# Writing Regexps 2021-22 / Regex Parser

Generated by Doxygen 1.9.3

# Chapter 1

# Namespace Index

## 1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

# Chapter 2

# Hierarchical Index

## 2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

# Chapter 3

# Class Index

## 3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 4

# File Index

## 4.1 File List

Here is a list of all files with brief descriptions:

# Chapter 5

# Namespace Documentation

## 5.1 wr22 Namespace Reference

### Namespaces

- namespace regex_parser

## 5.2 wr22::regex_parser Namespace Reference

### Namespaces

- namespace parser
- namespace regex
- namespace span
- namespace utils

## 5.3 wr22::regex_parser::parser Namespace Reference

### Namespaces

- namespace errors

### Classes

- class Parser

  *A regex parser.*

### Functions

- template<typename Iter , typename Sentinel >
  Parser (Iter begin, Sentinel end) -> Parser< Iter, Sentinel >

  *The type deduction guideline for `Parser`.*
- regex::SpannedPart parse_regex (const std::u32string_view &regex)

  *Parse a regular expression into its AST.*

### 5.3.1 Function Documentation

#### 5.3.1.1 parse_regex()

```
regex::SpannedPart wr22::regex_parser::parser::parse_regex (
            const std::u32string_view & regex )
```

Parse a regular expression into its AST.

The regular expression is a string view in the UTF-32 encoding. It is parsed and its object representation (see the docs for `regex::SpannedPart`) is built. The returned representation is an owned object and its lifetime does not depend on the lifetime of the `regex` argument.

If the parsing fails, an exception is thrown. `errors::ParseError` is the base class for all exceptions thrown from this function, but more specific exceptions may be caught and handled separately. See the docs for the `errors.hpp` file for details.

**Returns**

the parsed regex AST if the parsing succeeds.

**Exceptions**

| | |
|---|---|
| *errors::ParseError* | if the parsing fails. |

#### 5.3.1.2 Parser()

```
template<typename Iter , typename Sentinel >
wr22::regex_parser::parser::Parser (
            Iter begin,
            Sentinel end ) -> Parser< Iter, Sentinel >
```

The type deduction guideline for `Parser`.

## 5.4 wr22::regex_parser::parser::errors Namespace Reference

### Classes

- class ExpectedEnd

  *The error when the parser expected the input to end, but it did not.*
- struct ParseError

  *The base class for parse errors.*
- class UnexpectedChar

  *The error when the parser got a character it didn't expect at the current position.*
- class UnexpectedEnd

  *The error when the parser hit the end of the input earlier than it expected.*

## 5.5 wr22::regex_parser::regex Namespace Reference

### Namespaces

- namespace capture
- namespace part

  *The namespace with the variants of `Part`.*

### Classes

- class Capture

  *Group capture behavior.*
- class Part

  *A part of a regular expression and its AST node type.*
- class SpannedPart

  *A version of `Part` including the span information (position in the input) of the root AST node (child nodes always contain it because they are represented as `SpannedPart`s themselves).*

### Enumerations

- enum class NamedCaptureFlavor { Apostrophes , Angles , AnglesWithP }

  *The flavor (dialect) of a named group capture.*

### Functions

- std::ostream & operator<< (std::ostream &out, const Capture &capture)
- void to_json (nlohmann::json &j, const Capture &capture)
- std::ostream & operator<< (std::ostream &out, NamedCaptureFlavor flavor)
- void to_json (nlohmann::json &j, NamedCaptureFlavor flavor)
- std::ostream & operator<< (std::ostream &out, const SpannedPart &part)

  *Convert a `SpannedPart` to a textual representation and write it to an `std::ostream`.*
- void to_json (nlohmann::json &j, const Part &part)
- void to_json (nlohmann::json &j, const SpannedPart &part)

### 5.5.1 Enumeration Type Documentation

#### 5.5.1.1 NamedCaptureFlavor

```
enum class wr22::regex_parser::regex::NamedCaptureFlavor  [strong]
```

The flavor (dialect) of a named group capture.

The most common variants are included. This list is subject to extension if deemed necessary. The source used as a reference is https://www.regular-expressions.info/named.html.

**Enumerator**

| | |
|---|---|
| Apostrophes | The flavor `(?'name'contents)`. Mostly used in C# and other .NET-oriented languages, although can also be found in certain versions Perl, Boost and elsewhere. |
| Angles | The flavor `(?<name>contents)`. Mostly used in C# and other .NET-oriented languages, although can also be found in certain versions Perl, Boost and elsewhere. |
| AnglesWithP | The flavor `(?P<name>contents)`. Found in Python, PCRE and elsewhere. |

### 5.5.2 Function Documentation

#### 5.5.2.1 operator<<() [1/3]

```
std::ostream & wr22::regex_parser::regex::operator<< (
            std::ostream & out,
            const Capture & capture )
```

#### 5.5.2.2 operator<<() [2/3]

```
std::ostream & wr22::regex_parser::regex::operator<< (
            std::ostream & out,
            const SpannedPart & spanned_part )
```

Convert a `SpannedPart` to a textual representation and write it to an `std::ostream`.

#### 5.5.2.3 operator<<() [3/3]

```
std::ostream & wr22::regex_parser::regex::operator<< (
            std::ostream & out,
            NamedCaptureFlavor flavor )
```

#### 5.5.2.4 to_json() [1/4]

```
void wr22::regex_parser::regex::to_json (
            nlohmann::json & j,
            const Capture & capture )
```

#### 5.5.2.5 to_json() [2/4]

```
void wr22::regex_parser::regex::to_json (
            nlohmann::json & j,
            const Part & part )
```

#### 5.5.2.6 to_json() [3/4]

```
void wr22::regex_parser::regex::to_json (
            nlohmann::json & j,
            const SpannedPart & part )
```

#### 5.5.2.7 to_json() [4/4]

```
void wr22::regex_parser::regex::to_json (
            nlohmann::json & j,
            NamedCaptureFlavor flavor )
```

## 5.6 wr22::regex_parser::regex::capture Namespace Reference

### Classes

- struct Index

  *Denotes a group captured by index.*
- struct Name

  *Denotes a group captured by name.*
- struct None

  *Denotes an non-capturing group.*

### Typedefs

- using Adt = utils::Adt< None, Index, Name >

### Functions

- void to_json (nlohmann::json &j, const None &capture)
- void to_json (nlohmann::json &j, const Index &capture)
- void to_json (nlohmann::json &j, const Name &capture)

### 5.6.1 Typedef Documentation

#### 5.6.1.1 Adt

```
using wr22::regex_parser::regex::capture::Adt = typedef utils::Adt<None, Index, Name>
```

### 5.6.2 Function Documentation

#### 5.6.2.1 to_json() [1/3]

```
void wr22::regex_parser::regex::capture::to_json (
            nlohmann::json & j,
            const Index & capture )
```

#### 5.6.2.2 to_json() [2/3]

```
void wr22::regex_parser::regex::capture::to_json (
            nlohmann::json & j,
            const Name & capture )
```

#### 5.6.2.3 to_json() [3/3]

```
void wr22::regex_parser::regex::capture::to_json (
            nlohmann::json & j,
            const None & capture )
```

## 5.7 wr22::regex_parser::regex::part Namespace Reference

The namespace with the variants of `Part`.

### Classes

- struct Alternatives
  *A regex part with the list of alternatives to be matched.*
- struct Empty
  *An empty regex part.*
- struct Group
  *A regex part that represents a group in parentheses.*
- struct Literal
  *An regex part that matches a single character literally.*
- struct Optional
  *A regex part specifying an optional quantifier (`(expression)?`).*
- struct Plus
  *A regex part specifying an "at least one" quantifier (`(expression)+`).*
- struct Sequence
  *A regex part with the list of items to be matched one after another.*
- struct Star
  *A regex part specifying an "at least zero" quantifier (`(expression)*`).*
- struct Wildcard
  *A regex part specifying any single character (`.`).*

## Typedefs

- using Adt = utils::Adt< Empty, Literal, Alternatives, Sequence, Group, Optional, Plus, Star, Wildcard >

## Functions

- void to_json (nlohmann::json &j, const part::Empty &part)
- void to_json (nlohmann::json &j, const part::Literal &part)
- void to_json (nlohmann::json &j, const part::Alternatives &part)
- void to_json (nlohmann::json &j, const part::Sequence &part)
- void to_json (nlohmann::json &j, const part::Group &part)
- void to_json (nlohmann::json &j, const part::Optional &part)
- void to_json (nlohmann::json &j, const part::Plus &part)
- void to_json (nlohmann::json &j, const part::Star &part)
- void to_json (nlohmann::json &j, const part::Wildcard &part)

### 5.7.1 Detailed Description

The namespace with the variants of `Part`.

See the docs for the `Part` type for additional information.

### 5.7.2 Typedef Documentation

#### 5.7.2.1 Adt

```
using wr22::regex_parser::regex::part::Adt = typedef utils::  Adt<Empty, Literal, Alternatives,
Sequence, Group, Optional, Plus, Star, Wildcard>
```

### 5.7.3 Function Documentation

#### 5.7.3.1 to_json() [1/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Alternatives & part )
```

#### 5.7.3.2 to_json() [2/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Empty & part )
```

#### 5.7.3.3 to_json() [3/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Group & part )
```

#### 5.7.3.4 to_json() [4/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Literal & part )
```

#### 5.7.3.5 to_json() [5/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Optional & part )
```

#### 5.7.3.6 to_json() [6/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Plus & part )
```

#### 5.7.3.7 to_json() [7/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Sequence & part )
```

#### 5.7.3.8 to_json() [8/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Star & part )
```

#### 5.7.3.9 to_json() [9/9]

```
void wr22::regex_parser::regex::part::to_json (
            nlohmann::json & j,
            const part::Wildcard & part )
```

## 5.8 wr22::regex_parser::span Namespace Reference

### Classes

- struct InvalidSpan

  *The exception thrown on an attempt to construct an invalid span.*
- class Span

  *Character position range in the input string.*

### Functions

- std::ostream & operator<< (std::ostream &out, Span span)
- void to_json (nlohmann::json &j, Span span)

### 5.8.1 Function Documentation

#### 5.8.1.1 operator<<()

```
std::ostream & wr22::regex_parser::span::operator<< (
            std::ostream & out,
            Span span )
```

#### 5.8.1.2 to_json()

```
void wr22::regex_parser::span::to_json (
            nlohmann::json & j,
            Span span )
```

## 5.9 wr22::regex_parser::utils Namespace Reference

### Namespaces

- namespace detail

### Classes

- class Adt

  *A helper class that simplifies creation of algebraic data types.*
- class Box

  *A copyable and equality-comparable wrapper around `std::unique_ptr`.*
- struct BoxIsEmpty

### Functions

- template<typename... Variants>
  bool operator== (const Adt< Variants... > &lhs, const Adt< Variants... > &rhs)

  *Compare two compatible ADTs for equality.*
- template<typename... Variants>
  bool operator!= (const Adt< Variants... > &lhs, const Adt< Variants... > &rhs)

  *Compare two compatible ADTs for non-equality.*
- template<typename T >
  Box (T &&value) -> Box< T >

  *Type deduction guideline for `Box` (value initialization).*
- template<typename T >
  Box (std::unique_ptr< T > ptr) -> Box< T >

  *Type deduction guideline for `Box` (`std::unique_ptr` adoption).*
- template<typename T , typename U >
  bool operator== (const Box< T > &lhs, const Box< U > &rhs)
- template<typename T , typename U >
  bool operator!= (const Box< T > &lhs, const Box< U > &rhs)

### 5.9.1 Function Documentation

#### 5.9.1.1 Box() [1/2]

```
template<typename T >
wr22::regex_parser::utils::Box (
            std::unique_ptr< T > ptr ) -> Box< T >
```

Type deduction guideline for `Box` (`std::unique_ptr` adoption).

