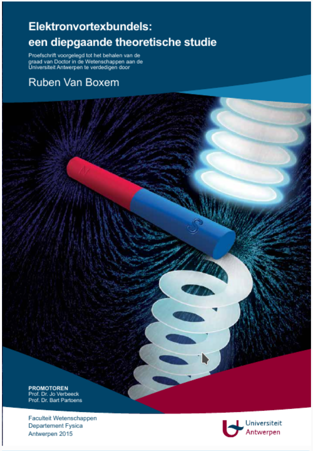
Parsing CSS with Boost.Spirit.X3

Ruben Van Boxem

February 4th 2019

Introduction

About me



medicim

https://github.com/rubenvb

SkUI

SkUI

TEACH YOURSELF
EVERYTHING

Modern C++next UI framework:

- signal/slot
- friendly API
- fast
- platform integration

SkUI

Modern C++next UI framework:

- signal/slot
- friendly API
- fast
- platform integration

SkUI

Modern C++next UI framework:

- signal/slot
- friendly API
- fast
- platform integration

SkUI

Modern C++next UI framework:

- signal/slot
- friendly API
- fast
- platform integration

All help is welcome: `www.github.com/skui-org/skui`

About this presentation

Introduction

Representing input

Practical Boost.Spirit

CSS

CSS in Spirit

Representing input

EBNF: Extended Backus Naur Form

A representation of a context-free grammar that can express itself:

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
       | "'" | '"' | "=" | "|" | "." | "," | ";" ;
character = letter | digit | symbol | "_" ;

identifier = letter , { letter | digit | "_" } ;
terminal = "'" , character , { character } , "'"
         | '"' , character , { character } , '"' ;

lhs = identifier ;
rhs = identifier
    | terminal
    | "[" , rhs , "]"
    | "{" , rhs , "}"
    | "(" , rhs , ")"
    | rhs , "|" , rhs
    | rhs , "," , rhs ;

rule = lhs , "=" , rhs , ";" ;
grammar = { rule } ;
```

Reading input

Generally two ways to process input:

1. hand-written: recursive decent, state machine, other “ingenious" designs …
2. generated: ANTLR, YACC/Bison, Elkhound, Whale Calf, and some other large wild animals or parts thereof.

Reading input

Generally two ways to process input:

1. hand-written: recursive decent, state machine, other “ingenious" designs ...
2. generated: ANTLR, YACC/Bison, Elkhound, Whale Calf, and some other large wild animals or parts thereof.

Trade-off between

- learning curve
- readability & expressiveness
- maintainability
- bug-proneness
- performance

Steps and stones

input - “stream of bytes"

↓

lex: splitting into *tokens*

↓

parse: transform into more useful representation

↓

AST: Abstract Syntax Tree: data structure corresponding to original input

Steps and stones

input - “stream of bytes"

↓

lex: splitting into *tokens*

↓

parse: transform into more useful representation

↓

AST: Abstract Syntax Tree: data structure corresponding to original input

↓

Make millions

Steps and stones

input - “stream of bytes"

↓

lex: splitting into *tokens*

↓

parse: transform into more useful representation

↓

AST: Abstract Syntax Tree: data structure corresponding to original input

↓

Make millions

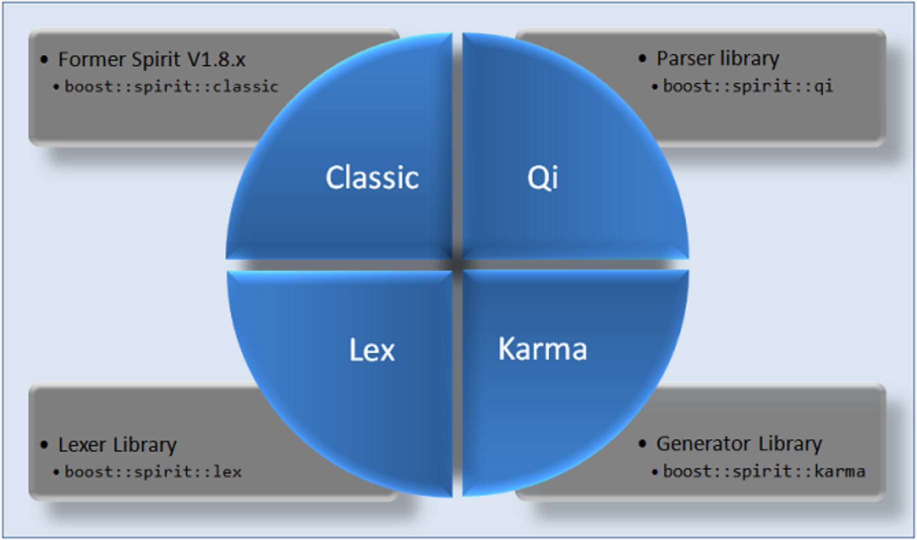
Practical Boost.Spirit

C++ talks grammar

Can we make C++ work for us?

1. express the content, not the I/O
2. separate concerns:
   representation — grammar — errors
3. easy on the eyes, given C++
4. easy to extend & debug

Boost.Spirit framework anno 2010



Boosting compatibility

Boost provides compatibility with nearly any platform and toolchain.

Boosting compatibility

Boost provides compatibility with nearly any platform and toolchain.

Spirit was written before C++ 11/14/17/20.

Built on top of some heavy-weight Boost components:

- **MPL**: meta template programming
- **Phoenix**: functional programming
- **Fusion**: generic sequence programming, limited introspection
- **Preprocessor**: macro magic

Boosting compatibility

Boost provides compatibility with nearly any platform and toolchain.

Spirit was written before C++ 11/14/17/20.

Built on top of some heavy-weight Boost components:

- **MPL**: meta template programming
- **Phoenix**: functional programming
- **Fusion**: generic sequence programming, limited introspection
- **Preprocessor**: macro magic

On the upside: the power of these libraries is well integrated into the constructs of Spirit.

Boost.Spirit.X3

Focuses on the *parser* component – replaces Qi

Dropped Phoenix integration which can be replaced quite lightly by e.g. C++ 14 lambdas and other functional features of modern C++ .

Seems to be somewhat feature-complete, but lacking some notable useful bits from its predecessors.

Basic concepts

- **parser**: object describing the form of a specific form of input:

