Parsing CSS with Boost.Spirit.X3

Ruben Van Boxem
February 4th 2019
Introduction
About me

Elektronvortexbundels: een diepgaande theoretische studie

Promotoren
Karel D. de Verdiere
Prof. El. S.H. Bendsen

Ruben Van Boxem

Faculteit Wetenschappen
Department Fysica
Antwerpen 2015

https://github.com/rubenvb
Modern C++ next UI framework:

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

All help is welcome:
www.github.com/skui-org/skui
Modern C++ next UI framework:

- signal/slot
- friendly API
- fast
- platform integration
Modern C++ next UI framework:

- signal/slot
- friendly API
- fast
- platform integration
Modern C++ next UI framework:

- signal/slot
- friendly API
- fast
- platform integration
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
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 , ";" ;
graham = { 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.
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
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
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

- Former Spirit V1.8.x
  - boost::spirit::classic
- Parser library
  - boost::spirit::qi
- Lexer Library
  - boost::spirit::lex
- Generator Library
  - boost::spirit::karma

Classic

Qi

Lex

Karma
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:

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

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

Parser matched if `success == true`.

Complete match if also `first == last`. 
```cpp
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.
";
    else if (first != last)
        std::cerr << "parsing did not match full input.\n";
    else
        std::cout << "result: " << result << '\n';
}
```
Example parser:

```
parser expression     →  attribute type
```

In case of no attribute: `x3::unused_type`
Little bit of explanation

Example parser:

\[
pars\text{e}\ \text{expression} \rightarrow\ \text{attribute}\ \text{type}
\]

In case of no attribute: \text{x3::unused\_type}

Example input:

\[
3.14 \rightarrow 3.14
\]

Spirit is greedy, so partial matches are often possible!
Warming up

\[
\text{double}_- \rightarrow \text{double}
\]
Warming up

```
double_ → double
```

Matches:

```
3.14 → 3.14
1e-10 → .0000000001
-5 → -5.
```
# Warming up

**double_** → **double**

<table>
<thead>
<tr>
<th>Matches:</th>
<th>Does not match:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14 → 3.14</td>
<td>a string</td>
</tr>
<tr>
<td>1e−10 → .0000000001</td>
<td>0xFF → 0.</td>
</tr>
<tr>
<td>−5 → −5.</td>
<td>mY_S1lly_P@sSw0rD</td>
</tr>
</tbody>
</table>
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_

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14</td>
<td>3.14</td>
</tr>
<tr>
<td>1e-10</td>
<td>.0000000001</td>
</tr>
<tr>
<td>-5</td>
<td>-5.</td>
</tr>
</tbody>
</table>

### int_

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>-3</td>
<td>-3</td>
</tr>
</tbody>
</table>

### Does not match:

- a string
- 0xFF → 0.
- MY_Silly_P@Sw0rD

---

Ruben Van Boxem  
Parsing CSS with Boost.Spirit.X3  
February 4th 2019
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:

<table>
<thead>
<tr>
<th>float_</th>
<th>double_</th>
<th>long_double</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>oct</td>
<td>hex</td>
</tr>
<tr>
<td>ushort_</td>
<td>ulong_</td>
<td>uint_</td>
</tr>
<tr>
<td>short_</td>
<td>long_</td>
<td>int_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long_long</td>
</tr>
</tbody>
</table>

Literal forms for all of the above except bin, oct, and hex:
# Numeric parsers

## Number parsers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td><code>float_</code></td>
</tr>
<tr>
<td>double</td>
<td><code>double_</code></td>
</tr>
<tr>
<td>long</td>
<td><code>long_</code></td>
</tr>
<tr>
<td>long double</td>
<td><code>long_double</code></td>
</tr>
<tr>
<td>bin</td>
<td><code>bin</code></td>
</tr>
<tr>
<td>oct</td>
<td><code>oct</code></td>
</tr>
<tr>
<td>hex</td>
<td><code>hex</code></td>
</tr>
<tr>
<td>ushort</td>
<td><code>ushort_</code></td>
</tr>
<tr>
<td>ulong</td>
<td><code>ulong_</code></td>
</tr>
<tr>
<td>uint</td>
<td><code>uint_</code></td>
</tr>
<tr>
<td>ulong long</td>
<td><code>ulong_long</code></td>
</tr>
<tr>
<td>short</td>
<td><code>short_</code></td>
</tr>
<tr>
<td>long</td>
<td><code>long_</code></td>
</tr>
<tr>
<td>int</td>
<td><code>int_</code></td>
</tr>
<tr>
<td>long long</td>
<td><code>long_long</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td><code>float_(3.14 f)</code></td>
<td>unused_type</td>
</tr>
<tr>
<td>ushort</td>
<td><code>ushort_(42)</code></td>
<td>unused_type</td>
</tr>
<tr>
<td>int</td>
<td><code>int_(365)</code></td>
<td>unused_type</td>
</tr>
</tbody>
</table>
### Numeric parsers