#### 5.9.1.2 Box() [2/2]

```
template<typename T >
wr22::regex_parser::utils::Box (
            T && value ) -> Box< T >
```

Type deduction guideline for Box (value initialization).

#### 5.9.1.3 operator"!=() [1/2]

```
template<typename... Variants>
bool wr22::regex_parser::utils::operator!= (
            const Adt< Variants... > & lhs,
            const Adt< Variants... > & rhs )
```

Compare two compatible ADTs for non-equality.

#### 5.9.1.4 operator"!=() [2/2]

```
template<typename T , typename U >
bool wr22::regex_parser::utils::operator!= (
            const Box< T > & lhs,
            const Box< U > & rhs )
```

#### 5.9.1.5 operator==() [1/2]

```
template<typename... Variants>
bool wr22::regex_parser::utils::operator== (
            const Adt< Variants... > & lhs,
            const Adt< Variants... > & rhs )
```

Compare two compatible ADTs for equality.

#### 5.9.1.6 operator==() [2/2]

```
template<typename T , typename U >
bool wr22::regex_parser::utils::operator== (
            const Box< T > & lhs,
            const Box< U > & rhs )
```

## 5.10 wr22::regex_parser::utils::detail Namespace Reference

### Namespaces

- namespace adt

## 5.11 wr22::regex_parser::utils::detail::adt Namespace Reference

### Classes

- struct MultiCallable

# Chapter 6

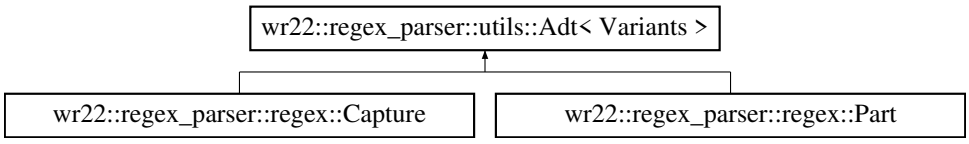
# Class Documentation

## 6.1 wr22::regex_parser::utils::Adt< Variants > Class Template Reference

A helper class that simplifies creation of algebraic data types.

`#include <adt.hpp>`

Inheritance diagram for wr22::regex_parser::utils::Adt< Variants >:




### Public Types

- using VariantType = std::variant< Variants... >

  *A convenience type alias for the concrete `std::variant` type used.*

### Public Member Functions

- template<typename V >
  Adt (V variant)

  *Constructor for each of the variants.*
- template<typename... Fs>
  decltype(auto) visit (Fs &&... visitors) const

  *Visit the ADT, applying the suitable function from the list of visitors on the variant held.*
- template<typename... Fs>
  decltype(auto) visit (Fs &&... visitors)

  *Visit the ADT, applying the suitable function from the list of visitors on the variant held.*
- const VariantType & as_variant () const

  *Access the underlying `std::variant` type (constant version).*
- VariantType & as_variant ()

  *Access the underlying `std::variant` type (non-constant version).*

### Protected Attributes

- VariantType m_variant

## 6.1.1 Detailed Description

**template<typename... Variants>**
**class wr22::regex_parser::utils::Adt< Variants >**

A helper class that simplifies creation of algebraic data types.

Algebraic data types are data types that can have one type of a predefined set of variants, but be stored and represented as values of one common type. In C++, `std::variant` serves exactly this purpose. It is, however, not very convenient to work with or build upon, so this class is designed to simplify building new algebraic data types. It still uses `std::variant` under the hood.

The template type parameters are the types that the variants may hold (must be distinct types).

## 6.1.2 Member Typedef Documentation

### 6.1.2.1 VariantType

```
template<typename...  Variants>
using wr22::regex_parser::utils::Adt< Variants >::VariantType = std::variant<Variants...>
```

A convenience type alias for the concrete `std::variant` type used.

## 6.1.3 Constructor & Destructor Documentation

### 6.1.3.1 Adt()

```
template<typename...  Variants>
template<typename V >
wr22::regex_parser::utils::Adt< Variants >::Adt (
            V variant )  [inline]
```

Constructor for each of the variants.

Construct an instance holding a specified variant. The type `V` of the variant provided must be one of the types from `Variants`. Note that this constructor is purposefully implicit, so that the variants as separate types are transparently converted to this common type when necessary.

The variant is taken by value and moved thereafter, so that, when constructing the common type, the variant may be either copied or moved, depending on the user's intentions.

## 6.1.4 Member Function Documentation

### 6.1.4.1 as_variant() [1/2]

```
template<typename... Variants>
VariantType & wr22::regex_parser::utils::Adt< Variants >::as_variant ( ) [inline]
```

Access the underlying `std::variant` type (non-constant version).

### 6.1.4.2 as_variant() [2/2]

```
template<typename... Variants>
const VariantType & wr22::regex_parser::utils::Adt< Variants >::as_variant ( ) const [inline]
```

Access the underlying `std::variant` type (constant version).

### 6.1.4.3 visit() [1/2]

```
template<typename... Variants>
template<typename... Fs>
decltype(auto) wr22::regex_parser::utils::Adt< Variants >::visit (
            Fs &&... visitors ) [inline]
```

Visit the ADT, applying the suitable function from the list of visitors on the variant held.

This is the non-constant version of the method. See the docs for the constant version for a detailed description and code examples. The only thing different in this version of the method is that the visitors get called with a non-const lvalue reference to the variants instead of a const reference.

### 6.1.4.4 visit() [2/2]

```
template<typename... Variants>
template<typename... Fs>
decltype(auto) wr22::regex_parser::utils::Adt< Variants >::visit (
            Fs &&... visitors ) const [inline]
```

Visit the ADT, applying the suitable function from the list of visitors on the variant held.

Using this method is essentially the same as using `std::visit` on the variant, except that, for convenience, multiple visitors are joined into one big visitor. That is, a typical `Adt` usage might look like this:

```
struct MyAdt : public Adt<int, double> {
    // Make the constructor available in the derived class.
    using Adt<int, double>::Adt;
};
// <...>
void func() {
    // Variant type: double.
    MyAdt my_adt = 3.14;
    // Prints "Double: 3.14".
    my_adt.visit(
        [](int x)    { std::cout « "Int: "    « x « std::endl; },
        [](double x) { std::cout « "Double: " « x « std::endl; }
    );
}
```

This is the constant version of the method. Visitors must be callable with the const reference to variant types.

### 6.1.5 Member Data Documentation

#### 6.1.5.1 m_variant

```
template<typename... Variants>
VariantType wr22::regex_parser::utils::Adt< Variants >::m_variant [protected]
```

The documentation for this class was generated from the following file:

- include/wr22/regex_parser/utils/adt.hpp

## 6.2 wr22::regex_parser::regex::part::Alternatives Struct Reference

A regex part with the list of alternatives to be matched.

`#include <part.hpp>`

### Public Member Functions

- Alternatives (std::vector< SpannedPart > alternatives)
- bool operator== (const Alternatives &rhs) const =default

### Public Attributes

- std::vector< SpannedPart > alternatives

  *The list of the alternatives.*

### Static Public Attributes

- static constexpr const char * code_name = "alternatives"

### 6.2.1 Detailed Description

A regex part with the list of alternatives to be matched.

Alternatives in regular expressions are subexpressions by |. For the whole expression part's match to succeed, at least one of the subexpressions must match the input successfully.

As an example, `a|(b)|cde` would be represented as an `Alternatives` part with 3 alternatives. The alternatives themselves are represented recursively as `SpannedPart`s.

### 6.2.2 Constructor & Destructor Documentation

#### 6.2.2.1 Alternatives()

```
wr22::regex_parser::regex::part::Alternatives::Alternatives (
            std::vector< SpannedPart > alternatives )  [explicit]
```

### 6.2.3 Member Function Documentation

#### 6.2.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Alternatives::operator== (
            const Alternatives & rhs ) const  [default]
```

### 6.2.4 Member Data Documentation

#### 6.2.4.1 alternatives

```
std::vector<SpannedPart> wr22::regex_parser::regex::part::Alternatives::alternatives
```

The list of the alternatives.

#### 6.2.4.2 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Alternatives::code_name = "alternatives"
[static], [constexpr]
```

The documentation for this struct was generated from the following files:

- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.3 wr22::regex_parser::utils::Box< T > Class Template Reference

A copyable and equality-comparable wrapper around `std::unique_ptr`.

```
#include <box.hpp>
```

## Public Member Functions

- Box (T &&value)

  *Constructor that places a value inside the wrapped* `std::unique_ptr`.
- Box (std::unique_ptr< T > ptr)

  *Constructor that adopts an existing* `std::unique_ptr`.
- template<typename Dummy = T>
  Box (const Box &other)

  *Copy constructor.*
- const T & operator* () const

  *Derefencing operator: obtain a const reference to the stored value.*
- T & operator* ()

  *Derefencing operator: obtain a reference to the stored value.*

## Static Public Member Functions

- template<typename... Args>
  static Box< T > construct_in_place (Args &&... args)

  *Construct a value on the heap in place.*

### 6.3.1 Detailed Description

**template<typename T>**
**class wr22::regex_parser::utils::Box< T >**

A copyable and equality-comparable wrapper around `std::unique_ptr`.

The behavior of this wrapper regarding copying and equality comparison are akin to that of Rust's `std::boxed::Box`, and hence the class's name. Namely, when testing for (in)equality, the wrapped values are compared instead of raw pointers, and, when wrapped values are copyable, copying a `Box` creates another `std::unique_ptr` with a copy of the wrapped value.

A `Box` usually contains a value. However, it may become empty when it is moved from. To ensure safety, most operations on an empty box will throw a `BoxIsEmpty` exception instead of causing undefined bahavior.

### 6.3.2 Constructor & Destructor Documentation

#### 6.3.2.1 Box() [1/3]

```
template<typename T >
wr22::regex_parser::utils::Box< T >::Box (
            T && value )  [inline], [explicit]
```

Constructor that places a value inside the wrapped `std::unique_ptr`.

Takes the value by a universal reference and, due to perfect forwarding, both copy and move initialization is possible.

#### 6.3.2.2 Box() [2/3]

```
template<typename T >
wr22::regex_parser::utils::Box< T >::Box (
            std::unique_ptr< T > ptr )  [inline], [explicit]
```

Constructor that adopts an existing `std::unique_ptr`.

