Using RE2C

Nov 20, 2017

RE2C is a popular open-source lexer generator for C and C++, with a flexible interface. As a code generator, it needs to be configured to access the user's data stream. The examples in their documentation assume access via pointers, but it's flexible enough to handle most C++ iterators, if configured correctly. I will demonstrate that in this post.

Basic Usage

RE2C runs as a preprocessor, replacing specially tagged comments with code. For example:

```
auto token = YYCURSOR;   // remember token start

/*!re2c
   "a" | "alpha" | "aleph" | "aardvark" {
      return std::string(token, YYCURSOR);
   }
   * { return std::nullopt; }
*/
```

The comment beginning with `!re2c` produces a lexer that uses `YYCURSOR` as an iterator and recognizes one of four tokens ("a", "alpha", "aleph", or "aardvark") and returns an empty optional if none are recognized.

The generated code assumes YYCURSOR is pointer-like:

```
YYCTYPE yych;
yych = *YYCURSOR;
if (yych == 'a') goto yy4;
++YYCURSOR;
{ return std::nullopt; }
```

You can see that if our character sequence did not begin with an 'a' we skip it and flag an error by returning an empty optional. The next part gets a little more interesting:

```
yy4:
   yych = *(YYMARKER = ++YYCURSOR);
   if (yych == 'a') goto yy6;
   if (yych == 'l') goto yy8;
yy5:
   {
      return std::string(token, YYCURSOR);
   }
```

The lexer always returns the longest valid token, so we branch to check for "aardvark" if a second 'a' has followed the first, or to check for "aleph" and "alpha" if an 'l' appears. If neither of those, we return just the "a", a valid token, which we have already recognized.

This bit of code also introduces `YYMARKER`, another pointer-like variable used for *backtracking*. When presented with multiple lexing options, RE2C will investigate each and restore the state of `YYCURSOR` from it as needed.

YYCURSOR, YYMARKER, and Concepts

We want our generated lexer to be able to operate on any kind of iterator, not just pointers. That is, instead of an interface like this:

```
using YYCTYPE = char;   // or unsigned char, wchar_t etc...
std::optional<Token>
lex(YYCTYPE * & begin, YYCTYPE * end) {
...
```

we want one like this:

```
template<typename Iter>
std::optional<Token>
lex(Iter & begin, Iter end) {
using YYCTYPE = std::iterator_traits<Iter>::value_type;
...
```

We need to know what kinds of operations RE2C might perform on its inputs and what their semantics are, or put another way, what Concepts the iterators must satisfy. From the generated code we can see that at least a ForwardIterator is required, because we save and restore `YYCURSOR` (making it "multipass"). What other requirements might there be? The RE2C docs helpfully explain. Let's look at a representative sample:

Expression	Iterator Concept
`++YYCURSOR`	any
`yych = *YYCURSOR`	any
`YYMARKER = ++YYCURSOR`	Forward
`if ((YYLIMIT - YYCURSOR) < n)`	RandomAccess

Although RE2C is plainly designed for use with pointers (which are RandomAccessIterators) we can almost implement all the required operations with a ForwardIterator. This would greatly expand the kinds of data we could operate on, allowing e.g. a buffering adapter on top of an input stream (such as

boost::spirit::istream_iterator) vs. loading an entire file into memory.

Through the use of its generic API RE2C will let you substitute your own code for its defaults. We can replace the limit test code from above by doing this:

```
#define YYLESSTHAN(n) (std::distance(YYCURSOR, YYLIMIT) < n)
```