#### Number parsers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float_</td>
</tr>
<tr>
<td>double</td>
<td>double_</td>
</tr>
<tr>
<td>long_double</td>
<td>long_double</td>
</tr>
<tr>
<td>bin</td>
<td>bin</td>
</tr>
<tr>
<td>oct</td>
<td>oct</td>
</tr>
<tr>
<td>hex</td>
<td>hex</td>
</tr>
<tr>
<td>ushort</td>
<td>ushort_</td>
</tr>
<tr>
<td>ulong</td>
<td>ulong_</td>
</tr>
<tr>
<td>uint</td>
<td>uint_</td>
</tr>
<tr>
<td>ulong_long</td>
<td>ulong_long</td>
</tr>
<tr>
<td>short</td>
<td>short_</td>
</tr>
<tr>
<td>long</td>
<td>long_</td>
</tr>
<tr>
<td>int</td>
<td>int_</td>
</tr>
<tr>
<td>long_long</td>
<td>long_long</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float_(3.14f)</td>
<td>unused_type</td>
</tr>
<tr>
<td>ushort</td>
<td>ushort_(42)</td>
<td>unused_type</td>
</tr>
<tr>
<td>int</td>
<td>int_(365)</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

#### 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...

<table>
<thead>
<tr>
<th>&quot;a string&quot;</th>
<th>→ unused_type</th>
</tr>
</thead>
</table>

Matches

<table>
<thead>
<tr>
<th>a string</th>
<th>→ unused_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>alnum</td>
<td>→ char</td>
</tr>
</tbody>
</table>

Does not match:

<table>
<thead>
<tr>
<th>not a string</th>
</tr>
</thead>
</table>
Warming up...

"a string" → unused_type

Matches

\[
\begin{align*}
\text{a string} & \rightarrow \text{unused_type} \\
\text{alnum} & \rightarrow \text{char}
\end{align*}
\]

Does not match:

\[
\begin{align*}
\text{not a string}
\end{align*}
\]

Matches:

\[
\begin{align*}
a & \rightarrow 'a' \\
T & \rightarrow 'T' \\
R & \rightarrow 'R' \\
Q & \rightarrow 'Q'
\end{align*}
\]
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:

<table>
<thead>
<tr>
<th>Literal</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>unused_type</td>
</tr>
<tr>
<td>lit ('a')</td>
<td>unused_type</td>
</tr>
<tr>
<td>char_ ('a')</td>
<td>unused_type</td>
</tr>
<tr>
<td>char_ (&quot;abc&quot;)</td>
<td>unused_type</td>
</tr>
<tr>
<td>char_ ('a', 'z')</td>
<td>unused_type</td>
</tr>
<tr>
<td>char_ (charset)</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alnum</td>
<td>alphanumeric</td>
</tr>
<tr>
<td>alpha</td>
<td>alphabetic</td>
</tr>
<tr>
<td>blank</td>
<td>blank</td>
</tr>
<tr>
<td>ctrl</td>
<td>control</td>
</tr>
<tr>
<td>digit</td>
<td>digit</td>
</tr>
<tr>
<td>graph</td>
<td>graphic</td>
</tr>
<tr>
<td>print</td>
<td>printable</td>
</tr>
<tr>
<td>punct</td>
<td>punctuation</td>
</tr>
<tr>
<td>space</td>
<td>space</td>
</tr>
<tr>
<td>xdigit</td>
<td>x-digit</td>
</tr>
<tr>
<td>lower</td>
<td>lowercase</td>
</tr>
<tr>
<td>upper</td>
<td>uppercase</td>
</tr>
</tbody>
</table>
## Parser operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused</td>
<td>fails if a matches</td>
</tr>
<tr>
<td>&amp;a</td>
<td>unused</td>
<td>matches if a matches</td>
</tr>
<tr>
<td>-a</td>
<td>optional&lt;A&gt;</td>
<td>match a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector&lt;A&gt;</td>
<td>match a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector&lt;A&gt;</td>
<td>match a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant&lt;A, B&gt;</td>
</tr>
<tr>
<td>a % b</td>
<td>vector&lt;A&gt;</td>
<td>list of a delimited by b</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>tuple&lt;A, B&gt;</td>
<td>sequence: a followed by b</td>
</tr>
</tbody>
</table>
## Parser operators