Takes the `std::unique_ptr` by value, so the latter must be either passed directly as an rvalue or `std::move()`d into the argument. However, please note that if your code snippet looks like this:
`Box(std::make_unique<T>(args...))`

Then you should take a look at the `construct_in_place` method:
`Box<T>::construct_in_place(args...)`

#### 6.3.2.3 Box() [3/3]

```
template<typename T >
template<typename Dummy = T>
wr22::regex_parser::utils::Box< T >::Box (
            const Box< T > & other )  [inline]
```

Copy constructor.

Creates another `std::unique_ptr` with a copy of the currently wrapped value.

**Parameters**

| | |
|---|---|
| *`other`* | the `Box` from which to copy. |

**Exceptions**

| | |
|---|---|
| *BoxIsEmpty* | if `other` is empty. |

### 6.3.3 Member Function Documentation

#### 6.3.3.1 construct_in_place()

```
template<typename T >
template<typename... Args>
static Box< T > wr22::regex_parser::utils::Box< T >::construct_in_place (
            Args &&... args )  [inline], [static]
```

Construct a value on the heap in place.

Forwards the arguments to `std::make_unique` and wraps the resulting `std::unique_ptr`.

#### 6.3.3.2 operator*() [1/2]

```
template<typename T >
T & wr22::regex_parser::utils::Box< T >::operator* ( )  [inline]
```

Derefencing operator: obtain a reference to the stored value.

**Exceptions**

| | |
|---|---|
| *BoxIsEmpty* | if this Box does not contain a value at the moment. |

#### 6.3.3.3 operator*() [2/2]

```
template<typename T >
const T & wr22::regex_parser::utils::Box< T >::operator* ( ) const  [inline]
```

Derefencing operator: obtain a const reference to the stored value.

**Exceptions**

| | |
|---|---|
| *BoxIsEmpty* | if this Box does not contain a value at the moment. |

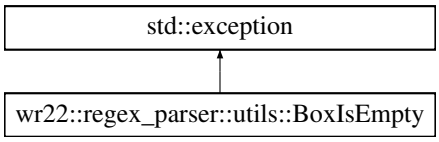
The documentation for this class was generated from the following file:

- include/wr22/regex_parser/utils/box.hpp

## 6.4 wr22::regex_parser::utils::BoxIsEmpty Struct Reference

```
#include <box.hpp>
```

Inheritance diagram for wr22::regex_parser::utils::BoxIsEmpty:




### Public Member Functions

- const char * what () const noexcept override

### 6.4.1 Member Function Documentation

#### 6.4.1.1 what()

```
const char * wr22::regex_parser::utils::BoxIsEmpty::what ( ) const  [override], [noexcept]
```

The documentation for this struct was generated from the following files:

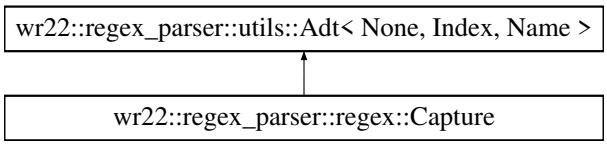
- include/wr22/regex_parser/utils/box.hpp
- src/utils/box.cpp

## 6.5 wr22::regex_parser::regex::Capture Class Reference

Group capture behavior.

```
#include <capture.hpp>
```

Inheritance diagram for wr22::regex_parser::regex::Capture:




### Additional Inherited Members

### 6.5.1 Detailed Description

Group capture behavior.

A group can be captured by index (when one writes `(contents)`), by name (e.g. `(?<name>contents)` in some dialects) or not captured at all (`(?:contents)`). Objects of this type determine how exactly a certain group is going to be captured. This is a variant type (see `Part` and `utils::Adt` for a more detailed explanation of the concept). The variants for this class (explicitly or implicitly convertible to this type) are located in the `capture` namespace.

The documentation for this class was generated from the following file:

- include/wr22/regex_parser/regex/capture.hpp

## 6.6 wr22::regex_parser::regex::part::Empty Struct Reference

An empty regex part.

```
#include <part.hpp>
```

## Public Member Functions

- Empty ()=default
- bool operator== (const Empty &rhs) const =default

## Static Public Attributes

- static constexpr const char * code_name = "empty"

### 6.6.1 Detailed Description

An empty regex part.

Corresponds to an empty regular expression (`""`) or the contents of an empty parenthesized group (`"()"`).

### 6.6.2 Constructor & Destructor Documentation

#### 6.6.2.1 Empty()

```
wr22::regex_parser::regex::part::Empty::Empty ( )  [explicit], [default]
```

### 6.6.3 Member Function Documentation

#### 6.6.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Empty::operator== (
            const Empty & rhs ) const  [default]
```

### 6.6.4 Member Data Documentation

#### 6.6.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Empty::code_name = "empty"  [static],
[constexpr]
```

The documentation for this struct was generated from the following file:

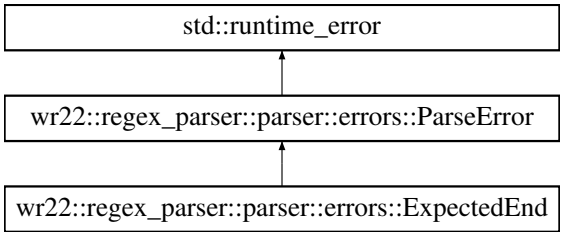
- include/wr22/regex_parser/regex/part.hpp

# 6.7 wr22::regex_parser::parser::errors::ExpectedEnd Class Reference

The error when the parser expected the input to end, but it did not.

`#include <errors.hpp>`

Inheritance diagram for wr22::regex_parser::parser::errors::ExpectedEnd:




## Public Member Functions

- ExpectedEnd (size_t position, char32_t char_got)

  *Constructor.*
- size_t position () const

  *Get the input position. See the constructor docs for a more detailed description.*
- char32_t char_got () const

  *Get the character the parser has received.*

## 6.7.1 Detailed Description

The error when the parser expected the input to end, but it did not.

## 6.7.2 Constructor & Destructor Documentation

### 6.7.2.1 ExpectedEnd()

```
wr22::regex_parser::parser::errors::ExpectedEnd::ExpectedEnd (
            size_t position,
            char32_t char_got )
```

Constructor.

**Parameters**

| | |
|---|---|
| *position* | the 0-based position in the input when the parser has encountered the end of input. |
| *char_got* | the character that the parser has received instead of the end of input. |

### 6.7.3 Member Function Documentation

#### 6.7.3.1 char_got()

```
char32_t wr22::regex_parser::parser::errors::ExpectedEnd::char_got ( ) const
```

Get the character the parser has received.

See the constructor docs for a more detailed description.

#### 6.7.3.2 position()

```
size_t wr22::regex_parser::parser::errors::ExpectedEnd::position ( ) const
```

Get the input position. See the constructor docs for a more detailed description.

The documentation for this class was generated from the following files:

- include/wr22/regex_parser/parser/errors.hpp
- src/parser/errors.cpp

## 6.8 wr22::regex_parser::regex::part::Group Struct Reference

A regex part that represents a group in parentheses.

```
#include <part.hpp>
```

### Public Member Functions

- Group (Capture capture, SpannedPart inner)
  *Convenience constructor.*
- bool operator== (const Group &rhs) const =default

### Public Attributes

- Capture capture
  *Capture behavior.*
- utils::Box< SpannedPart > inner
  *The smart pointer to the group contents.*

### Static Public Attributes

- static constexpr const char * code_name = "group"

### 6.8.1 Detailed Description

A regex part that represents a group in parentheses.

A group in regular expressions is virtually everything that is enclosed with parentheses: `(some group)`, `(?:blablabla)` and `(?P<group_name>group contents)` are all groups.

A group has two main attributes: (1) how it is captured during matching and (2) the contents of the group. The contents is simply another `SpannedPart`. The capture behavior is expressed by a separate type `Capture`. See its docs for additional info, and take a look at `https://www.regular-expressions.info/brackets.html` for an introduction to or a recap of regex groups and capturing.

### 6.8.2 Constructor & Destructor Documentation

#### 6.8.2.1 Group()

```
wr22::regex_parser::regex::part::Group::Group (
            Capture capture,
            SpannedPart inner )  [explicit]
```

Convenience constructor.

### 6.8.3 Member Function Documentation

#### 6.8.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Group::operator== (
            const Group & rhs ) const  [default]
```

### 6.8.4 Member Data Documentation

#### 6.8.4.1 capture

```
Capture wr22::regex_parser::regex::part::Group::capture
```

Capture behavior.

#### 6.8.4.2 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Group::code_name = "group"  [static],
[constexpr]
```

#### 6.8.4.3 inner

```
utils::Box<SpannedPart> wr22::regex_parser::regex::part::Group::inner
```

The smart pointer to the group contents.

The documentation for this struct was generated from the following files:

- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.9 wr22::regex_parser::regex::capture::Index Struct Reference

Denotes a group captured by index.

```
#include <capture.hpp>
```

### Public Member Functions

- Index ()=default
- bool operator== (const Index &rhs) const =default

### Static Public Attributes

- static constexpr const char * code_name = "index"

### 6.9.1 Detailed Description

Denotes a group captured by index.

### 6.9.2 Constructor & Destructor Documentation

#### 6.9.2.1 Index()

```
wr22::regex_parser::regex::capture::Index::Index ( )  [explicit], [default]
```

### 6.9.3 Member Function Documentation

#### 6.9.3.1 operator==()

```
bool wr22::regex_parser::regex::capture::Index::operator== (
            const Index & rhs ) const  [default]
```

### 6.9.4 Member Data Documentation

#### 6.9.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::capture::Index::code_name = "index"  [static],
[constexpr]
```

The documentation for this struct was generated from the following file:

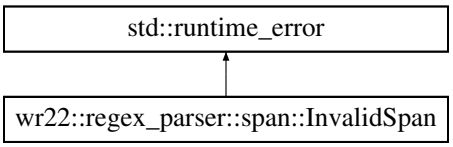
- include/wr22/regex_parser/regex/capture.hpp

## 6.10 wr22::regex_parser::span::InvalidSpan Struct Reference

The exception thrown on an attempt to construct an invalid span.

```
#include <span.hpp>
```

Inheritance diagram for wr22::regex_parser::span::InvalidSpan:




### Public Member Functions

- InvalidSpan (size_t begin, size_t end)

### Public Attributes

- size_t begin
- size_t end

### 6.10.1 Detailed Description

The exception thrown on an attempt to construct an invalid span.

See the documentation for Span for additional information.

### 6.10.2 Constructor & Destructor Documentation

#### 6.10.2.1 InvalidSpan()

```
wr22::regex_parser::span::InvalidSpan::InvalidSpan (
            size_t begin,
            size_t end )
```

### 6.10.3 Member Data Documentation

#### 6.10.3.1 begin

```
size_t wr22::regex_parser::span::InvalidSpan::begin
```

#### 6.10.3.2 end

```
size_t wr22::regex_parser::span::InvalidSpan::end
```

The documentation for this struct was generated from the following files:

- include/wr22/regex_parser/span/span.hpp
- src/span/span.cpp

## 6.11 wr22::regex_parser::regex::part::Literal Struct Reference

An regex part that matches a single character literally.