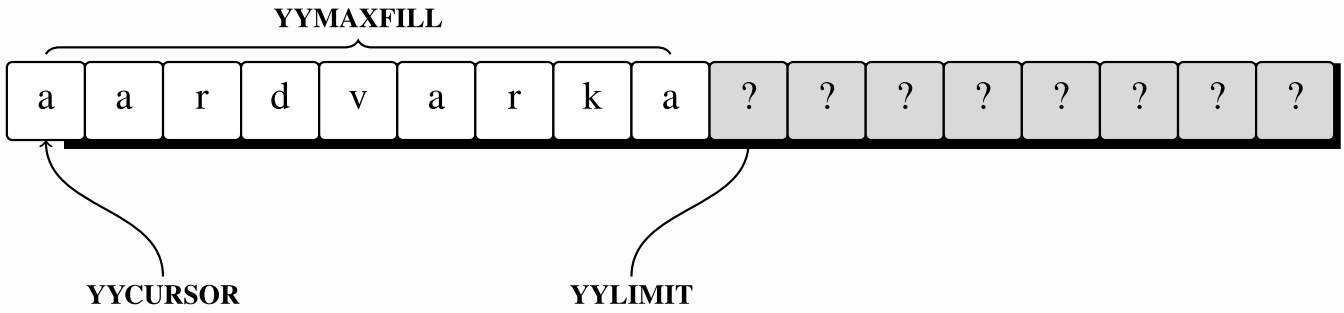
`std::distance` will do the appropriate thing for whatever kind of iterator you supply, allowing us to support ForwardIterators, though at a high cost - a linear scan through the remaining characters on each token. But why is `YYLESSTHAN` needed at all?

YYLIMIT, YYFILL, and Lookahead

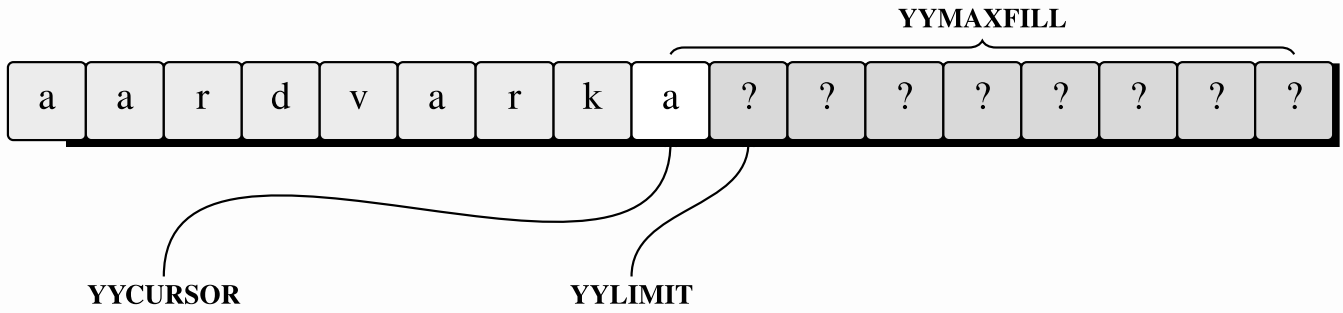
In its default configuration RE2C avoids testing for EOF on every character by checking larger chunks at a time, with this code:

```
if ((YYLIMIT - YYCURSOR) < YYMAXFILL) YYFILL(YYMAXFILL);
```

Where `YYCURSOR` and `YYLIMIT` are the boundaries of the input, `YYMAXFILL` is a function of the complexity of the lexer, and `YYFILL()` is defined by the user to either supply more data, or exit. This is very efficient, and allows for buffering, too. Unfortunately it introduces a complication: how should one handle the end of the input? RE2C expects at least a fixed amount (`YYMAXFILL`) of data to be present, but at the end there will generally be less, even in valid input. For example:



Initially the lexer has more than `YYMAXFILL` (8, in our example) characters remaining, and the initial check passes. “aardvark” is recognized and the lexer returns with `YYCURSOR` pointing to the remaining input. Now the lexer state looks like this:



When the lexer resumes work, only one character remains. Without the `YYMAXFILL` test the generated state machine would search into the invalid region starting at `YYLIMIT`, checking for longer words that start with ‘a’. But *with* the test, it will skip the remaining valid token “a” entirely. We need an efficient solution that doesn’t miss any input.

Strategy 1: Copy and Pad

The RE2C docs suggest that if there is some character you know will never be part of a matching expression, you can add `YYMAXFILL` of them after the end of your sequence. Then you can define YYFILL to exit the lexer when the unpadded input is exhausted. That works fine if you are buffering input anyway (e.g., with very large files), but requires an extra copy of the input otherwise.

Strategy 2: Fake a Padded Range

If you prefer not to copy the input data you can fake the padding on the end by wrapping the original iterator in a new one that supplies pad characters when the original range is exhausted. The cost is an extra comparison in every increment or dereference, but no copies are required. As in Strategy 1, we turn off the YYLIMIT check.

