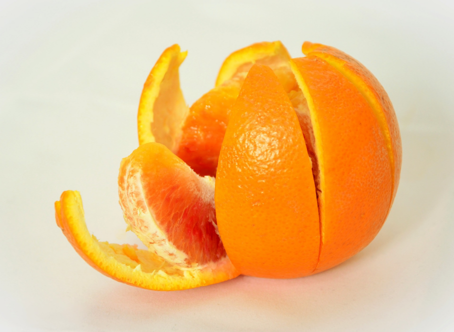
Boost.Spirit Tutorial

Parsing Structured Text with C++

Timo Bingmann

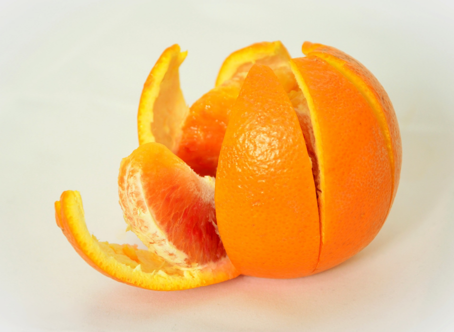
12. September 2018 @ Karlsruhe C++ Meetup



Boost.Spirit Tutorial
Parsing Structured Text with C++

Timo Bingmann

12. September 2018 @ Karlsruhe C++ Meetup



What is This Talk About?

How to parse the following strings using C++?

- "`5`"?
- "`[5, 42, 69, 256]`" as a `std::vector<int>`?
- "AAPL;Apple;252.50;" into a `struct Stock` from CSV?
- "`y = 6 * 9 + 42 * x`" as an expression?
- "2018-09-10-13-34;12017.39;12018.01;12014.28;2680;0;" as a stock market bar?
- "Bars(5m,Ticks(AAPL) * Ticks(EURUSD) / Ticks(DAX))" as a calculation with parameterized operations?
- Or HTML and other markup?

```
<h1>Example for <b>C++ HTML Parser<b></h1>
This HTML <b>snippet</b> parser can also interpret
*Markdown* style.
```

Parsing Structured Text

People think:

“I need no parser… all my data is in JSON.”

Parsing Structured Text

People think:

"I need no parser... all my data is in JSON."

And the truth is:

- Any reading of strings into (numeric) variables is parsing.
- Text is a common and future-proof way to store information.

Examples:

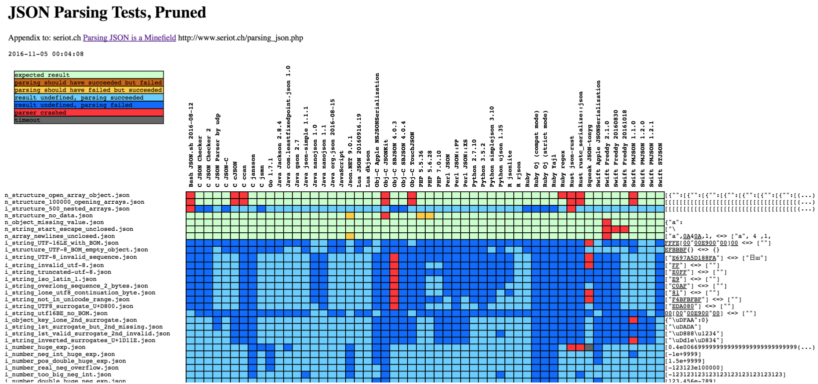
- Parsing numbers, email addresses, CSV files, arithmetic expressions, binary data, or any structured user input.
- Reading HTML documents, JSON data, HTTP protocol lines, or program code.

Parsing Structured Text

People think:

“I need no parser... all my data is in JSON.”