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

```
float_ >> int_   \rightarrow  tuple<float, int>
```

### Matches:

- `3.14 5` \(\rightarrow\) `{3.14f, 5}`
- `5 42` \(\rightarrow\) `{5.f, 42}`

### Does not match:

- `5 3.14` \(\rightarrow\) `{5.f, 3}`
- `3.1 abc`
## Parser operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused</td>
<td>fails if <code>a</code> matches</td>
</tr>
<tr>
<td>&amp;a</td>
<td>unused</td>
<td>matches if <code>a</code> matches</td>
</tr>
<tr>
<td>-a</td>
<td>optional(&lt;A&gt;)</td>
<td>match a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant(&lt;A, B&gt;)</td>
</tr>
<tr>
<td>a % b</td>
<td>vector(&lt;A&gt;)</td>
<td>list of <code>a</code> delimited by <code>b</code></td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>tuple(&lt;A, B&gt;)</td>
<td>sequence: <code>a</code> followed by <code>b</code></td>
</tr>
</tbody>
</table>

\[
\text{float}_- | \text{int} \rightarrow \text{variant} <\text{float}, \text{int}>
\]

**Matches:**

| 3.14       | \rightarrow 3.14f |
| 5          | \rightarrow 5     |

**Does not match:**

| abc        | \rightarrow 3.14f |
| 3.14 5     | \rightarrow 3.14f |
## Parser operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused</td>
<td>fails if a matches</td>
</tr>
<tr>
<td>&amp;a</td>
<td>unused</td>
<td>matches if a matches</td>
</tr>
<tr>
<td>-a</td>
<td>optional&lt;A&gt;</td>
<td>match a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector&lt;A&gt;</td>
<td>match a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector&lt;A&gt;</td>
<td>match a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant&lt;A, B&gt;</td>
</tr>
<tr>
<td>a</td>
<td>% b</td>
<td>vector&lt;A&gt;</td>
</tr>
<tr>
<td>a</td>
<td>&gt;&gt; b</td>
<td>tuple&lt;A, B&gt;</td>
</tr>
</tbody>
</table>

### Matches:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>! f l o a t _</td>
<td>&gt;&gt; i n t _</td>
<td>→</td>
</tr>
</tbody>
</table>

### Examples:

- **Matches:**
  - `a 5 → 5`
  - `* 42 → 42`

- **Does not match:**
  - `3.14 5 → 3`
## Parser operators

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

\& float_ >> int_ \rightarrow int

### Matches:

$3.14 \ 5 \rightarrow 5$

$1e-10 \ 20 \rightarrow 20$

### Does not match:

$5$

$3.14 \ abc$
### Parser operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused</td>
<td>fails if a matches</td>
</tr>
<tr>
<td>&amp;a</td>
<td>unused</td>
<td>matches if a matches</td>
</tr>
<tr>
<td>-a</td>
<td>optional(&lt;A&gt;)</td>
<td>match a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant(&lt;A, B&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>% b</td>
<td>vector(&lt;A&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>&gt;&gt; b</td>
<td>tuple(&lt;A, B&gt;)</td>
</tr>
</tbody>
</table>

\[
\text{float}_{} >> \text{int}_{} \rightarrow \text{tuple}<\text{float}, \text{optional}<\text{int}>\]

**Matches:**

\[
\begin{align*}
3.14 & \ 5 \rightarrow \{3.14 \ f, \ 5\} \\
3.14 & \rightarrow \{3.14 \ f, \ \{}\} \\
\end{align*}
\]

**Does not match:**

\[
\begin{align*}
5 \ 3.14 & \rightarrow \{5. \ f, \ 3\} \\
3.14 \ a & \rightarrow \{3.14 \ f, \ \{}\} \\
\end{align*}
\]
### Parser operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused</td>
<td>fails if <code>a</code> matches</td>
</tr>
<tr>
<td>&amp;a</td>
<td>unused</td>
<td>matches if <code>a</code> matches</td>
</tr>
<tr>
<td>-a</td>
<td>optional(&lt;A&gt;)</td>
<td>match a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant(&lt;A, B&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>% b</td>
<td>vector(&lt;A&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>&gt;&gt; b</td>
<td>tuple(&lt;A, B&gt;)</td>
</tr>
</tbody>
</table>

\[ *\text{float}_\] \rightarrow \text{vector}<\text{float}>  

**Matches:**

\[
\begin{align*}
3.14 & \rightarrow \{3.14f, 5.0f\} \\
3.14 & \rightarrow \{\} 
\end{align*}
\]