```
#include <part.hpp>
```

### Public Member Functions

- Literal (char32_t character)
- bool operator== (const Literal &rhs) const =default

## Public Attributes

- char32_t character

## Static Public Attributes

- static constexpr const char * code_name = "literal"

### 6.11.1 Detailed Description

An regex part that matches a single character literally.

Corresponds to a plain character in a regular expression. E.g. the regex "`foo`" contains three character literals: `f`, `o` and `o`.

### 6.11.2 Constructor & Destructor Documentation

#### 6.11.2.1 Literal()

```
wr22::regex_parser::regex::part::Literal::Literal (
            char32_t character )  [explicit]
```

### 6.11.3 Member Function Documentation

#### 6.11.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Literal::operator== (
            const Literal & rhs ) const  [default]
```

### 6.11.4 Member Data Documentation

#### 6.11.4.1 character

```
char32_t wr22::regex_parser::regex::part::Literal::character
```

#### 6.11.4.2 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Literal::code_name = "literal"  [static],
[constexpr]
```

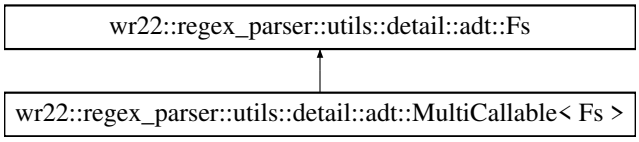
The documentation for this struct was generated from the following files:

- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.12 wr22::regex_parser::utils::detail::adt::MultiCallable< Fs > Struct Template Reference

```
#include <adt.hpp>
```

Inheritance diagram for wr22::regex_parser::utils::detail::adt::MultiCallable< Fs >:




### Public Member Functions

- MultiCallable (Fs &&... fs)

### 6.12.1 Constructor & Destructor Documentation

#### 6.12.1.1 MultiCallable()

```
template<typename...  Fs>
wr22::regex_parser::utils::detail::adt::MultiCallable< Fs >::MultiCallable (
            Fs &&...  fs )  [inline]
```

The documentation for this struct was generated from the following file:

- include/wr22/regex_parser/utils/adt.hpp

## 6.13 wr22::regex_parser::regex::capture::Name Struct Reference

Denotes a group captured by name.

```
#include <capture.hpp>
```

## Public Member Functions

- Name (std::string name, NamedCaptureFlavor flavor)
- bool operator== (const Name &rhs) const =default

## Public Attributes

- std::string name
- NamedCaptureFlavor flavor

## Static Public Attributes

- static constexpr const char * code_name = "name"

### 6.13.1 Detailed Description

Denotes a group captured by name.

A specific name and the syntax variant for this name's specification (see `NamedCaptureFlavor`) are stored.

### 6.13.2 Constructor & Destructor Documentation

#### 6.13.2.1 Name()

```
wr22::regex_parser::regex::capture::Name::Name (
            std::string name,
            NamedCaptureFlavor flavor )  [explicit]
```

### 6.13.3 Member Function Documentation

#### 6.13.3.1 operator==()

```
bool wr22::regex_parser::regex::capture::Name::operator== (
            const Name & rhs ) const  [default]
```

### 6.13.4 Member Data Documentation

#### 6.13.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::capture::Name::code_name = "name"  [static],
[constexpr]
```

#### 6.13.4.2 flavor

```
NamedCaptureFlavor wr22::regex_parser::regex::capture::Name::flavor
```

#### 6.13.4.3 name

```
std::string wr22::regex_parser::regex::capture::Name::name
```

The documentation for this struct was generated from the following files:

- include/wr22/regex_parser/regex/capture.hpp
- src/parser/capture.cpp

## 6.14 wr22::regex_parser::regex::capture::None Struct Reference

Denotes an non-capturing group.

```
#include <capture.hpp>
```

### Public Member Functions

- None ()=default
- bool operator== (const None &rhs) const =default

### Static Public Attributes

- static constexpr const char * code_name = "none"

### 6.14.1 Detailed Description

Denotes an non-capturing group.

### 6.14.2 Constructor & Destructor Documentation

#### 6.14.2.1 None()

```
wr22::regex_parser::regex::capture::None::None ( )  [explicit], [default]
```

### 6.14.3 Member Function Documentation

#### 6.14.3.1 operator==()

```
bool wr22::regex_parser::regex::capture::None::operator== (
            const None & rhs ) const  [default]
```

### 6.14.4 Member Data Documentation

#### 6.14.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::capture::None::code_name = "none"  [static],
[constexpr]
```

The documentation for this struct was generated from the following file:

- include/wr22/regex_parser/regex/capture.hpp

## 6.15 wr22::regex_parser::regex::part::Optional Struct Reference

A regex part specifying an optional quantifier ( `(expression)?`).

```
#include <part.hpp>
```

### Public Member Functions

- Optional (SpannedPart inner)
  *Convenience constructor.*
- bool operator== (const Optional &rhs) const =default

### Public Attributes

- utils::Box< SpannedPart > inner
  *The smart pointer to the subexpression under the quantifier.*

## Static Public Attributes

- static constexpr const char * code_name = "optional"

### 6.15.1 Detailed Description

A regex part specifying an optional quantifier ( `(expression)?`).

### 6.15.2 Constructor & Destructor Documentation

#### 6.15.2.1 Optional()

```
wr22::regex_parser::regex::part::Optional::Optional (
            SpannedPart inner )  [explicit]
```

Convenience constructor.

### 6.15.3 Member Function Documentation

#### 6.15.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Optional::operator== (
            const Optional & rhs ) const  [default]
```

### 6.15.4 Member Data Documentation

#### 6.15.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Optional::code_name = "optional"  [static],
[constexpr]
```

#### 6.15.4.2 inner

```
utils::Box<SpannedPart> wr22::regex_parser::regex::part::Optional::inner
```

The smart pointer to the subexpression under the quantifier.

The documentation for this struct was generated from the following files:

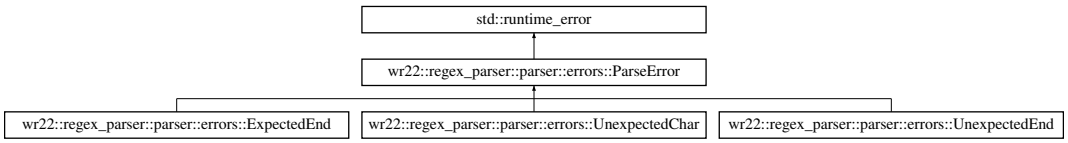
- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.16 wr22::regex_parser::parser::errors::ParseError Struct Reference

The base class for parse errors.

```
#include <errors.hpp>
```

Inheritance diagram for wr22::regex_parser::parser::errors::ParseError:




### 6.16.1 Detailed Description

The base class for parse errors.

This exception type should be caught if it is desired to catch all parse errors. However, there are more specific exceptions deriving from this one that can be handled separately for greater flexibility.

The documentation for this struct was generated from the following file:

- include/wr22/regex_parser/parser/errors.hpp

## 6.17 wr22::regex_parser::parser::Parser< Iter, Sentinel > Class Template Reference

A regex parser.

## Public Member Functions

- Parser (Iter begin, Sentinel end)
  *Constructor.*
- void expect_end ()
  *Ensure that the parser has consumed all of the input.*
- regex::SpannedPart parse_regex ()
  *Parse a regex consuming part of the remaining input.*
- regex::SpannedPart parse_alternatives ()
  *Intermediate rule: parse a pipe-separated list of alternatives (e.g.*
- regex::SpannedPart parse_sequence ()
  *Intermediate rule: parse a sequence of atoms (e.g.*
- regex::SpannedPart parse_sequence_or_empty ()
  *Intermediate rule: parse a possibly empty sequence of atoms.*
- regex::SpannedPart parse_atom ()
  *Intermediate rule: parse an atom.*
- regex::SpannedPart parse_wildcard ()
  *Intermediate rule: parse a wildcard (`.`*
- regex::SpannedPart parse_char_literal ()
  *Intermediate rule: parse a character literal.*
- regex::SpannedPart parse_group ()
  *Intermediate rule: parse a parenthesized group (any capture variant).*
- std::pair< std::string, Span > parse_group_name ()
  *Intermediate rule: parse a group name.*

### 6.17.1 Detailed Description

**template<typename Iter, typename Sentinel>**
**requires requires(Iter iter, Sentinel end) { ++iter; { *iter } -> std::convertible_to<char32_t>; { iter == end } -> std::convertible_↩ to<bool>; { iter != end } -> std::convertible_to<bool>; }**
**class wr22::regex_parser::parser::Parser< Iter, Sentinel >**

A regex parser.

For additional information see the methods' docs, particularly the constructor and the `parse_regex` method.

### 6.17.2 Constructor & Destructor Documentation

#### 6.17.2.1 Parser()

```
template<typename Iter , typename Sentinel >
wr22::regex_parser::parser::Parser< Iter, Sentinel >::Parser (
            Iter begin,
            Sentinel end )  [inline]
```

Constructor.

This constructor stores a pair of forward iterators that should generate a sequence of Unicode code points (`char32_t`). The `begin` iterator and the `end` sentinel may have different types provided that the iterator can is equality comparable with the sentinel.

SAFETY: The iterators must not be invalidated as long as this `Parser` object is still alive.

## 6.17.3 Member Function Documentation

### 6.17.3.1 expect_end()

```
template<typename Iter , typename Sentinel >
void wr22::regex_parser::parser::Parser< Iter, Sentinel >::expect_end ( )  [inline]
```

Ensure that the parser has consumed all of the input.

Does nothing if all input has been consumed.

**Exceptions**

| | |
|---|---|
| *errors::ExpectedEnd* | if this is not the case. |

### 6.17.3.2 parse_alternatives()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_alternatives (
)  [inline]
```

Intermediate rule: parse a pipe-separated list of alternatives (e.g.

`a|bb|ccc`).

**Returns**

the list of parsed alternatives packed into `regex::part::Alternatives` or, if and only if the list of alternatives contains exactly 1 element, the only alternative unchanged.

**Exceptions**

| | |
|---|---|
| *errors::ParseError* | if the input cannot be parsed. |

### 6.17.3.3 parse_atom()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_atom ( )  [inline]
```

Intermediate rule: parse an atom.

Currently, this grammar only recognizes two kinds of atoms: character literals (individual plain characters in a regex) and parenthesized groups. As the project development goes on, new kinds of atoms will be added.

**Returns**

the parsed atom (some variant of `regex::SpannedPart` depending on the atom kind).

**Exceptions**

| | |
|---|---|
| *errors::ParseError* | if the input cannot be parsed. |

#### 6.17.3.4 parse_char_literal()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_char_literal (
) [inline]
```

Intermediate rule: parse a character literal.

**Returns**

the parsed character literal (`regex::part::Literal`).

