.org/doc/libs/1_53_0/doc/html/proto.html

11. Continuation-passing style on Wikipedia
http://en.wikipedia.org/wiki/Continuation-passing_style

12. Icon, The Icon Programming Language
http://www.cs.arizona.edu/icon

13. Katahdin
http://www.chrisseaton.com/katahdin

14. B. Milewski, Haskell and C++ template metaprogramming
http://bartoszmilewski.wordpress.com/2009/10/26/
haskellc-video-and-slides

15. B. Milewski, E. Niebler
Compile-Time/Run-Time Functional Programming in C++
http://2012.cppnow.org/session/variadic-template-
metaprogramming-using-monads

16. The MPL Reference Manual
http://www.boost.org/doc/libs/1_53_0/libs/mpl/doc/refmanual.html

17. The User Manual of mpllibs metaparse
http://abel.web.elte.hu/mpllibs/metaparse/manual.html

18. Á. Sinkovics, Z. Porkoláb,
Metaparse – Compile-time parsing with template metaprogramming
http://2012.cppnow.org/session/metaparse-compile-time-parsing-
with-template-metaprogramming

19. The Stratego Program Transformation Language
http://strategoxt.org/

20. Template Haskell
http://www.haskell.org/haskellwiki/Template_Haskell

21. The xlxpressive library
https://github.com/istvans/mpllibs/tree/master/mpllibs/xlxpressive

22. The XMF programming language
http://itcentre.tvu.ac.uk/clark/xmf.html

23. The User's Guide of Xpressive
http://www.boost.org/doc/libs/1_53_0/doc/html/xpressive/
user_s_guide.html

24. Xpressive – "Nested Regexes and Sub-Match Scoping" and "Nested Results"
http://www.boost.org/doc/libs/1_53_0/doc/html/xpressive/
user_s_guide.html#boost_xpressive.user_s_guide.grammars_and_nested
_matches.nested_regexes_and_sub_match_scoping

A The grammar

```
reg_exp ::= seq ('|' seq)*
seq ::= unary_item*
unary_item ::= item (('*'|'+'|'?'|repeat) '?'?)?
repeat ::= '{' (number ',' number|',' number| number ',') '}'
item ::= bos|eos|any|bracket_exp|group|set_item
set_item ::= bschar|number|letter|non_alphabet
non_alphabet ::= space|','|';'|':'|'='|'~'|'<'|'>'|
                 '-'|'_'|'!'|'@'|'#'|'%'|'&'|'/'
letter ::= 'A'-'Z'|'a'-'z'
number ::= '0'-'9'
bos ::= '^'
eos ::= '$'
any ::= '.'
bracket_exp ::= '(' (reg_exp|qexp) ')'
qexp ::= '?' (no_back_ref|icase|keep|before|not_before|
             after|not_after|mark_tag_create|mark_tag_use) reg_exp
no_back_ref      ::= ":"
icase            ::= "i:"
keep             ::= '>'
before           ::= '='
not_before       ::= '!'
after            ::= "<="
not_after        ::= "<!"
mark_tag_create  ::= "P<" name '>'
mark_tag_use     ::= "P=" name
name             ::= letter+
bschar ::= '\' (bs_backslash|bs_back_ref|bs_boundary|
               bs_digit|bs_word|bs_space|bs_new_line|
               bs_caret|bs_dollar|bs_full_stop|bs_plus)
bs_backslash ::= '\'
bs_back_ref  ::= number
bs_boundary  ::= 'b'|not_bs_boundary
not_bs_boundary ::= 'B'
bs_digit     ::= 'd'|not_bs_digit
not_bs_digit ::= 'D'
bs_word      ::= 'w'|not_bs_word
not_bs_word  ::= 'W'
bs_space     ::= 's'|not_bs_space
not_bs_space ::= 'S'
bs_new_line  ::= "r\n"|'n'
bs_caret     ::= '^'
bs_dollar    ::= '$'
bs_full_stop ::= '.'
bs_plus      ::= '+'
```

```
group ::= '[' (char_group|'^'? set)
set ::= (range_exp|set_item)+ ']'
range_exp ::= number set_num|letter set_abc|set_item
set_num     ::= '-' num_range|set_item
num_range   ::= number
set_abc     ::= '-' abc_range|set_item
abc_range   ::= letter
spaces ::= space*
space ::= ' '|'\n'|'\t'|'\r'
char_group  ::= "[:" ('a' set_a|'b' set_b|'c' set_c|'d' set_d|
                     'g' set_g|'l' set_l|'p' set_p|'s' set_s|
                     'u' set_u|'x' set_x|set)
set_a       ::= 'l' set_al|set
set_al      ::= 'n' set_aln|'p' set_alp|set
set_aln     ::= 'u' set_alnu|set
set_alnu    ::= 'm' set_alnum|set
set_alnum   ::= ':' set_alnumT|set
set_alnumT  ::= ']' set_alnumX|set
set_alnumX  ::= ']'
set_alp     ::= 'h' set_alph|set
set_alph    ::= 'a' set_alpha|set
set_alpha   ::= ':' set_alphaT|set
set_alphaT  ::= ']' set_alphaX|set
set_alphaX  ::= ']'
set_b       ::= 'l' set_bl|set
set_bl      ::= 'a' set_bla|set
set_bla     ::= 'n' set_blan|set
set_blan    ::= 'k' set_blank|set
set_blank   ::= ':' set_blankT|set
set_blankT  ::= ']' set_blankX|set
set_blankX  ::= ']'
set_c       ::= 'n' set_cn|set
set_cn      ::= 't' set_cnt|set
set_cnt     ::= 'r' set_cntr|set
set_cntr    ::= 'l' set_cntrl|set
set_cntrl   ::= ':' set_cntrlT|set
set_cntrlT  ::= ']' set_cntrlX|set
set_cntrlX  ::= ']'
set_d       ::= 'i' set_di|set
set_di      ::= 'g' set_dig|set
set_dig     ::= 'i' set_digi|set
set_digi    ::= 't' set_digit|set
set_digit   ::= ':' set_digitT|set
set_digitT  ::= ']' set_digitX|set
set_digitX  ::= ']'
```

```
set_g       ::= 'r' set_gr|set
set_gr      ::= 'a' set_gra|set
set_gra     ::= 'p' set_grap|set
set_grap    ::= 'h' set_graph|set
set_graph   ::= ':' set_graphT|set
set_graphT  ::= ']' set_graphX|set
set_graphX  ::= ']'
set_l       ::= 'o' set_lo|set
set_lo      ::= 'w' set_low|set
set_low     ::= 'e' set_lowe|set
set_lowe    ::= 'r' set_lower|set
set_lower   ::= ':' set_lowerT|set
set_lowerT  ::= ']' set_lowerX|set
set_lowerX  ::= ']'
set_p       ::= 'r' set_pr|'u' set_pu|set
set_pr        ::= 'i' set_pri|set
set_pri       ::= 'n' set_prin|set
set_prin      ::= 't' set_print|set
set_print     ::= ':' set_printT|set
set_printT    ::= ']' set_printX|set
set_printX    ::= ']'
set_pu        ::= 'n' set_pun|set
set_pun       ::= 'c' set_punc|set
set_punc      ::= 't' set_punct|set
set_punct     ::= ':' set_punctT|set
set_punctT    ::= ']' set_punctX|set
set_punctX    ::= ']'
set_s       ::= 'p' set_sp|set
set_sp      ::= 'a' set_spa|set
set_spa     ::= 'c' set_spac|set
set_spac    ::= 'e' set_space|set
set_space   ::= ':' set_spaceT|set
set_spaceT  ::= ']' set_spaceX|set
set_spaceX  ::= ']'
set_u       ::= 'p' set_up|set
set_up      ::= 'p' set_upp|set
set_upp     ::= 'e' set_uppe|set
set_uppe    ::= 'r' set_upper|set
set_upper   ::= ':' set_upperT|set
set_upperT  ::= ']' set_upperX|set
set_upperX  ::= ']'
set_x       ::= 'x' set_xd|set
set_xd      ::= 'd' set_xdi|set
set_xdi     ::= 'i' set_xdig|set
set_xdig    ::= 'g' set_xdigi|set
```

```
set_xdigi   ::= 'i' set_xdigit|set
set_xdigit  ::= ':' set_xdigitT|set
set_xdigitT ::= ']' set_xdigitX|set
set_xdigitX ::= ']'
```