Making a Padded Range

The Boost Iterator Library is full of features for transforming and composing iterators. The one most useful in our case is the iterator facade, which helps you build a proper Iterator with a little configuration. We define the class like this:

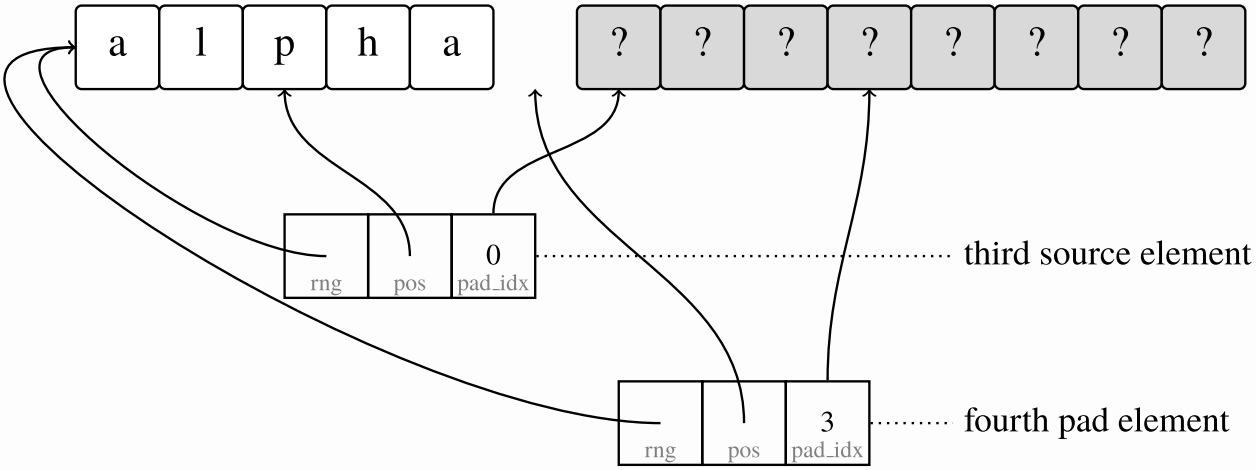
```
template<typename SrcIt> struct iterator
: boost::iterator_facade<
    iterator,
    typename std::iterator_traits<SrcIt>::value_type,
    typename std::iterator_traits<SrcIt>::iterator_category,
    typename std::iterator_traits<SrcIt>::value_type const&>
{
```

Our wrapped iterator is a read-only iterator of the same category as the source. Inside the class we store the position within the source range and the

index within the “padding”, as well as a pointer to the parent range:

```
padded_range<SrcIt, Pad, Count> const * rng_;  // parent range
SrcIt           pos_;                  // position within src
std::size_t     pad_idx_;                  // position within padding
```

A sample wrapped range with two iterators:



The first indicates a position within the original range; the second gives an offset within the padding. The padding is “virtual”, existing only as a value and a count, so an index is enough.

We also need to define a few member functions required by `iterator_facade`. For example:

```
typedef typename std::iterator_traits<SrcIt>::difference_type difference_type;
difference_type distance_to(iterator const& other) const {
    return std::distance(pos_, other.pos_) +
        (difference_type)(other.pad_idx_) - (difference_type)(pad_idx_);
}
```

Here we took advantage of `std::distance` to get the maximum efficiency available from the source iterator, i.e. constant time execution if available, and linear otherwise.

Using the Padded Range with RE2C

Now all that’s necessary is to supply our templated lexer with appropriately padded versions of the original iterators:

```
using padded_range_t = padded_range<Iter, 'x', YYMAXFILL>;  // no 'x' in tokens
padded_range_t padded(begin, end);
auto it = padded.begin();
while (it != padded.end_input()) {    // boundary between src and padding
    // some data remains
    auto tok = lex(it, padded.end()); // RE2C-generated lexer
    if (tok) {
        // good; do something with *tok
    else {
        // empty optional - probably an error (no match)
    }
}
```

and now our input sequence can be ForwardIterator based.

For full examples see this repo.

Engineering. Software. Software Engineering.

Engineering. Software. Software Engineering.

jefftrull
jaafartrull

Some thoughts on EDA, C++, and electronics.