**Exceptions**

| | |
|---|---|
| *errors::UnexpectedEnd* | if all characters from the input have already been consumed. |

#### 6.17.3.5 parse_group()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_group ( )
[inline]
```

Intermediate rule: parse a parenthesized group (any capture variant).

**Returns**

the parsed group (`regex::part::Group`).

#### 6.17.3.6 parse_group_name()

```
template<typename Iter , typename Sentinel >
std::pair< std::string, Span > wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_↩
group_name ( ) [inline]
```

Intermediate rule: parse a group name.

**Returns**

the UTF-8 encoded group name as an `std::string`.

#### 6.17.3.7 parse_regex()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_regex ( )
[inline]
```

Parse a regex consuming part of the remaining input.

This is **the** method that should be called to parse a regular expression because it represents the root rule of the regex grammar. Please note that this method may not consume all of the parser's input. Hence, if a whole regex is to be parsed, the `expect_end` method should be called afterwards.

**Returns**

the parsed regex AST (some variant of `regex::SpannedPart` depending on the input).

**Exceptions**

| | |
|---|---|
| *errors::ParseError* | if the input cannot be parsed. |

#### 6.17.3.8 parse_sequence()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_sequence ( )
[inline]
```

Intermediate rule: parse a sequence of atoms (e.g.

`a(?:b)[c-e]`).

**Returns**

the list of parsed atoms packed into `regex::part::Sequence` or, if and only if this list of contains exactly 1 element, the only atom unchanged.

**Exceptions**

| | |
|---|---|
| *errors::ParseError* | if the input cannot be parsed. |

#### 6.17.3.9 parse_sequence_or_empty()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_sequence_or_↩
empty ( )  [inline]
```

Intermediate rule: parse a possibly empty sequence of atoms.

**Returns**

`regex::part::Empty` if the sequence is empty, or calls `parse_sequence` otherwise.

**Exceptions**

| | |
|---|---|
| *errors::ParseError* | if the input cannot be parsed. |

#### 6.17.3.10 parse_wildcard()

```
template<typename Iter , typename Sentinel >
regex::SpannedPart wr22::regex_parser::parser::Parser< Iter, Sentinel >::parse_wildcard ( )
[inline]
```

Intermediate rule: parse a wildcard (`.`

).

**Returns**

the wildcard AST node.

**Exceptions**

| | |
|---|---|
| *errors::UnexpectedEnd* | if all characters from the input have already been consumed. |
| *errors::UnexpectedChar* | if the next input character is not `.`. |

The documentation for this class was generated from the following file:

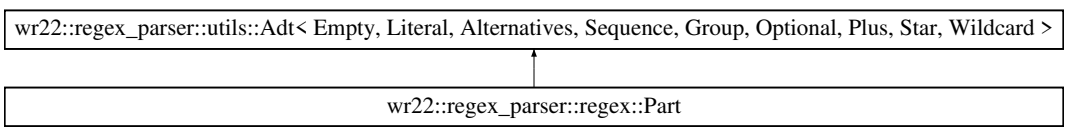
- src/parser/regex.cpp

## 6.18 wr22::regex_parser::regex::Part Class Reference

A part of a regular expression and its AST node type.

`#include <part.hpp>`

Inheritance diagram for wr22::regex_parser::regex::Part:



### Additional Inherited Members

### 6.18.1 Detailed Description

A part of a regular expression and its AST node type.

The parsed regular expressions are represented as abstract syntax trees (ASTs). These are tree-like data structures where each node represents a regular expression part (or the whole regex), and, depending on their type, these nodes may have subexpressions. Subexpressions are `Part`s themselves, which also have child expressions and so on. For example, `part::Sequence` has a number of subexpressions, and each of them is of the type `Part` and is an AST node.

Each regex part has its own simple function. For example, `part::Alternatives` tries to match several alternative subexpressions against the input and succeeds if at least one of them does; and `part::Sequence` matches several subexpressions one after another, requiring them all to match respective parts of the input. By combining these simple nodes, it becomes possible to represent complex regular expressions. For example, the regex `aaa|bb` can be represented as a `part::Alternatives`, where each of the alternatives is a `parts::Sequence` of `part::Literal`s.

The `Part` itself is represented by `std::variant` via the helper class `utils::Adt`. In a nutshell, it allows a regex part to "have" one of the several predefined types (the so-called variants, which are defined in the `part` namespace), but still be represented as a `Part`. For the list of operations that can be performed on this type, e.g. to check if an instance of `Parts` has a specific variant and, if yes, access the value of this variant, see the documentation for the `utils::Adt` class, which `Part` inherits from.

Note that this type contains no span information for the root AST node. For a spanned version, see `SpannedPart`.

The documentation for this class was generated from the following file:

- include/wr22/regex_parser/regex/part.hpp

## 6.19 wr22::regex_parser::regex::part::Plus Struct Reference

A regex part specifying an "at least one" quantifier (`(expression)+`).

`#include <part.hpp>`

### Public Member Functions

- Plus (SpannedPart inner)

  *Convenience constructor.*
- bool operator== (const Plus &rhs) const =default

### Public Attributes

- utils::Box< SpannedPart > inner

  *The smart pointer to the subexpression under the quantifier.*

### Static Public Attributes

- static constexpr const char * code_name = "plus"

### 6.19.1 Detailed Description

A regex part specifying an "at least one" quantifier ( `(expression)+`).

### 6.19.2 Constructor & Destructor Documentation

#### 6.19.2.1 Plus()

```
wr22::regex_parser::regex::part::Plus::Plus (
            SpannedPart inner )  [explicit]
```

Convenience constructor.

### 6.19.3 Member Function Documentation

#### 6.19.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Plus::operator== (
            const Plus & rhs ) const  [default]
```

### 6.19.4 Member Data Documentation

#### 6.19.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Plus::code_name = "plus"  [static],
[constexpr]
```

#### 6.19.4.2 inner

```
utils::Box<SpannedPart> wr22::regex_parser::regex::part::Plus::inner
```

The smart pointer to the subexpression under the quantifier.

The documentation for this struct was generated from the following files:

- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.20 wr22::regex_parser::regex::part::Sequence Struct Reference

A regex part with the list of items to be matched one after another.

`#include <part.hpp>`

### Public Member Functions

- Sequence (std::vector< SpannedPart > items)
- bool operator== (const Sequence &rhs) const =default

### Public Attributes

- std::vector< SpannedPart > items

  *The list of the subexpressions.*

### Static Public Attributes

- static constexpr const char * code_name = "sequence"

### 6.20.1 Detailed Description

A regex part with the list of items to be matched one after another.

Sequences in regular expressions are just subexpressions going directly one after another. As an example, `a[b-e].` is a sequence of 3 subexpressions: `a`, `[b-e]` and `.`. As an another example, `ab` is a sequence of 2 subexpressions: `a` and `b`.

### 6.20.2 Constructor & Destructor Documentation

#### 6.20.2.1 Sequence()

```
wr22::regex_parser::regex::part::Sequence::Sequence (
            std::vector< SpannedPart > items )  [explicit]
```

### 6.20.3 Member Function Documentation

#### 6.20.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Sequence::operator== (
            const Sequence & rhs ) const  [default]
```

### 6.20.4 Member Data Documentation

#### 6.20.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Sequence::code_name = "sequence"  [static],
[constexpr]
```

#### 6.20.4.2 items

```
std::vector<SpannedPart> wr22::regex_parser::regex::part::Sequence::items
```

The list of the subexpressions.

The documentation for this struct was generated from the following files:

- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.21 wr22::regex_parser::span::Span Class Reference

Character position range in the input string.

```
#include <span.hpp>
```

### Public Member Functions

- size_t length () const

  *Get the length of the span (the number of characters covered).*
- size_t begin () const

  *Get the `begin` position of the span.*
- size_t end () const

  *Get the `end` position of the span.*
- bool operator== (const Span &other) const =default
- bool operator!= (const Span &other) const =default

### Static Public Member Functions

- static Span make_empty (size_t position)

  *Construct an empty span that "starts" at a given position.*
- static Span make_single_position (size_t position)

  *Construct a span that captures only one position.*
- static Span make_from_positions (size_t begin, size_t end)

  *Construct a span with given values of `begin` and `end` without any transformations.*
- static Span make_with_length (size_t begin, size_t length)

  *Construct a span with a given value of `begin` and a given length.*

### 6.21.1 Detailed Description

Character position range in the input string.

The range is encoded by two numbers: `begin`, the position (0-based index) of the first character in the range, and `end`, the past-the-end position, or the 0-based index of the last character in the range **plus 1**. This is to be consistent with the behavior of C++ iterators and `begin()`/`end()` functions on STL containers. Please note, however, that the `begin()`/`end()` methods here are just accessors that are not used for iteration, they return plain indices which have no iterator semantics.

Invalid spans (`begin > end`) are not allowed and their construction will result in an error. See the documentation for the relevant methods for details.

### 6.21.2 Member Function Documentation

#### 6.21.2.1 begin()

```
size_t wr22::regex_parser::span::Span::begin ( ) const
```

Get the `begin` position of the span.

#### 6.21.2.2 end()

```
size_t wr22::regex_parser::span::Span::end ( ) const
```

Get the `end` position of the span.

#### 6.21.2.3 length()

```
size_t wr22::regex_parser::span::Span::length ( ) const
```

Get the length of the span (the number of characters covered).

#### 6.21.2.4 make_empty()

```
Span wr22::regex_parser::span::Span::make_empty (
            size_t position )  [static]
```

Construct an empty span that "starts" at a given position.

The resulting span will have `begin = position` and `end = position`.

#### 6.21.2.5 make_from_positions()

```
Span wr22::regex_parser::span::Span::make_from_positions (
            size_t begin,
            size_t end )  [static]
```

Construct a span with given values of `begin` and `end` without any transformations.

**Exceptions**

| | |
|---|---|
| *InvalidSpan* | if `end < begin`. |

#### 6.21.2.6 make_single_position()

```
Span wr22::regex_parser::span::Span::make_single_position (
            size_t position )  [static]
```

Construct a span that captures only one position.

The resulting span will have `begin = position` and `end = position + 1`.

**Exceptions**

| | |
|---|---|
| *InvalidSpan* | if `position + 1` overflows `size_t`. Note that the error message might not be precise enough. |

#### 6.21.2.7 make_with_length()

```
Span wr22::regex_parser::span::Span::make_with_length (
            size_t begin,
            size_t length )  [static]
```

Construct a span with a given value of `begin` and a given length.

The length is determined by the number of characters covered by this span, and, since `begin` and `end` form a half-interval, it equals `end - begin`.

**Exceptions**

| | |
|---|---|
| *InvalidSpan* | if `begin + length` overflows `size_t`. Note that the error message might not be precise enough. |

#### 6.21.2.8 operator"!=()

```
bool wr22::regex_parser::span::Span::operator!= (
            const Span & other ) const  [default]
```

#### 6.21.2.9 operator==()

```
bool wr22::regex_parser::span::Span::operator== (
            const Span & other ) const  [default]
```

The documentation for this class was generated from the following files:

- include/wr22/regex_parser/span/span.hpp
- src/span/span.cpp

## 6.22 wr22::regex_parser::regex::SpannedPart Class Reference

A version of `Part` including the span information (position in the input) of the root AST node (child nodes always contain it because they are represented as `SpannedPart`s themselves).

```
#include <part.hpp>
```

### Public Member Functions

- SpannedPart (Part part, span::Span span)
- bool operator== (const SpannedPart &other) const =default
- bool operator!= (const SpannedPart &other) const =default
- const Part & part () const

  *Access the wrapped `Part` (const version).*
- Part & part ()

  *Access the wrapped `Part` (non-const version).*
- span::Span span () const

  *Get the associated span.*

### 6.22.1 Detailed Description

A version of `Part` including the span information (position in the input) of the root AST node (child nodes always contain it because they are represented as `SpannedPart`s themselves).