**Does not match:**

\[
\begin{align*}
3.14 \text{ a } 5 & \rightarrow \{3.14\} \\
a \ 5 \ 3.14 & \rightarrow \end{align*}
\]
## Parser operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused</td>
<td>fails if a matches</td>
</tr>
<tr>
<td>&amp;a</td>
<td>unused</td>
<td>matches if a matches</td>
</tr>
<tr>
<td>-a</td>
<td>optional(&lt;A&gt;)</td>
<td>match a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant(&lt;A, B&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>% b</td>
<td>vector(&lt;A&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>&gt;&gt; b</td>
<td>tuple(&lt;A, B&gt;)</td>
</tr>
</tbody>
</table>

\[ +\text{int} \rightarrow \text{vector}\(<\text{int}>\) \]

**Matches:**

\[
\begin{array}{c}
1 2 3 \rightarrow \{1, 2, 3\} \\
9 \rightarrow \{9\}
\end{array}
\]

**Does not match:**

\[
\begin{array}{c}
5 3.14 \rightarrow \{5, 3\}
\end{array}
\]
## Parser operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused</td>
<td>fails if <code>a</code> matches</td>
</tr>
<tr>
<td>&amp;a</td>
<td>unused</td>
<td>matches if <code>a</code> matches</td>
</tr>
<tr>
<td>-a</td>
<td>optional(&lt;A&gt;)</td>
<td>match a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector(&lt;A&gt;)</td>
<td>match a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant(&lt;A, B&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>% b</td>
<td>vector(&lt;A&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>&gt;&gt; b</td>
<td>tuple(&lt;A, B&gt;)</td>
</tr>
</tbody>
</table>

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

Matches:  

```
−3, 3  →  {−3, 3}
1       →  {1}
```

Does not match:  

```
1 2 3    →  {1}
```
**String parsers**

"literal string"

```cpp
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

```cpp
lit("number") >> ':' >> double
```
Symbol table

Symbol table:

```cpp
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:

```
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:

```
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:

<table>
<thead>
<tr>
<th>digit</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>2</td>
</tr>
<tr>
<td>nine</td>
<td>9</td>
</tr>
</tbody>
</table>

Does not match:

- forty-two
- one hundred → 1
- millions
Symbol table

Symbol table:

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

Use as

\[
\text{digits } \rightarrow \text{ digit}
\]

<table>
<thead>
<tr>
<th>Matches</th>
<th>Does not match:</th>
</tr>
</thead>
<tbody>
<tr>
<td>one → 1</td>
<td>forty-two</td>
</tr>
<tr>
<td>two → 2</td>
<td>one hundred → 1</td>
</tr>
<tr>
<td>nine → 9</td>
<td>millions</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eol</code></td>
<td>unused</td>
<td>matches end of line</td>
</tr>
<tr>
<td><code>oei</code></td>
<td>unused</td>
<td>matches end of input</td>
</tr>
<tr>
<td><code>eps</code></td>
<td>unused</td>
<td>matches empty string <em>(i.e. always)</em></td>
</tr>
<tr>
<td><code>eps(b)</code></td>
<td>unused</td>
<td>matches empty string if <code>b</code> == <code>true</code></td>
</tr>
<tr>
<td><code>attr(v)</code></td>
<td><code>decltype(v)</code></td>
<td>consumes no input, produces <code>v</code> as attribute</td>
</tr>
</tbody>
</table>
## Parser directives

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexeme[a]</td>
<td>A</td>
<td>disable skipper, pre-skip</td>
</tr>
<tr>
<td>no_skip[a]</td>
<td>A</td>
<td>disable skipper</td>
</tr>
<tr>
<td>no_case[a]</td>
<td>A</td>
<td>inhibit case-sensitivity</td>
</tr>
<tr>
<td>omit[a]</td>
<td>unused</td>
<td>ignore attribute of a</td>
</tr>
<tr>
<td>matches[a]</td>
<td>bool</td>
<td>return true if a matches</td>
</tr>
<tr>
<td>raw[a]</td>
<td>iterator_range</td>
<td>returns iterator range</td>
</tr>
<tr>
<td>expect[a]</td>
<td>A</td>
<td>throws exception if a does not match</td>
</tr>
<tr>
<td>repeat[a]</td>
<td>vector&lt;A&gt;</td>
<td>repeat a zero or more times</td>
</tr>
<tr>
<td>skip[a]</td>
<td>A</td>
<td>force skipping in lexeme or no_skip</td>
</tr>
</tbody>
</table>
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:

\[\text{parser[action]}\]