And parsing JSON is actually a minefield:
`http://seriot.ch/parsing_json.php`



Example of Stock Market Data

```
BEGINDATA TTS-514562 INTRADAY1 1000
2018-09-10-13-32;12010.62;12012.96;12010.41;12012.80;921;0;
2018-09-10-13-33;12013.01;12017.45;12013.01;12017.39;2866;0;
2018-09-10-13-34;12017.39;12018.01;12014.28;12014.39;2680;0;
2018-09-10-13-35;12014.39;12015.14;12014.21;12014.57;1262;0;
2018-09-10-13-36;12014.57;12016.30;12014.57;12016.23;1929;0;
2018-09-10-13-37;12016.28;12016.28;12014.79;12015.08;2486;0;
2018-09-10-13-38;12014.96;12015.61;12014.29;12015.61;2085;0;
2018-09-10-13-39;12015.61;12017.08;12015.61;12016.96;2440;0;
--packet end--
```

Boost Spirit Parser for Stock Market Data

```
Example:
2018-09-10-13-36;12014.57;12016.30;12014.57;12016.23;1929;0;

std::istringstream in(web.data());
std::string line;
struct TOhlcBar tick;
while (std::getline(in, line))
  tools::ParseOrDie(line,
    qi::uint_ >> '-' >> qi::uint_ >> '-' >> qi::uint_ >> '-' >>
    qi::uint_ >> '-' >> qi::uint_ >> ';' >>
    (qi::double_ | qi::lit("N/A") >> qi::attr(NAN)) >> ';' >>
    (qi::double_ | qi::lit("N/A") >> qi::attr(NAN)) >> ';' >>
    (qi::double_ | qi::lit("N/A") >> qi::attr(NAN)) >> ';' >>
    qi::double_ >> ';' >> qi::ulong_long >> ";0;",
    tick.ts.year, tick.ts.month, tick.ts.day,
    tick.ts.hour, tick.ts.minute,
    tick.open, tick.high, tick.low, tick.close,
    tick.size);
}
```

Flashback: Grammars

Remember the Chomsky Hierarchy?

- Type 3:
- Type 2:
- Type 1:
- Type 0:

from Wikipedia

Flashback: Grammars

Remember the Chomsky Hierarchy?

- Type 3: **regular**
- Type 2: **context-free**
- Type 1: **context-sensitive**
- Type 0: **recursively enumerable**

from Wikipedia

Flashback: Grammars

Remember the Chomsky Hierarchy?

- Type 3: **regular**
  $\{S \to aA, \quad A \to aA, \quad A \to bB, \quad B \to bB, \quad B \to \varepsilon\}$ – $a^n b^m$
- Type 2: **context-free**
  $\{S \to aSb, \quad S \to ab\}$ – $a^n b^n$, or
  $\{S \to A, \quad A \to A \text{ '+' } A, \quad A \to P, \quad P \to P \text{ '*' } P, \quad P \to int\}$,
- Type 1: **context-sensitive**
  $\{S \to aBC, S \to aSBC, CB \to CZ, CZ \to WZ, WZ \to WC,$
  $WC \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc\}$ – $a^n b^n c^n$.
- Type 0: **recursively enumerable**

Grammars in Practice

- Type 3: **regular**
  Regular expressions! Now also available in C++11.
  (insert here a demo on how to use regex)

from https://xkcd.com/208/

Grammars in Practice

- Type 3: **regular**
  Regular expressions! Now also available in C++11.
  (insert here a demo on how to use regex)
  Also: re2c library (generates actual finite automatons).

But what if regex is not enough?

Grammars in Practice

- Type 3: **regular**
  Regular expressions! Now also available in C++11.
  (insert here a demo on how to use regex)
  Also: re2c library (generates actual finite automatons).

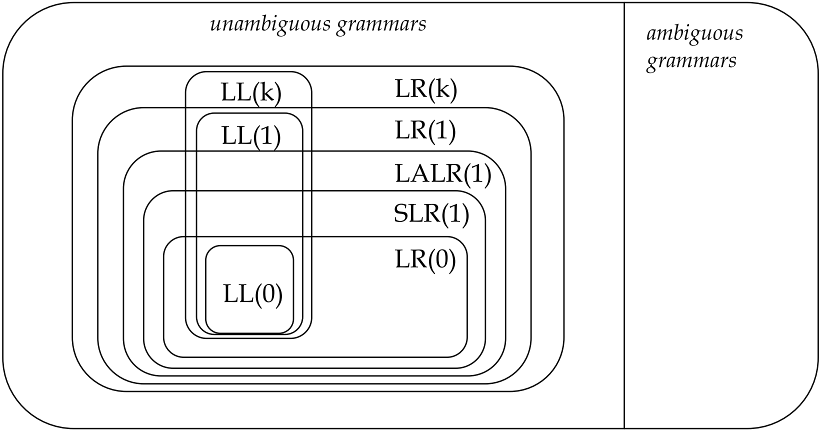
- Type 2: **context-free**
  Either code it by hand, or **use parser generators**.