### 6.22.2 Constructor & Destructor Documentation

#### 6.22.2.1 SpannedPart()

```
wr22::regex_parser::regex::SpannedPart::SpannedPart (
            Part part,
            span::Span span )  [explicit]
```

### 6.22.3 Member Function Documentation

#### 6.22.3.1 operator"!=()

```
bool wr22::regex_parser::regex::SpannedPart::operator!= (
            const SpannedPart & other ) const  [default]
```

#### 6.22.3.2 operator==()

```
bool wr22::regex_parser::regex::SpannedPart::operator== (
            const SpannedPart & other ) const  [default]
```

#### 6.22.3.3 part() [1/2]

```
Part & wr22::regex_parser::regex::SpannedPart::part ( )
```

Access the wrapped `Part` (non-const version).

#### 6.22.3.4 part() [2/2]

```
const Part & wr22::regex_parser::regex::SpannedPart::part ( ) const
```

Access the wrapped `Part` (const version).

#### 6.22.3.5 span()

```
span::Span wr22::regex_parser::regex::SpannedPart::span ( ) const
```

Get the associated span.

The documentation for this class was generated from the following files:

- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.23 wr22::regex_parser::regex::part::Star Struct Reference

A regex part specifying an "at least zero" quantifier ( `(expression)*`).

`#include <part.hpp>`

### Public Member Functions

- Star (SpannedPart inner)

  *Convenience constructor.*
- bool operator== (const Star &rhs) const =default

### Public Attributes

- utils::Box< SpannedPart > inner

  *The smart pointer to the subexpression under the quantifier.*

### Static Public Attributes

- static constexpr const char * code_name = "star"

### 6.23.1 Detailed Description

A regex part specifying an "at least zero" quantifier ( `(expression)*`).

### 6.23.2 Constructor & Destructor Documentation

#### 6.23.2.1 Star()

```
wr22::regex_parser::regex::part::Star::Star (
            SpannedPart inner )  [explicit]
```

Convenience constructor.

### 6.23.3 Member Function Documentation

#### 6.23.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Star::operator== (
            const Star & rhs ) const  [default]
```

### 6.23.4 Member Data Documentation

#### 6.23.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Star::code_name = "star"  [static],
[constexpr]
```

#### 6.23.4.2 inner

```
utils::Box<SpannedPart> wr22::regex_parser::regex::part::Star::inner
```

The smart pointer to the subexpression under the quantifier.

The documentation for this struct was generated from the following files:

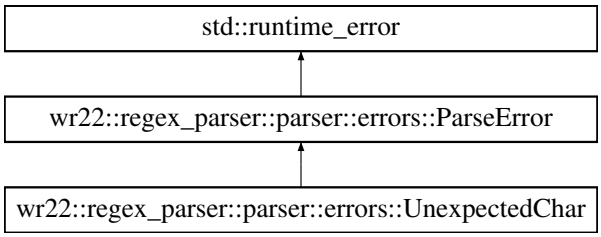
- include/wr22/regex_parser/regex/part.hpp
- src/regex/part.cpp

## 6.24 wr22::regex_parser::parser::errors::UnexpectedChar Class Reference

The error when the parser got a character it didn't expect at the current position.

```
#include <errors.hpp>
```

Inheritance diagram for wr22::regex_parser::parser::errors::UnexpectedChar:




### Public Member Functions

- UnexpectedChar (size_t position, char32_t char_got, std::string expected)

  *Constructor.*
- size_t position () const

  *Get the input position. See the constructor docs for a more detailed description.*
- char32_t char_got () const

  *Get the character the parser has received.*
- const std::string & expected () const

  *Get the description of expected characters.*

### 6.24.1 Detailed Description

The error when the parser got a character it didn't expect at the current position.

### 6.24.2 Constructor & Destructor Documentation

#### 6.24.2.1 UnexpectedChar()

```
wr22::regex_parser::parser::errors::UnexpectedChar::UnexpectedChar (
            size_t position,
            char32_t char_got,
            std::string expected )
```

Constructor.

**Parameters**

| | |
|---|---|
| *position* | the 0-based position in the input when the parser has encountered the unexpected character. |
| *char_got* | the character that the parser has received. |
| *expected* | a textual description of a class of characters expected instead. |

### 6.24.3 Member Function Documentation

#### 6.24.3.1 char_got()

```
char32_t wr22::regex_parser::parser::errors::UnexpectedChar::char_got ( ) const
```

Get the character the parser has received.

See the constructor docs for a more detailed description.

#### 6.24.3.2 expected()

```
const std::string & wr22::regex_parser::parser::errors::UnexpectedChar::expected ( ) const
```

Get the description of expected characters.

See the constructor docs for a more detailed description.

#### 6.24.3.3 position()

```
size_t wr22::regex_parser::parser::errors::UnexpectedChar::position ( ) const
```

Get the input position. See the constructor docs for a more detailed description.

The documentation for this class was generated from the following files:

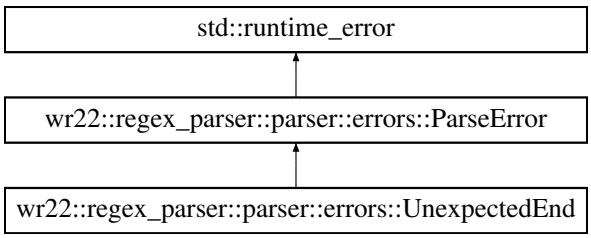
- include/wr22/regex_parser/parser/errors.hpp
- src/parser/errors.cpp

## 6.25 wr22::regex_parser::parser::errors::UnexpectedEnd Class Reference

The error when the parser hit the end of the input earlier than it expected.

```
#include <errors.hpp>
```

Inheritance diagram for wr22::regex_parser::parser::errors::UnexpectedEnd:




### Public Member Functions

- UnexpectedEnd (size_t position, std::string expected)
  *Constructor.*
- size_t position () const
  *Get the input position. See the constructor docs for a more detailed description.*
- const std::string & expected () const
  *Get the description of expected characters.*

### 6.25.1 Detailed Description

The error when the parser hit the end of the input earlier than it expected.

### 6.25.2 Constructor & Destructor Documentation

#### 6.25.2.1 UnexpectedEnd()

```
wr22::regex_parser::parser::errors::UnexpectedEnd::UnexpectedEnd (
            size_t position,
            std::string expected )
```

Constructor.

**Parameters**

| | |
|---|---|
| *position* | the 0-based position in the input when the parser has encountered the end of input. |
| *expected* | a textual description of a class of characters expected instead. |

### 6.25.3 Member Function Documentation

#### 6.25.3.1 expected()

```
const std::string & wr22::regex_parser::parser::errors::UnexpectedEnd::expected ( ) const
```

Get the description of expected characters.

See the constructor docs for a more detailed description.

#### 6.25.3.2 position()

```
size_t wr22::regex_parser::parser::errors::UnexpectedEnd::position ( ) const
```

Get the input position. See the constructor docs for a more detailed description.

The documentation for this class was generated from the following files:

- include/wr22/regex_parser/parser/errors.hpp
- src/parser/errors.cpp

## 6.26 wr22::regex_parser::regex::part::Wildcard Struct Reference

A regex part specifying any single character (`.`).

```
#include <part.hpp>
```

### Public Member Functions

- Wildcard ()=default
- bool operator== (const Wildcard &rhs) const =default

### Static Public Attributes

- static constexpr const char * code_name = "wildcard"

### 6.26.1 Detailed Description

A regex part specifying any single character (`.`).

### 6.26.2 Constructor & Destructor Documentation

#### 6.26.2.1 Wildcard()

```
wr22::regex_parser::regex::part::Wildcard::Wildcard ( )  [explicit], [default]
```

### 6.26.3 Member Function Documentation

#### 6.26.3.1 operator==()

```
bool wr22::regex_parser::regex::part::Wildcard::operator== (
            const Wildcard & rhs ) const  [default]
```

### 6.26.4 Member Data Documentation

#### 6.26.4.1 code_name

```
constexpr const char* wr22::regex_parser::regex::part::Wildcard::code_name = "wildcard"  [static],
[constexpr]
```

The documentation for this struct was generated from the following file:

- include/wr22/regex_parser/regex/part.hpp

# Chapter 7

# File Documentation

## 7.1 include/wr22/regex_parser/parser/errors.hpp File Reference

```
#include <exception>
#include <stdexcept>
#include <string>
```

### Classes

- struct wr22::regex_parser::parser::errors::ParseError

  *The base class for parse errors.*
- class wr22::regex_parser::parser::errors::UnexpectedEnd

  *The error when the parser hit the end of the input earlier than it expected.*
- class wr22::regex_parser::parser::errors::ExpectedEnd

  *The error when the parser expected the input to end, but it did not.*
- class wr22::regex_parser::parser::errors::UnexpectedChar

  *The error when the parser got a character it didn't expect at the current position.*

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::parser
- namespace wr22::regex_parser::parser::errors

## 7.2 errors.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // STL
4 #include <exception>
5 #include <stdexcept>
6 #include <string>
7
8 namespace wr22::regex_parser::parser::errors {
9
15 struct ParseError : public std::runtime_error {
16     using std::runtime_error::runtime_error;
17 };
18
20 class UnexpectedEnd : public ParseError {
21 public:
27     UnexpectedEnd(size_t position, std::string expected);
28
30     size_t position() const;
33     const std::string& expected() const;
34
35 private:
36     size_t m_position;
37     std::string m_expected;
38 };
39
41 class ExpectedEnd : public ParseError {
42 public:
48     ExpectedEnd(size_t position, char32_t char_got);
49
51     size_t position() const;
54     char32_t char_got() const;
55
56 private:
57     size_t m_position;
58     char32_t m_char_got;
59 };
60
62 class UnexpectedChar : public ParseError {
63 public:
70     UnexpectedChar(size_t position, char32_t char_got, std::string expected);
71
73     size_t position() const;
76     char32_t char_got() const;
79     const std::string& expected() const;
80
81 private:
82     size_t m_position;
83     char32_t m_char_got;
84     std::string m_expected;
85 };
86
87 }  // namespace wr22::regex_parser::parser::errors
```

## 7.3 include/wr22/regex_parser/parser/regex.hpp File Reference

```
#include <wr22/regex_parser/regex/part.hpp>
#include <string_view>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::parser

### Functions

- regex::SpannedPart wr22::regex_parser::parser::parse_regex (const std::u32string_view &regex)