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:

```cpp
parser[action]
```

This is useful for *e.g.*

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

Required prototypes:

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_val</code></td>
<td>A reference to the attribute of the innermost rule invoking the parser</td>
</tr>
<tr>
<td><code>_where</code></td>
<td>Iterator range to the input stream</td>
</tr>
<tr>
<td><code>_attr</code></td>
<td>A reference to the attribute of the parser</td>
</tr>
<tr>
<td><code>_pass</code></td>
<td>A reference to a <code>bool</code> flag that can be used to force the parser to fail</td>
</tr>
</tbody>
</table>
# Semantic action context

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_val</code></td>
<td>A reference to the attribute of the innermost rule invoking the parser</td>
</tr>
<tr>
<td><code>_where</code></td>
<td>Iterator range to the input stream</td>
</tr>
<tr>
<td><code>_attr</code></td>
<td>A reference to the attribute of the parser</td>
</tr>
<tr>
<td><code>_pass</code></td>
<td>A reference to a <code>bool</code> flag that can be used to force the parser to fail</td>
</tr>
</tbody>
</table>

Example:

Divide rule value by 100:

```cpp
_val(context) /= 100.;
```
# Semantic action context

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_val</code></td>
<td>A reference to the attribute of the innermost rule invoking the parser</td>
</tr>
<tr>
<td><code>_where</code></td>
<td>Iterator range to the input stream</td>
</tr>
<tr>
<td><code>_attr</code></td>
<td>A reference to the attribute of the parser</td>
</tr>
<tr>
<td><code>_pass</code></td>
<td>A reference to a <code>bool</code> flag that can be used to force the parser to fail</td>
</tr>
</tbody>
</table>

Example:

Increment the rule’s value with the parsed attribute:

```cpp
val(ctx) += attr(ctx);
```
### Semantic action context

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_val</code></td>
<td>A reference to the attribute of the innermost rule invoking the parser</td>
</tr>
<tr>
<td><code>_where</code></td>
<td>Iterator range to the input stream</td>
</tr>
<tr>
<td><code>_attr</code></td>
<td>A reference to the attribute of the parser</td>
</tr>
<tr>
<td><code>_pass</code></td>
<td>A reference to a <code>bool</code> flag that can be used to force the parser to fail</td>
</tr>
</tbody>
</table>

**Example:**

Make the parser fail:

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

    int value{0};
    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 successful: " << value << '\n';
        else
            std::cout << "incomplete parse: " << value << '\n';
    }
    else
        std::cout << "parse unsuccessful\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:

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

Defining a rule:

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

Defining a rule:

```cpp
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 `\%=`:

```cpp
\%= /* actual parser */;
```
Rule syntax

Defining a rule:

```cpp
class 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 `%=`

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

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

```cpp
BOOST_SPIRIT_DECLARE
BOOST_SPIRIT_DEFINE
BOOST_SPIRIT_INSTANTIATE
```
CSS
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

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

More formally:

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

Simple enough, right?
Some details previously omitted for brevity:

- the format of `value` depends on which property

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

- properties such as `background-color` are more specific than `background`:
  ```css
  background: aquamarine url("faded.png")
  no-repeat fixed center;
  ```
Some details previously omitted for brevity:

- the format of `value` depends on which property

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

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

```css
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

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

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

```css
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
Basic CSS syntax:

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

```cpp
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:

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

```cpp
const auto declaration_block
    = x3::rule<struct declaration_block , css::declaration_block>
    { "declaration_block" }
    = lit ( '{' ) >> +declaration >> '}' ;
```
Let's ignore the complexity of selectors for now and assume we have a great selector parser:

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

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

```cpp
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`:

```cpp
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`:

```cpp
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; }
```

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

```cpp
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!
namespace css
{
    enum class align_content : std::uint8_t
    {
        stretch,
        center,
        flex_start,
        flex_end,
        space_between,
        space_around
    }
}
Symbol tables for enums

```cpp
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?

```css
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?
Assigning a member with an attribute:

```cpp
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] >> ';';
}
```
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

```css
#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

```cpp
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**:

```cpp
BOOST_FUSION_ADAPT_STRUCT( skui :: css :: color ,
    red , green , blue , alpha )
```
CSS Color: the details

```cpp
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**:

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

```cpp
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?

```cpp
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?

```cpp
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}}];
```

```cpp
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;
};
```
I started this thinking: easy syntax, quick parser.
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.
You can find all the code discussed above here:

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

Any questions?