Boost.Spirit 2.5.2 Reference Card

About Attributes

Every parser has an associated synthesized attribute with an associated type. Rules and grammars can explicitly specify the type of the associated synthesized attribute. Attribute types synthesized by Spirit are associated with these concepts:

Unused	Unused attribute type
Optional<A>	Optional attribute possibly holding type A
Tuple<A,B>	Sequence attribute containing types A and B
Variant<A,B>	Variant attribute containing either type A or B
Vector<A>	Container attribute holding elements of type A

Examples of types that model these concepts include:

Optional<A>	`boost::optional<A>`
Tuple<A,B>	`std::tuple<A,B>`
Variant<A,B>	`boost::variant<A,B>`
Vector<A>	`std::vector<A>`
Vector<Ch>	`std::basic_string<Ch>`

Nonterminal Parsers

`rule<It,RT(A1,...,An),Skip,Loc>`	Rule	`RT`
`rule<It>`		*Unused*
`rule<It,Skip>`		*Unused*
`rule<It,Loc>`		*Unused*
`rule<It,Skip,Loc>`		*Unused*
`grammar<It,RT(A1,...,An),Skip,Loc>`	Grammar	`RT`
`grammar<It>`		*Unused*
`grammar<It,Skip>`		*Unused*
`grammar<It,Loc>`		*Unused*
`grammar<It,Skip,Loc>`		*Unused*

Primitive Parsers

`attr(a)`	Attribute	`A`
`eoi`	End of Input	*Unused*
`eol`	End of Line	*Unused*
`eps`	Epsilon	*Unused*
`eps(p)`		*Unused*
`symbols<Ch,T,Lookup>`	Symbol Table	`T`

Unary Parsers

`&a`	And Predicate	*Unused*
`!a`	Not Predicate	*Unused*
`*a`	Zero or More	**Vector**<A>
`*u`		*Unused*
`-a`	Optional	**Optional**<A>
`-u`		*Unused*
`+a`	One or More	**Vector**<A>
`+u`		*Unused*
`attr_cast<T>(p)`	Attribute Cast	`T`

Binary Parsers

`a - b`	Difference	`A`
`u - b`		*Unused*
`a % b`	Separated List	**Vector**<A>
`u % b`		*Unused*

N-ary Parsers

`a | b`	Alternative	**Variant**<A,B>
`a | u`		**Optional**<A>
`a | b | u`		**Optional<Variant**<A,B>>
`u | b`		**Optional**<B>
`u | u`		*Unused*
`a | a`		`A`
`a > b`	Expect	**Tuple**<A,B>
`a > u`		`A`
`u > b`		`B`
`u > u`		*Unused*
`a > vA`		**Vector**<A>
`vA > a`		**Vector**<A>
`vA > vA`		**Vector**<A>
`a ^ b`	Permute	**Tuple<Optional**<A>,**Optional**<B>>
`a ^ u`		**Optional**<A>
`u ^ b`		**Optional**<B>
`u ^ u`		*Unused*
`a >> b`	Sequence	**Tuple**<A,B>
`a >> u`		`A`
`u >> b`		`B`
`u >> u`		*Unused*
`a >> a`		**Vector**<A>
`a >> vA`		**Vector**<A>
`vA >> a`		**Vector**<A>
`vA >> vA`		**Vector**<A>
`a || b`	Sequence Or	**Tuple<Optional**<A>,**Optional**<B>>
`a || u`		**Optional**<A>
`u || a`		**Optional**<A>
`u || u`		*Unused*
`a || a`		**Vector<Optional**<A>>

Parser Directives

`as<T>{}[a]`	Atomic Assignment	`T`
`hold[a]`	Hold Attribute	`A`
`hold[u]`		*Unused*
`lexeme[a]`	Lexeme	`A`
`lexeme[u]`		*Unused*
`matches[a]`	Matches	`bool`
`no_case[a]`	Case Insensitive	`A`
`no_case[u]`		*Unused*
`no_skip[a]`	No Skipping	`A`
`no_skip[u]`		*Unused*
`omit[a]`	Omit Attribute	*Unused*
`raw[a]`	Raw Iterators	`iterator_range<It>`
`raw[u]`		*Unused*
`repeat[a]`	Repeat	**Vector**<A>
`repeat[u]`		*Unused*
`repeat(n)[a]`		**Vector**<A>
`repeat(n)[u]`		*Unused*
`repeat(min,max)[a]`		**Vector**<A>
`repeat(min,max)[u]`		*Unused*
`repeat(min,inf)[a]`		**Vector**<A>
`repeat(min,inf)[u]`		*Unused*
`skip[a]`	Skip Whitespace	`A`
`skip[u]`		*Unused*
`skip(p)[a]`		`A`
`skip(p)[u]`		*Unused*