```
x3::double_
```

Basic concepts

- **parser**: object describing the form of a specific form of input:

```
x3::double_
```

- **attribute**: synthesized “result” of a parser, can be nothing (`unused_type`)

Basic concepts

- **parser**: object describing the form of a specific form of input:

```
x3::double_
```

- **attribute**: synthesized "result" of a parser, can be nothing (`unused_type`)
- **skipper**: object describing what consists of "whitespace" and is skipped over when looking for the next matching token.

Basic concepts

- **parser**: object describing the form of a specific form of input:

```
x3::double_
```

- **attribute**: synthesized "result" of a parser, can be nothing (`unused_type`)
- **skipper**: object describing what consists of "whitespace" and is skipped over when looking for the next matching token.
- **rule**: an agglomeration of parsers providing *attribute scope* and metadata for error handling and debugging.

Parsing an input range

Two main entry points:

```
bool success = parse(first, last,
                     parser,
                     result);
```

```
bool success = phrase_parse(first, last,
                            parser,
                            skipper,
                            result);
```

Parser matched if `success == true`.

Complete match if also `first == last`.

Test bed boilerplate

```
int main()
{
  std::string input = /* bytes we want parsed */;
  const auto parser = /* This is where the magic goes */;

  auto first = input.begin();
  auto last = input.end();
  std::string result;
  bool success = x3::parse(first, last,
                           parser,
                           result);
  if(!success)
    std::cerr << "parsing failed.\n";
  else if(first != last)
    std::cerr << "parsing did not match full input.\n";
  else
    std::cout << "result: " << result << '\n';
}
```

Little bit of explanation

Example parser:

```
parser expression  →  attribute type
```

In case of no attribute: `x3::unused_type`

Little bit of explanation

Example parser:

```
parser expression  →  attribute type
```

In case of no attribute: `x3::unused_type`

Example input:

```
3.14  →  3.14
```

Spirit is greedy, so partial matches are often possible!

Warming up

```
double_  →  double
```

Warming up

```
double_    →    double
```

Matches:

```
3.14     →    3.14
1e-10    →    .0000000001
-5       →    -5.
```

Warming up

```
double_    →    double
```

Matches:

```
3.14     →    3.14
1e-10    →    .0000000001
-5       →    -5.
```

Does not match:

```
a string
0xFF    →    0.
mY_S1lly_P@sSw0rD
```

Warming up

```
double_    →    double
```

Matches:

```
3.14     →    3.14
1e-10    →    .0000000001
-5       →    -5.
```

Does not match:

```
a string
0xFF    →    0.
mY_S1lly_P@sSw0rD
```

```
int_    →    int
```

Warming up

```
double_    →    double
```

Matches:

```
3.14     →    3.14
1e-10    →    .0000000001
-5       →    -5.
```

Does not match:

```
a string
0xFF     →    0.
mY_S1lly_P@sSw0rD
```

```
int_    →    int
```

Matches:

```
0     →    0
2     →    2
-3    →    -3
```

Warming up

```
double_ → double
```

Matches:

```
3.14    → 3.14
1e-10   → .0000000001
-5      → -5.
```

Does not match:

```
a string
0xFF → 0.
mY_S1lly_P@sSw0rD
```

```
int_ → int
```

Matches:

```
0  → 0
2  → 2
-3 → -3
```

Does not match:

```
3.1415 → 3
EDA34CF
-1e0   → -1
```

Numeric parsers

Number parsers:

`float_`	`double_`	`long_double`	
`bin`	`oct`	`hex`	
`ushort_`	`ulong_`	`uint_`	`ulong_long`
`short_`	`long_`	`int_`	`long_long`

Numeric parsers

Number parsers:

```
float_   double_  long_double
bin      oct      hex
ushort_  ulong_   uint_   ulong_long
short_   long_    int_    long_long
```

Literal forms for all of the above except `bin`, `oct`, and `hex`:

```
float_(3.14f)   →   unused_type
ushort_(42)     →   unused_type
int_(365)       →   unused_type
```

Numeric parsers

Number parsers:

```
float_    double_    long_double
bin       oct        hex
ushort_   ulong_     uint_      ulong_long
short_    long_      int_       long_long
```

Literal forms for all of the above except `bin`, `oct`, and `hex`:

```
float_(3.14f)    →    unused_type
ushort_(42)      →    unused_type
int_(365)        →    unused_type
```

Guts to define custom numeric parsers

```
[u]int_parser<T, Radix, MinDigits, MaxDigits>
real_parser<T, RealPolicies>
```

Warming up...

```
"a string"    →    unused_type
```

Warming up...

```
"a string"  →  unused_type
```

Matches

```
a string  →  unused_type
```

Warming up...

```
"a string"    →    unused_type
```

Matches

```
a string   →   unused_type
```

Does not match:

```
not a string
```

Warming up...

```
"a string"  →  unused_type
```

Matches

```
a string  →  unused_type
```

Does not match:

```
not a string
```

```
alnum  →  char
```

Warming up...

```
"a string"  →  unused_type
```

Matches

```
a string  →  unused_type
```

Does not match:

```
not a string
```

```
alnum  →  char
```

Matches:

```
a  →  'a'
T  →  'T'
R  →  'R'
Q  →  'Q'
```

Warming up...

```
"a string"    →    unused_type
```

Matches

```
a string    →    unused_type
```

Does not match:

```
not a string
```

```
alnum    →    char
```

Matches:

```
a    →    'a'
T    →    'T'
R    →    'R'
Q    →    'Q'
```

Does not match:

```
four    →    'f'
_
*
&
```

Character parsers

Literals:

`'a'`	→	`unused_type`
`lit('a')`	→	`unused_type`
`char_('a')`	→	`unused_type`
`char_("abc")`	→	`unused_type`
`char_('a', 'z')`	→	`unused_type`
`char_(charset)`	→	`unused_type`

These correspond to the `std::is...` functions:

`alnum`	`alpha`	`blank`
`cntrl`	`digit`	`graph`
`print`	`punct`	`space`
`xdigit`	`lower`	`upper`

Parser operators

Expression	Attribute	Description
`!a`	unused	fails if a matches
`&a`	unused	matches if a matches
`-a`	`optional<A>`	match a zero or one time
`*a`	`vector<A>`	match a zero or more times
`+a`	`vector<A>`	match a one or more times
`a | b`	`variant<A, B>`	alternative: a or b
`a % b`	`vector<A>`	list of a delimited by b
`a >> b`	`tuple<A, B>`	sequence: a followed by b

Parser operators

<table>
<tr><th>Expression</th><th>Attribute</th><th>Description</th></tr>
<tr><td>!a</td><td>unused</td><td>fails if a matches</td></tr>
<tr><td>&a</td><td>unused</td><td>matches if a matches</td></tr>
<tr><td>-a</td><td>optional<A></td><td>match a zero or one time</td></tr>
<tr><td>*a</td><td>vector<A></td><td>match a zero or more times</td></tr>
<tr><td>+a</td><td>vector<A></td><td>match a one or more times</td></tr>
<tr><td>a | b</td><td>variant<A, B></td><td>alternative: a or b</td></tr>
<tr><td>a % b</td><td>vector<A></td><td>list of a delimited by b</td></tr>
<tr><td>a >> b</td><td>tuple<A, B></td><td>sequence: a followed by b</td></tr>
</table>