  *Parse a regular expression into its AST.*

## 7.4 regex.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // wr22
4 #include <wr22/regex_parser/regex/part.hpp>
5
6 // stl
7 #include <string_view>
8
9 namespace wr22::regex_parser::parser {
10
23 regex::SpannedPart parse_regex(const std::u32string_view& regex);
24
25 }  // namespace wr22::regex_parser::parser
```

## 7.5 include/wr22/regex_parser/regex/capture.hpp File Reference

```
#include <wr22/regex_parser/regex/named_capture_flavor.hpp>
#include <wr22/regex_parser/utils/adt.hpp>
#include <iosfwd>
#include <string>
#include <nlohmann/json.hpp>
```

### Classes

- struct wr22::regex_parser::regex::capture::None

  *Denotes an non-capturing group.*
- struct wr22::regex_parser::regex::capture::Index

  *Denotes a group captured by index.*
- struct wr22::regex_parser::regex::capture::Name

  *Denotes a group captured by name.*
- class wr22::regex_parser::regex::Capture

  *Group capture behavior.*

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::regex
- namespace wr22::regex_parser::regex::capture

### Typedefs

- using wr22::regex_parser::regex::capture::Adt = utils::Adt< None, Index, Name >

### Functions

- void wr22::regex_parser::regex::capture::to_json (nlohmann::json &j, const None &capture)
- void wr22::regex_parser::regex::capture::to_json (nlohmann::json &j, const Index &capture)
- void wr22::regex_parser::regex::capture::to_json (nlohmann::json &j, const Name &capture)
- std::ostream & wr22::regex_parser::regex::operator<< (std::ostream &out, const Capture &capture)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, const Capture &capture)

## 7.6 capture.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // wr22
4 #include <wr22/regex_parser/regex/named_capture_flavor.hpp>
5 #include <wr22/regex_parser/utils/adt.hpp>
6
7 // stl
8 #include <iosfwd>
9 #include <string>
10
11 // nlohmann
12 #include <nlohmann/json.hpp>
13
14 namespace wr22::regex_parser::regex {
15
16 class Capture;
17
18 namespace capture {
20     struct None {
21         explicit None() = default;
22         bool operator==(const None& rhs) const = default;
23         static constexpr const char* code_name = "none";
24     };
25     void to_json(nlohmann::json& j, const None& capture);
26
28     struct Index {
29         explicit Index() = default;
30         bool operator==(const Index& rhs) const = default;
31         static constexpr const char* code_name = "index";
32     };
33     void to_json(nlohmann::json& j, const Index& capture);
34
39     struct Name {
40         explicit Name(std::string name, NamedCaptureFlavor flavor);
41
42         std::string name;
43         NamedCaptureFlavor flavor;
44         bool operator==(const Name& rhs) const = default;
45         static constexpr const char* code_name = "name";
46     };
47     void to_json(nlohmann::json& j, const Name& capture);
48
49     using Adt = utils::Adt<None, Index, Name>;
50 }  // namespace capture
51
57 //
61 class Capture : public capture::Adt {
62 public:
63     using capture::Adt::Adt;
64 };
65
66 std::ostream& operator<<(std::ostream& out, const Capture& capture);
67 void to_json(nlohmann::json& j, const Capture& capture);
68
69 }  // namespace wr22::regex_parser::regex
```

## 7.7 include/wr22/regex_parser/regex/named_capture_flavor.hpp File Reference

```
#include <iosfwd>
#include <nlohmann/json.hpp>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::regex

### Enumerations

- enum class wr22::regex_parser::regex::NamedCaptureFlavor { wr22::regex_parser::regex::Apostrophes , wr22::regex_parser::regex::Angles , wr22::regex_parser::regex::AnglesWithP }

  *The flavor (dialect) of a named group capture.*

### Functions

- std::ostream & wr22::regex_parser::regex::operator<< (std::ostream &out, NamedCaptureFlavor flavor)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, NamedCaptureFlavor flavor)

## 7.8 named_capture_flavor.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // stl
4 #include <iosfwd>
5
6 // nlohmann
7 #include <nlohmann/json.hpp>
8
9 namespace wr22::regex_parser::regex {
10
15 enum class NamedCaptureFlavor
16 {
19     Apostrophes,
22     Angles,
24     AnglesWithP,
25 };
26
27 std::ostream& operator<<(std::ostream& out, NamedCaptureFlavor flavor);
28 void to_json(nlohmann::json& j, NamedCaptureFlavor flavor);
29
30 }  // namespace wr22::regex_parser::regex
```

## 7.9 include/wr22/regex_parser/regex/part.hpp File Reference

```
#include <nlohmann/json_fwd.hpp>
#include <wr22/regex_parser/regex/capture.hpp>
#include <wr22/regex_parser/span/span.hpp>
#include <wr22/regex_parser/utils/adt.hpp>
#include <wr22/regex_parser/utils/box.hpp>
#include <iosfwd>
#include <memory>
#include <vector>
#include <nlohmann/json.hpp>
```

### Classes

- struct wr22::regex_parser::regex::part::Empty

  *An empty regex part.*
- struct wr22::regex_parser::regex::part::Literal

  *An regex part that matches a single character literally.*
- struct wr22::regex_parser::regex::part::Alternatives

*A regex part with the list of alternatives to be matched.*

- struct wr22::regex_parser::regex::part::Sequence

  *A regex part with the list of items to be matched one after another.*

- struct wr22::regex_parser::regex::part::Group

  *A regex part that represents a group in parentheses.*

- struct wr22::regex_parser::regex::part::Optional

  *A regex part specifying an optional quantifier (`(expression)?`).*

- struct wr22::regex_parser::regex::part::Plus

  *A regex part specifying an "at least one" quantifier (`(expression)+`).*

- struct wr22::regex_parser::regex::part::Star

  *A regex part specifying an "at least zero" quantifier (`(expression)*`).*

- struct wr22::regex_parser::regex::part::Wildcard

  *A regex part specifying any single character (`.`).*

- class wr22::regex_parser::regex::Part

  *A part of a regular expression and its AST node type.*

- class wr22::regex_parser::regex::SpannedPart

  *A version of `Part` including the span information (position in the input) of the root AST node (child nodes always contain it because they are represented as `SpannedPart`s themselves).*

## Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::regex
- namespace wr22::regex_parser::regex::part

  *The namespace with the variants of `Part`.*

## Typedefs

- using wr22::regex_parser::regex::part::Adt = utils::Adt< Empty, Literal, Alternatives, Sequence, Group, Optional, Plus, Star, Wildcard >

## Functions

- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Empty &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Literal &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Alternatives &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Sequence &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Group &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Optional &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Plus &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Star &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Wildcard &part)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, const Part &part)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, const SpannedPart &part)
- std::ostream & wr22::regex_parser::regex::operator<< (std::ostream &out, const SpannedPart &part)

  *Convert a `SpannedPart` to a textual representation and write it to an `std::ostream`.*

# 7.10 part.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // wr22
4 #include <nlohmann/json_fwd.hpp>
5 #include <wr22/regex_parser/regex/capture.hpp>
6 #include <wr22/regex_parser/span/span.hpp>
7 #include <wr22/regex_parser/utils/adt.hpp>
8 #include <wr22/regex_parser/utils/box.hpp>
9
10 // stl
11 #include <iosfwd>
12 #include <memory>
13 #include <vector>
14
15 // nlohmann
16 #include <nlohmann/json.hpp>
17
18 namespace wr22::regex_parser::regex {
19
20 // Forward declarations.
21 class Part;
22 class SpannedPart;
23
27 namespace part {
32     struct Empty {
33         explicit Empty() = default;
34         bool operator==(const Empty& rhs) const = default;
35         static constexpr const char* code_name = "empty";
36     };
37     void to_json(nlohmann::json& j, const Empty& part);
38
43     struct Literal {
44         explicit Literal(char32_t character);
45         bool operator==(const Literal& rhs) const = default;
46         static constexpr const char* code_name = "literal";
47
48         char32_t character;
49     };
50     void to_json(nlohmann::json& j, const Literal& part);
51
60     struct Alternatives {
61         explicit Alternatives(std::vector<SpannedPart> alternatives);
62         bool operator==(const Alternatives& rhs) const = default;
63         static constexpr const char* code_name = "alternatives";
64
66         std::vector<SpannedPart> alternatives;
67     };
68     void to_json(nlohmann::json& j, const Alternatives& part);
69
75     struct Sequence {
76         explicit Sequence(std::vector<SpannedPart> items);
77         bool operator==(const Sequence& rhs) const = default;
78         static constexpr const char* code_name = "sequence";
79
81         std::vector<SpannedPart> items;
82     };
83     void to_json(nlohmann::json& j, const Sequence& part);
84
95     struct Group {
97         explicit Group(Capture capture, SpannedPart inner);
98         bool operator==(const Group& rhs) const = default;
99         static constexpr const char* code_name = "group";
100
102         Capture capture;
104         utils::Box<SpannedPart> inner;
105     };
106     void to_json(nlohmann::json& j, const Group& part);
107
109     struct Optional {
111         explicit Optional(SpannedPart inner);
112         bool operator==(const Optional& rhs) const = default;
113         static constexpr const char* code_name = "optional";
114
116         utils::Box<SpannedPart> inner;
117     };
118     void to_json(nlohmann::json& j, const Optional& part);
119
121     struct Plus {
123         explicit Plus(SpannedPart inner);
124         bool operator==(const Plus& rhs) const = default;
125         static constexpr const char* code_name = "plus";
126
```

```
128         utils::Box<SpannedPart> inner;
129     };
130     void to_json(nlohmann::json& j, const Plus& part);
131
133     struct Star {
135         explicit Star(SpannedPart inner);
136         bool operator==(const Star& rhs) const = default;
137         static constexpr const char* code_name = "star";
138
140         utils::Box<SpannedPart> inner;
141     };
142     void to_json(nlohmann::json& j, const Star& part);
143
145     struct Wildcard {
146         explicit Wildcard() = default;
147         bool operator==(const Wildcard& rhs) const = default;
148         static constexpr const char* code_name = "wildcard";
149     };
150     void to_json(nlohmann::json& j, const Wildcard& part);
151
152     using Adt = utils::
153         Adt<Empty, Literal, Alternatives, Sequence, Group, Optional, Plus, Star, Wildcard>;
154 }  // namespace part
155
182 class Part : public part::Adt {
183 public:
184     using part::Adt::Adt;
185 };
186 void to_json(nlohmann::json& j, const Part& part);
187
190 class SpannedPart {
191 public:
192     explicit SpannedPart(Part part, span::Span span);
193
194     bool operator==(const SpannedPart& other) const = default;
195     bool operator!=(const SpannedPart& other) const = default;
196
198     const Part& part() const;
200     Part& part();
201
203     span::Span span() const;
204
205 private:
206     Part m_part;
207     span::Span m_span;
208 };
209 void to_json(nlohmann::json& j, const SpannedPart& part);
210
212 std::ostream& operator<<(std::ostream& out, const SpannedPart& part);
213
214 }  // namespace wr22::regex_parser::regex
```

## 7.11 include/wr22/regex_parser/span/span.hpp File Reference