  Example of a grammar in extended Backus-Naur form:

```
term     = sum, ('+', sum)*;
sum      = product, ('*', product)*;
product  = integer | group;
group    = '(', term, ')';
```

Grammars in Practice

Type 2: context-free subtypes:



from `http://web.stanford.edu/class/cs143/`

Grammars in Practice

Type 2: context-free subtypes:

- **LR(k)** shift-reduce rules, or
  "deterministic context-free" for pushdown automata

  Term → Sum

  Sum → Sum '+' Product, Sum → Product

  Product → Product '*' Product, Product → *int*

- **LL(k)** or **LL(*)**: recursive descent, left-most derivation

  Term → Sum,

  Sum → Prod, Sum → Prod Sum2,

  Sum2 → '+' Sum, Sum2 → '+' Sum Sum2,

  Prod2 → '*' Prod, Prod2 → '*' Prod Prod2,

  Prod → *int*, Prod → *int* Prod2

Parsing in Practice

complexity of grammar

Lex+Yacc (GNU bison+flex), AntLR, lemon, etc

See my flex-bison-cpp-example repository

Boost.Spirit

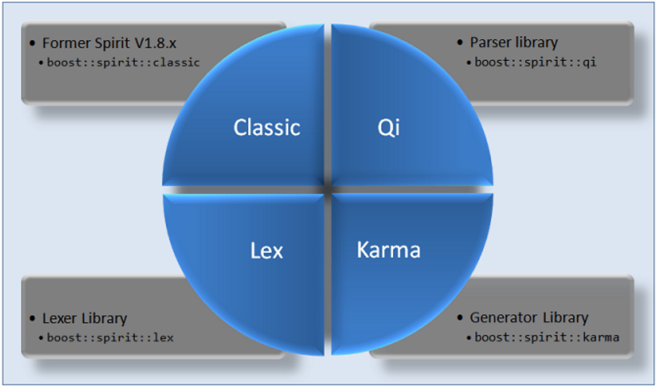
- + grammar as C++ code
- + header-only compilation
- + optimization → fast
- + tight integration with code
- – long compilation times
- – Boost dependency
- – ugly template errors
- – hard to learn...

`std::istream`, `sscanf()`

hand-written

Beware of security issues!

Boost Spirit



from Boost.Spirit documentation

Boost Spirit Documentation:
https://www.boost.org/doc/libs/1_68_0/libs/spirit/doc/html/

Grammar with Boost.Spirit

Extended Backus-Naur form:

```
expr     = product, ('+', product)*;
product  = factor, ('*', factor)*;
factor   = integer | group;
group    = '(', expr, ')';
```

Boost.Spirit's domain-specific "language" in C++:

```
expr     = product >> *('+' >> product);
product  = factor >> *('*' >> factor);
factor   = int_ | group;
group    = '(' >> expr >> ')';
```

Boost.Spirit Live Coding

1. Learn to walk and parse simple integers and lists.
   Parse "`5`", "`[5, 42, 69, 256]`".

2. Create a parser for a simple arithmetic grammar.
   Parse "`5 + 6 * 9 + 42`" and evaluate correctly.

3. Parse CSV data directly into a C++ struct.
   Parse "AAPL;Apple;252.50;" into a struct.

4. Create an abstract syntax tree (AST) from arithmetic.
   Parse "`y = 6 * 9 + 42 * x`" and evaluate with variables.

5. Ogle some more crazy examples, e.g. how to parse

```
<h1>Example for <b>C++ HTML Parser<b></h1>
This HTML <b>snippet</b> parser can also interpret
*Markdown* style and enables additional tags
to <% invoke(C++", 42) %> functions.
```

Questions?

Thank you for your attention.

Questions?

Source code examples used in talk available at

`https://github.com/bingmann/2018-cpp-spirit-parsing`

for self study.

More of my work: `https://panthema.net`