```
float_ >> int_    →    tuple<float, int>
```

Matches:

```
3.14 5    →    {3.14f, 5}
5 42      →    {5.f, 42}
```

Does not match:

```
5 3.14    →    {5.f, 3}
3.1 abc
```

Parser operators

Expression	Attribute	Description
`!a`	unused	fails if a matches
`&a`	unused	matches if a matches
`-a`	`optional<A>`	match a zero or one time
`*a`	`vector<A>`	match a zero or more times
`+a`	`vector<A>`	match a one or more times
`a | b`	`variant<A, B>`	alternative: a or b
`a % b`	`vector<A>`	list of a delimited by b
`a >> b`	`tuple<A, B>`	sequence: a followed by b

```
float_ | int    →    variant<float, int>
```

Matches:

```
3.14    →    3.14f
5       →    5
```

Does not match:

```
abc
3.14 5    →    3.14f
```

Parser operators

Expression	Attribute	Description
`!a`	unused	fails if a matches
`&a`	unused	matches if a matches
`-a`	`optional<A>`	match a zero or one time
`*a`	`vector<A>`	match a zero or more times
`+a`	`vector<A>`	match a one or more times
`a | b`	`variant<A, B>`	alternative: a or b
`a % b`	`vector<A>`	list of a delimited by b
`a >> b`	`tuple<A, B>`	sequence: a followed by b

```
!float_ >> int_    →    int
```

Matches:

```
a 5     →    5
* 42    →    42
```

Does not match:

```
3.14 5
* 3.14    →    3
```

Parser operators

Expression	Attribute	Description
!a	unused	fails if a matches
&a	unused	matches if a matches
-a	optional<A>	match a zero or one time
*a	vector<A>	match a zero or more times
+a	vector<A>	match a one or more times
a | b	variant<A, B>	alternative: a or b
a % b	vector<A>	list of a delimited by b
a >> b	tuple<A, B>	sequence: a followed by b

```
&float_ >> int_    →    int
```

Matches:

```
3.14 5      →    5
1e-10 20    →    20
```

Does not match:

```
5
3.14 abc
```

Parser operators

Expression	Attribute	Description
`!a`	unused	fails if a matches
`&a`	unused	matches if a matches
`-a`	`optional<A>`	match a zero or one time
`*a`	`vector<A>`	match a zero or more times
`+a`	`vector<A>`	match a one or more times
`a | b`	`variant<A, B>`	alternative: a or b
`a % b`	`vector<A>`	list of a delimited by b
`a >> b`	`tuple<A, B>`	sequence: a followed by b

```
float_ >> -int_  →  tuple<float, optional<int>>
```

Matches:

```
3.14 5  →  {3.14f, 5}
3.14    →  {3.14f, {}}
```

Does not match:

```
5 3.14  →  {5.f, 3}
3.14 a  →  {3.14f, {}}
```

Parser operators

Expression	Attribute	Description
`!a`	unused	fails if a matches
`&a`	unused	matches if a matches
`-a`	`optional<A>`	match a zero or one time
`*a`	`vector<A>`	match a zero or more times
`+a`	`vector<A>`	match a one or more times
`a	b`	`variant<A, B>`	alternative: a or b
`a % b`	`vector<A>`	list of a delimited by b
`a >> b`	`tuple<A, B>`	sequence: a followed by b

```
*float_    →    vector<float>
```

Matches:

```
           →  {}
3.14 5     →  {3.14f, 5.f}
```

Does not match:

```
3.14 a 5    →    {3.14}
a 5 3.14
```

Parser operators

Expression	Attribute	Description
`!a`	unused	fails if a matches
`&a`	unused	matches if a matches
`-a`	`optional<A>`	match a zero or one time
`*a`	`vector<A>`	match a zero or more times
`+a`	`vector<A>`	match a one or more times
`a | b`	`variant<A, B>`	alternative: a or b
`a % b`	`vector<A>`	list of a delimited by b
`a >> b`	`tuple<A, B>`	sequence: a followed by b

```
+int_    →    vector<int>
```

Matches:

```
1 2 3  →  {1, 2, 3}
9      →  {9}
```

Does not match:

```
5 3.14    → {5, 3}
```

Parser operators

Expression	Attribute	Description
`!a`	unused	fails if a matches
`&a`	unused	matches if a matches
`-a`	`optional<A>`	match a zero or one time
`*a`	`vector<A>`	match a zero or more times
`+a`	`vector<A>`	match a one or more times
`a | b`	`variant<A, B>`	alternative: a or b
`a % b`	`vector<A>`	list of a delimited by b
`a >> b`	`tuple<A, B>`	sequence: a followed by b

```
int_ % ','    →    vector<int>
```

Matches:

```
-3, 3    →  {-3, 3}
1           →  {1}
```

Does not match:

```
1 2 3    → {1}
```

String parsers

```
"literal string"
lit("literal string")
string("literal string")
```

Due to C++ grammar, the `lit` helper is only needed if a parser expression begins with a literal.