```
#include <cstddef>
#include <stdexcept>
#include <ostream>
#include <nlohmann/json.hpp>
```

### Classes

- struct wr22::regex_parser::span::InvalidSpan

  *The exception thrown on an attempt to construct an invalid span.*
- class wr22::regex_parser::span::Span

  *Character position range in the input string.*

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::span

### Functions

- std::ostream & wr22::regex_parser::span::operator<< (std::ostream &out, Span span)
- void wr22::regex_parser::span::to_json (nlohmann::json &j, Span span)

## 7.12 span.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // stl
4 #include <cstddef>
5 #include <stdexcept>
6 #include <ostream>
7
8 // nlohmann
9 #include <nlohmann/json.hpp>
10
11 namespace wr22::regex_parser::span {
12
16 struct InvalidSpan : public std::runtime_error {
17     InvalidSpan(size_t begin, size_t end);
18
19     size_t begin;
20     size_t end;
21 };
22
34 class Span {
35 public:
39     static Span make_empty(size_t position);
40
46     static Span make_single_position(size_t position);
47
51     static Span make_from_positions(size_t begin, size_t end);
52
60     static Span make_with_length(size_t begin, size_t length);
61
63     size_t length() const;
64
66     size_t begin() const;
67
69     size_t end() const;
70
71     bool operator==(const Span& other) const = default;
72     bool operator!=(const Span& other) const = default;
73
74 private:
81     explicit Span(size_t begin, size_t end);
82
83     size_t m_begin;
84     size_t m_end;
85 };
86
87 std::ostream& operator«(std::ostream& out, Span span);
88 void to_json(nlohmann::json& j, Span span);
89
90 }  // namespace wr22::regex_parser::span
```

## 7.13 include/wr22/regex_parser/utils/adt.hpp File Reference

```
#include <utility>
#include <variant>
```

### Classes

- struct wr22::regex_parser::utils::detail::adt::MultiCallable< Fs >
- class wr22::regex_parser::utils::Adt< Variants >

  *A helper class that simplifies creation of algebraic data types.*

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::utils
- namespace wr22::regex_parser::utils::detail
- namespace wr22::regex_parser::utils::detail::adt

### Functions

- template<typename... Variants>
  bool wr22::regex_parser::utils::operator== (const Adt< Variants... > &lhs, const Adt< Variants... > &rhs)

  *Compare two compatible ADTs for equality.*
- template<typename... Variants>
  bool wr22::regex_parser::utils::operator!= (const Adt< Variants... > &lhs, const Adt< Variants... > &rhs)

  *Compare two compatible ADTs for non-equality.*

## 7.14 adt.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // stl
4 #include <utility>
5 #include <variant>
6
7 namespace wr22::regex_parser::utils {
8
9 namespace detail::adt {
10     // https://en.cppreference.com/w/cpp/utility/variant/visit#Example provides a very similar
11     // example of C++ template black magic.
12     template <typename... Fs>
13     struct MultiCallable : public Fs... {
14         MultiCallable(Fs&&... fs) : Fs(fs)... {}
15         using Fs::operator()...;
16     };
17 }  // namespace detail::adt
18
28 template <typename... Variants>
29 class Adt {
30 public:
32     using VariantType = std::variant<Variants...>;
33
43     template <typename V>
44     Adt(V variant) : m_variant(std::move(variant)) {}
45
74     template <typename... Fs>
75     decltype(auto) visit(Fs&&... visitors) const {
76         return std::visit(
77             detail::adt::MultiCallable<Fs...>(std::forward<Fs>(visitors)...),
78             m_variant);
79     }
80
87     template <typename... Fs>
88     decltype(auto) visit(Fs&&... visitors) {
89         return std::visit(
90             detail::adt::MultiCallable<Fs...>(std::forward<Fs>(visitors)...),
91             m_variant);
92     }
93
95     const VariantType& as_variant() const {
96         return m_variant;
97     }
98
100     VariantType& as_variant() {
101         return m_variant;
102     }
103
104 protected:
105     VariantType m_variant;
```

```
106 };
107
109 template <typename... Variants>
110 bool operator==(const Adt<Variants...>& lhs, const Adt<Variants...>& rhs) {
111     return lhs.as_variant() == rhs.as_variant();
112 }
113
115 template <typename... Variants>
116 bool operator!=(const Adt<Variants...>& lhs, const Adt<Variants...>& rhs) {
117     return !(lhs == rhs);
118 }
119
120 }  // namespace wr22::regex_parser::utils
```

## 7.15 include/wr22/regex_parser/utils/box.hpp File Reference

```
#include <exception>
#include <memory>
#include <utility>
```

### Classes

- struct wr22::regex_parser::utils::BoxIsEmpty
- class wr22::regex_parser::utils::Box< T >

  *A copyable and equality-comparable wrapper around `std::unique_ptr`.*

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::utils

### Functions

- template<typename T >
  wr22::regex_parser::utils::Box (T &&value) -> Box< T >

  *Type deduction guideline for `Box` (value initialization).*
- template<typename T >
  wr22::regex_parser::utils::Box (std::unique_ptr< T > ptr) -> Box< T >

  *Type deduction guideline for `Box` (`std::unique_ptr` adoption).*
- template<typename T , typename U >
  bool wr22::regex_parser::utils::operator== (const Box< T > &lhs, const Box< U > &rhs)
- template<typename T , typename U >
  bool wr22::regex_parser::utils::operator!= (const Box< T > &lhs, const Box< U > &rhs)

## 7.16 box.hpp

Go to the documentation of this file.

```
1 #pragma once
2
3 // stl
4 #include <exception>
5 #include <memory>
6 #include <utility>
7
8 namespace wr22::regex_parser::utils {
9
10 struct BoxIsEmpty : public std::exception {
11     const char* what() const noexcept override;
12 };
13
25 template <typename T>
26 class Box {
27 public:
32     explicit Box(T&& value) : m_ptr(std::make_unique<T>(std::forward<T>(value))) {}
33
49     explicit Box(std::unique_ptr<T> ptr) : m_ptr(std::move(ptr)) {}
50
57     template <typename Dummy = T>
58     Box(const Box& other) : m_ptr(std::make_unique<T>(*other)) {}
59
63     template <typename... Args>
64     static Box<T> construct_in_place(Args&&... args) {
65         return Box(std::make_unique<T>(std::forward<Args>(args)...));
66     }
67
71     const T& operator*() const {
72         if (m_ptr == nullptr) {
73             throw BoxIsEmpty{};
74         }
75         return *m_ptr;
76     }
77
81     T& operator*() {
82         if (m_ptr == nullptr) {
83             throw BoxIsEmpty{};
84         }
85         return *m_ptr;
86     }
87
88 private:
89     std::unique_ptr<T> m_ptr;
90 };
91
93 template <typename T>
94 Box(T&& value) -> Box<T>;
95
97 template <typename T>
98 Box(std::unique_ptr<T> ptr) -> Box<T>;
99
100 template <typename T, typename U>
101 bool operator==(const Box<T>& lhs, const Box<U>& rhs) {
102     return *lhs == *rhs;
103 }
104
105 template <typename T, typename U>
106 bool operator!=(const Box<T>& lhs, const Box<U>& rhs) {
107     return !(lhs == rhs);
108 }
109
110 }  // namespace wr22::regex_parser::utils
```

## 7.17 src/parser/capture.cpp File Reference

```
#include <wr22/regex_parser/regex/capture.hpp>
#include <wr22/regex_parser/regex/named_capture_flavor.hpp>
#include <iterator>
#include <ostream>
#include <boost/locale/utf.hpp>
#include <fmt/core.h>
#include <fmt/ostream.h>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::regex
- namespace wr22::regex_parser::regex::capture

### Functions

- std::ostream & wr22::regex_parser::regex::operator<< (std::ostream &out, const Capture &capture)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, const Capture &capture)
- void wr22::regex_parser::regex::capture::to_json (nlohmann::json &j, const None &capture)
- void wr22::regex_parser::regex::capture::to_json (nlohmann::json &j, const Index &capture)
- void wr22::regex_parser::regex::capture::to_json (nlohmann::json &j, const Name &capture)

## 7.18 src/parser/errors.cpp File Reference

```
#include <wr22/regex_parser/parser/errors.hpp>
#include <fmt/core.h>
#include <boost/locale/encoding_utf.hpp>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::parser
- namespace wr22::regex_parser::parser::errors

## 7.19 src/parser/regex.cpp File Reference

```
#include <wr22/regex_parser/parser/errors.hpp>
#include <wr22/regex_parser/parser/regex.hpp>
#include <wr22/regex_parser/regex/part.hpp>
#include <optional>
#include <stdexcept>
#include <string>
#include <vector>
#include <boost/locale/encoding_utf.hpp>
#include <boost/locale/utf.hpp>
```

### Classes

- class wr22::regex_parser::parser::Parser< Iter, Sentinel >

  *A regex parser.*

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::parser

### Functions

- template<typename Iter , typename Sentinel >
  wr22::regex_parser::parser::Parser (Iter begin, Sentinel end) -> Parser< Iter, Sentinel >

  *The type deduction guideline for `Parser`.*
- regex::SpannedPart wr22::regex_parser::parser::parse_regex (const std::u32string_view &regex)

  *Parse a regular expression into its AST.*

## 7.20 src/regex/named_capture_flavor.cpp File Reference

```
#include <wr22/regex_parser/regex/named_capture_flavor.hpp>
#include <ostream>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::regex

### Functions

- std::ostream & wr22::regex_parser::regex::operator<< (std::ostream &out, NamedCaptureFlavor flavor)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, NamedCaptureFlavor flavor)

## 7.21 src/regex/part.cpp File Reference

```
#include "wr22/regex_parser/span/span.hpp"
#include <boost/locale/encoding_utf.hpp>
#include <wr22/regex_parser/regex/part.hpp>
#include <iterator>
#include <ostream>
#include <boost/locale/utf.hpp>
#include <fmt/core.h>
#include <fmt/ostream.h>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::regex
- namespace wr22::regex_parser::regex::part

  *The namespace with the variants of* `Part`.

### Functions

- std::ostream & wr22::regex_parser::regex::operator<< (std::ostream &out, const SpannedPart &part)

  *Convert a* `SpannedPart` *to a textual representation and write it to an* `std::ostream`.
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Empty &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Literal &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Alternatives &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Sequence &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Group &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Optional &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Plus &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Star &part)
- void wr22::regex_parser::regex::part::to_json (nlohmann::json &j, const part::Wildcard &part)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, const Part &part)
- void wr22::regex_parser::regex::to_json (nlohmann::json &j, const SpannedPart &part)

## 7.22 src/span/span.cpp File Reference

```
#include <stdexcept>
#include <wr22/regex_parser/span/span.hpp>
#include <fmt/core.h>
#include <fmt/ostream.h>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::span

### Functions

- std::ostream & wr22::regex_parser::span::operator<< (std::ostream &out, Span span)
- void wr22::regex_parser::span::to_json (nlohmann::json &j, Span span)

## 7.23 src/utils/box.cpp File Reference

```
#include <wr22/regex_parser/utils/box.hpp>
```

### Namespaces

- namespace wr22
- namespace wr22::regex_parser
- namespace wr22::regex_parser::utils

# Index