Fig. 1. The syntax tree of the `^([abc][1-5])?x$` regular expression

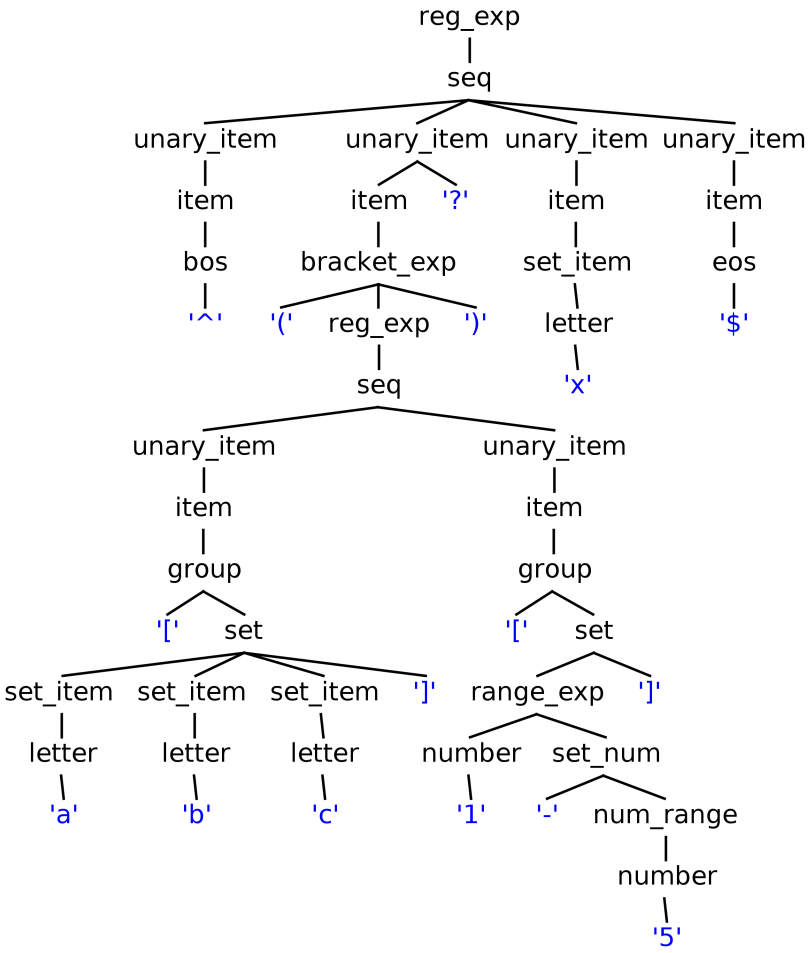

Fig. 2. Continuation-Passing Style