```
"number" >> ':' >> double
```

Needs to be

```
lit("number") >> ':' >> double_
```

Symbol table

Symbol table:

```
enum digit { one, two, /*...*/ nine };
struct table : x3::symbols<std::uint8_t>
{
  table(){ add("one", one)("two", two)/*..*/("nine", nine); }
} const digits;
```

Use as

```
digits → digit
```

Symbol table

Symbol table:

```
enum digit { one, two, /*...*/ nine };
struct table : x3::symbols<std::uint8_t>
{
  table(){ add("one", one)("two", two)/*..*/("nine", nine); }
} const digits;
```

Use as

```
digits → digit
```

Matches:

```
one  → 1
two  → 2
nine → 9
```

Symbol table

Symbol table:

```
enum digit { one, two, /*...*/ nine };
struct table : x3::symbols<std::uint8_t>
{
  table(){ add("one", one)("two", two)/*..*/("nine", nine); }
} const digits;
```

Use as

```
digits → digit
```

Matches:

```
one → 1
two → 2
nine → 9
```

Does not match:

```
forty-two
one hundred → 1
millions
```

Symbol table

Symbol table:

```
enum digit { one, two, /*...*/ nine };
struct table : x3::symbols<std::uint8_t>
{
  table(){ add("one", one)("two", two)/*..*/("nine", nine); }
} const digits;
```

Use as

```
digits → digit
```

Matches:

```
one  → 1
two  → 2
nine → 9
```

Does not match:

```
forty-two
one hundred → 1
millions
```

Powerful to parse directly to *e.g.* enums, but also more complicated types!

Auxiliary parsers

Special parsers useful in pre- and post-processing the attribute:

Expression	Attribute	Description
`eol`	unused	matches end of line
`oei`	unused	matches end of input
`eps`	unused	matches empty string (*i.e.* always)
`eps(b)`	unused	matches empty string if `b == true`
`attr(v)`	`decltype(v)`	consumes no input,
		produces `v` as attribute

Parser directives

Expression	Attribute	Description
`lexeme[a]`	A	disable skipper, pre-skip
`no_skip[a]`	A	disable skipper
`no_case[a]`	A	inhibit case-sensitivity
`omit[a]`	unused	ignore attribute of a
`matches[a]`	`bool`	return `true` if a matches
`raw[a]`	`iterator_range`	returns iterator range
`expect[a]`	A	throws exception if a does not match
`repeat[a]`	`vector<A>`	repeat a zero or more times
`skip[a]`	A	force skipping in `lexeme` or `no_skip`

Semantic actions

When you want to *act on* (the attribute of) a parser:

```
parser[action]
```

Semantic actions

When you want to *act on* (the attribute of) a parser:

```
parser[action]
```

This is useful for *e.g.*

- logging
- verification of external conditions
- other functionality not covered by the expressed attribute grammar

Semantic actions

When you want to *act on* (the attribute of) a parser:

```
parser[action]
```

This is useful for *e.g.*

- logging
- verification of external conditions
- other functionality not covered by the expressed attribute grammar

Required prototypes:

```
template<typename ContextType>
void action(const ContextType& context);

const auto action = [](auto& context){/*....*/};
```

Semantic action context

Function	Description
`_val`	A reference to the attribute of the innermost rule invoking the parser
`_where`	Iterator range to the input stream
`_attr`	A reference to the attribute of the parser
`_pass`	A reference to a `bool` flag that can be used to force the parser to fail

Semantic action context

Function	Description
`_val`	A reference to the attribute of the innermost rule invoking the parser
`_where`	Iterator range to the input stream
`_attr`	A reference to the attribute of the parser
`_pass`	A reference to a `bool` flag that can be used to force the parser to fail

Example:

Divide rule value by 100:

```
_val(context) =/ 100.;
```

Semantic action context

Function	Description
`_val`	A reference to the attribute of the innermost rule invoking the parser
`_where`	Iterator range to the input stream
`_attr`	A reference to the attribute of the parser
`_pass`	A reference to a `bool` flag that can be used to force the parser to fail

Example:

Increment the rule's value with the parsed attribute:

```
_val(ctx) += _attr(ctx);
```

Semantic action context

Function	Description
`_val`	A reference to the attribute of the innermost rule invoking the parser
`_where`	Iterator range to the input stream
`_attr`	A reference to the attribute of the parser
`_pass`	A reference to a `bool` flag that can be used to force the parser to fail

Example:

Make the parser fail:

```
_pass(ctx) = false;
```

Actionable examples

```
int main()
{
  std::string input = "1 2 3";
  auto first = input.begin();
  auto last = input.end();

  int value{};
  const auto add = [&value](auto& context) { value += x3::_attr(context); };
  bool result = x3::phrase_parse(first, last, *x3::int_[add], x3::blank);

  if(result)
  {
    if(first == last)
      std::cout << "parse succesful: " << value << '\n';
    else
      std::cout << "incomplete parse: " << value << '\n';
  }
  else
    std::cout << "parse unsuccesful\n";
}
```

Rule

Rules are named parsers. They help with

- debugging
- error reporting
- boundaries of semantic actions

Rule

Rules are named parsers. They help with

- debugging
- error reporting
- boundaries of semantic actions

Defining BOOST_SPIRIT_X3_DEBUG gives debug output:

```
<length>
  <try>2cm</try>
    <success></success>
  <attributes>[2, cm]</attributes>
</length>
```

Rule syntax

Defining a `rule`:

```
const auto a_rule = x3::rule<struct tag,
                             attribute_type>
                             {"rule name"}
                  = /*actual parser*/;
```

Rule syntax

Defining a `rule`:

```
const auto a_rule = x3::rule<struct tag,
                               attribute_type>
                               {"rule name"}
                  = /*actual parser*/;
```

In the presence of semantic actions, *auto attribute propagation* is disabled.

Re-enable with %=

```
                  %= /*actual parser*/;
```

Rule syntax

Defining a `rule`:

```
const auto a_rule = x3::rule<struct tag,
                             attribute_type>
                             {"rule name"}
                  = /*actual parser*/;
```

In the presence of semantic actions, *auto attribute propagation* is disabled.
Re-enable with %=

```
                 %= /*actual parser*/;
```

There are also ways of splitting declaration/definition through several macro's,

```
BOOST_SPIRIT_DECLARE
BOOST_SPIRIT_DEFINE
BOOST_SPIRIT_INSTANTIATE
```

CSS

Cascading Style Sheets

Hierarchical description of styling

Lives on the web

General enough to apply to other domains:

- XUL: Mozilla's XML User Interface Language
- Pango Text Attribute Markup Language
- QWidgets use QSS, a CSS dialect
- QML uses inline CSS-ish properties
- various derivatives using similar property names and values

Example

```
body {
    color: #325050;
    background: #fff;
    font-family: Arial, sans-serif;
    font-size: 70%;
}
```

More formally:

```
declaration = property, ":", value, ";" ;
rule_set = selector , "{", declaration+, "}" ;
```

Simple enough, right?

The Devil is in the details

Some details previously omitted for brevity:

- the format of `value` depends on which property

```
background-color: #f7f;
border-top-width: 1px;
border-collapse: collapse;
```

The Devil is in the details

Some details previously omitted for brevity:

- the format of `value` depends on which property

```
background-color: #f7f;
border-top-width: 1px;
border-collapse: collapse;
```

- properties such as `background-color` are more specific than `background`:

```
background: aquamarine url("faded.png")
            no-repeat fixed center;
```

The Devil is in the details

Some details previously omitted for brevity:

- the format of `value` depends on which property

```
background-color: #f7f;
border-top-width: 1px;
border-collapse: collapse;
```

- properties such as `background-color` are more specific than `background`:

```
background: aquamarine url("faded.png")
            no-repeat fixed center;
```

- unknown properties must be ignored

The Devil is in the details

Some details previously omitted for brevity:

- the format of `value` depends on which property

```
background-color: #f7f;
border-top-width: 1px;
border-collapse: collapse;
```

- properties such as `background-color` are more specific than `background`:

```
background: aquamarine url("faded.png")
            no-repeat fixed center;
```

- unknown properties must be ignored
- varying levels of extensions and strictness in existing implementations

CSS in Spirit

Extensive grammar and required

Basic CSS syntax:

```
selector { property : value ; }
```

Extensive grammar and required

Basic CSS syntax:

```
selector { property : value ; }
```

Every CSS property has different possible values with possibly other interpretations.

Extensive grammar and required

Basic CSS syntax:

```
selector { property : value ; }
```

Every CSS property has different possible values with possibly other interpretations.

Some properties possibly take multiple values, with different behaviour if less are specified.

Extensive grammar and required

Basic CSS syntax:

```
selector { property : value ; }
```

Every CSS property has different possible values with possibly other interpretations.

Some properties possibly take multiple values, with different behaviour if less are specified.

Some things are used in multiple (sub-)properties:

- length (relative and absolute)
- colour
- …

Skipping straight to the properties

Let's ignore the complexity of selectors for now and assume we have a great selector parser:

```
const auto selector = x3::rule<struct selector,
                                css::selector>
                                {"selector"}
                    = *(char_ - '{'); // not so great...
```

Skipping straight to the properties

Let’s ignore the complexity of selectors for now and assume we have a great selector parser:

```
const auto selector = x3::rule<struct selector,
                               css::selector>
                               {"selector"}
                    = *(char_ - '{'); // not so great...
```

```
const auto declaration_block
    = x3::rule<struct declaration_block,
               css::declaration_block>
               {"declaration_block"}
    = lit('{') >> +declaration >> '}';
```

Skipping straight to the properties

Let's ignore the complexity of selectors for now and assume we have a great selector parser:

```
const auto selector = x3::rule<struct selector,
                               css::selector>
                               {"selector"}
                  = *(char_ - '{'); // not so great...
```

```
const auto declaration_block
    = x3::rule<struct declaration_block,
               css::declaration_block>
               {"declaration_block"}
    = lit('{') >> +declaration >> '}';
```

```
const auto rule_set = x3::rule<struct rule_set,
                               css::rule_set,
                               {"rule-set"};
                  = selector >> declaration_block;
```

Properties and attributes

Goal: direct translation of CSS to a workable in-memory representation (no extra AST layer)

Properties and attributes

Goal: direct translation of CSS to a workable in-memory representation (no extra AST layer)

Each property can take on the values `inherit` and `initial`:

```
inline struct inherit_t final {} inherit;
inline struct initial_t final {} initial;

constexpr bool operator==(const inherit_t&, const inherit_t&) { return true; }
constexpr bool operator==(const initial_t&, const initial_t&) { return true; }
```

Properties and attributes

Goal: direct translation of CSS to a workable in-memory representation (no extra AST layer)

Each property can take on the values `inherit` and `initial`:

```
inline struct inherit_t final {} inherit;
inline struct initial_t final {} initial;

constexpr bool operator==(const inherit_t&, const inherit_t&) { return true; }
constexpr bool operator==(const initial_t&, const initial_t&) { return true; }
```

```
template<typename... ValueTypes>
using property = std::variant<ValueTypes...,
                              inherit_t,
                              initial_t>;
```

Declaration block

```
struct declaration_block
{
  property<css::align_content> align_content{};
  property<css::align_items> align_items{};
  property<css::align_self> align_self{};
  // ...
  css::background background{};
  css::border border{};
  // ...
};
```

Declaration block

```
struct declaration_block
{
  property<css::align_content> align_content{};
  property<css::align_items> align_items{};
  property<css::align_self> align_self{};
  // ...
  css::background background{};
  css::border border{};
  // ...
};
```

Let's first focus on the **properties** themselves!

Enum properties

```
namespace css
{
  enum class align_content : std::uint8_t
  {
    stretch,
    center,
    flex_start,
    flex_end,
    space_between,
    space_around
  };
}
```

Symbol tables for enums

```
namespace css::grammar
{
  struct align_content_table : x3::symbols<css::align_content>
  {
    align_content_table()
    {
      using css::align_content;

      add("stretch",         align_content::stretch)
         ("center",          align_content::center)
         ("flex-start",      align_content::flex_start)
         ("flex-end",        align_content::flex_end)
         ("space-between",   align_content::space_between)
         ("space-around",    align_content::space_around)
         ;
    }
  } const align_content;
}
```

The full property declaration

What about the rest?

```
align-content: center;
```

The full property declaration

What about the rest?

```
align-content: center;
```

We could use

```
"align-content" >> ':' >> grammar::align_content >> ';'
```

The full property declaration

What about the rest?

```
align-content: center;
```

We could use

```
"align-content" >> ':' >> grammar::align_content >> ';'
```

This means

- lots of repetition for all properties.
- has an attribute of type `css::align_content`
- how can we assign a `declaration_block` member?

Custom parsers to the rescue!

Assigning a member with an attribute:

```
template<typename PropertyType,
         typename ParserType,
         typename PointerToMemberType>
auto make_property(const PropertyType& property,
                   const ParserType& parser,
                   PointerToMemberType member)
{
  const auto setter = [member](auto& context)
  {
    _val(context).*member = std::move(_attr(context));
  };
  return lexeme[property] >> ':' >> parser[setter] >> ';';
}
```

Property declaration

```
const auto declaration
    = x3::rule<struct _,
               css::declaration_block>{"declaration"}
   %= make_property("align-content",
                    grammar::align_content,
                    &declaration_block::align_content)
    | make_property("align-self",
                    grammar::align_self,
                    &declaration_block::align_self)
    // ...
    ;
```

CSS Color

```
#ff0000
#f00
rgb(255, 0, 0)
rgba(255, 0, 0, 1)
red
hsl(0, 100%, 50%)
hsla(0, 100%, 0.5, 1)
```

CSS Color

```
#ff0000
#f00
rgb(255, 0, 0)
rgba(255, 0, 0, 1)
red
hsl(0, 100%, 50%)
hsla(0, 100%, 0.5, 1)
```

```
color = rule<struct color_, css::color>{"color"}
      = named_color
      | lexeme['#' >> ( color_hex_alpha
                      | color_hex
                      | color_short_hex_alpha
                      | color_short_hex
                      )]
      | color_rgb
      | color_rgba
      | color_hsl
      | color_hsla
      ;
```

CSS Color: the details

```
struct color
{
  std::uint8_t red, green, blue, alpha;
}
```

`named_color`: symbol table as for enums

CSS Color: the details

```
struct color
{
  std::uint8_t red, green, blue, alpha;
}
```

`named_color`: symbol table as for enums

For a hexadecimal color, we adapt the struct using **Boost.Fusion**:

```
BOOST_FUSION_ADAPT_STRUCT(skui::css::color,
                          red, green, blue, alpha)
```

CSS Color: the details

```
struct color
{
  std::uint8_t red, green, blue, alpha;
}
```

`named_color`: symbol table as for enums

For a hexadecimal color, we adapt the struct using **Boost.Fusion**:

```
BOOST_FUSION_ADAPT_STRUCT(skui::css::color,
                          red, green, blue, alpha)
```

```
constexpr auto uint8_hex = x3::uint_parser<std::uint8_t, 16, 2, 2>{};
const auto hex_color_alpha = rule<struct hca, css::color>{"hex alpha color"}
                           = uint8_hex >> uint8_hex >> uint8_hex >> uint8_hex;
const auto hex_color = rule<struct hex_color, css::color>{"hex color"}
                     = uint8_hex >> uint8_hex >> uint8_hex >> attr(255);
```

CSS color: the details

What about `#f00`?

CSS color: the details

What about `#f00`?

```
const auto single_hex_digit = uint_parser<std::uint8_t, 16, 1, 1>{};
const auto shorthand_hex
    = rule<struct shorthand_hex, std::uint8_t>{"shorthand hex digit"}
   %= single_hex_digit[multiply{std::uint8_t{17}}];
```

CSS color: the details

What about `#f00`?

```
const auto single_hex_digit = uint_parser<std::uint8_t, 16, 1, 1>{};
const auto shorthand_hex
    = rule<struct shorthand_hex, std::uint8_t>{"shorthand hex digit"}
   %= single_hex_digit[multiply{std::uint8_t{17}}];
```

```
template<typename ValueType>
struct multiply
{
  constexpr multiply(ValueType value) : factor{value} {}

  template<typename ContextType>
  void operator()(ContextType& context) const
  {
    using attribute_type = std::remove_reference_t<decltype(_attr(context))>;
    _attr(context) = _attr(context) * factor;
  }

private:
  const ValueType factor;
};
```

Moving on...

I started this thinking: easy syntax, quick parser.

Moving on...

I started this thinking: easy syntax, quick parser.

I dove head-first in CSS

Moving on...

I started this thinking: easy syntax, quick parser.

I dove head-first in CSS

And Boost Spirit X3

Moving on...

I started this thinking: easy syntax, quick parser.

I dove head-first in CSS

And Boost Spirit X3

... at the same time.

Moving on...

I started this thinking: easy syntax, quick parser.

I dove head-first in CSS

And Boost Spirit X3

... at the same time.

It's far from complete, but taking shape quite fast.

The end

You can find all the code discussed above here:

`https://github.com/skui-org/skui/tree/master/css`

Any questions?