Semantic Actions

`p[fa]`	Apply Semantic Action	`A`
`p[`*phoenix lambda*`]`		`A`

```
template<typename Attrib>
void fa(Attrib& attr);

template<typename Attrib, typename Context>
void fa(Attrib& attr, Context& context);

template<typename Attrib, typename Context>
void fa(Attrib& attr, Context& context, bool& pass);
```

Phoenix Placeholders

`_1`, `_2`, ..., `_N`	Nth Attribute of `p`
`_val`	Enclosing rule's synthesized attribute
`_r1`, `_r2`, ..., `_rN`	Enclosing rule's Nth inherited attribute.
`_a`, `_b`, ..., `_j`	Enclosing rule's local variables.
`_pass`	Assign `false` to `_pass` to force failure.

Iterator Parser API

```
bool parse<It, Exp>(
    It& first, It last, Exp const& expr);
bool parse<It, Exp, A1, ..., An>(
    It& first, It last, Exp const& expr,
    A1& a1, ..., An& an);
bool phrase_parse<It, Exp, Skipper>(
    It& first, It last, Exp const& expr,
    Skipper const& skipper,
    skip_flag post_skip = postskip);
bool phrase_parse<It, Exp, Skipper, A1, ..., An>(
    It& first, It last, Exp const& expr,
    Skipper const& skipper,
    A1& a1, ..., An& an);
bool phrase_parse<It, Exp, Skipper, A1, ..., An>(
    It& first, It last, Exp const& expr,
    Skipper const& skipper,
    skip_flag post_skip,
    A1& a1, ..., An& an);
```

Stream Parser API

```
unspecified match<Exp>(Exp const& expr);
unspecified match<Exp, A1, ..., An>(
    Exp const& expr,
    A1& a1, ..., An& an);
unspecified phrase_match<Exp, Skipper>(
    Exp const& expr,
    Skipper const& skipper,
    skip_flag post_skip = postskip);
unspecified phrase_match<Exp, Skipper, A1, ..., An>(
    Exp const& expr,
    Skipper const& skipper,
    skip_flag post_skip,
    A1& a1, ..., An& an);
```


Send comments and corrections to Richard Thomson, ⟨legalize@xmission.com⟩

Binary Value Parsers

`byte_`	Native Byte	`uint_least8_t`
`byte_(b)`		*Unused*
`word`	Native Word	`uint_least16_t`
`word(w)`		*Unused*
`dword`	Native Double Word	`uint_least32_t`
`dword(dw)`		*Unused*
`qword`	Native Quad Word	`uint_least64_t`
`qword(qw)`		*Unused*
`bin_float`	Native Float	`float`
`bin_float(f)`		*Unused*
`bin_double`	Native Double	`double`
`bin_double(d)`		*Unused*
`little_`*item*	Little Endian *item*	as above
`little_`*item*`(w)`		*Unused*
`big_`*item*	Big Endian *item*	as above
`big_`*item*`(w)`		*Unused*

Character Encodings

`ascii`	7-bit ASCII
`iso8859_1`	ISO 8859-1
`standard`	Using `<cctype>`
`standard_wide`	Using `<cwctype>`

Character Parsers

`c`	Character Literal	*Unused*
`lit(c)`		*Unused*
`ns::char_`	Any Character	`ns::char_type`
`ns::char_(c)`	Character Value	`ns::char_type`
`ns::char_(f,l)`	Character Range	`ns::char_type`
`ns::char_(str)`	Any Character in String	`ns::char_type`
`~cp`	Characters not in `cp`	Attribute of `cp`

Character Class Parsers

`ns::alnum`	Letters or Digits	`ns::char_type`
`ns::alpha`	Alphabetic	`ns::char_type`
`ns::blank`	Spaces or Tabs	`ns::char_type`
`ns::cntrl`	Control Characters	`ns::char_type`
`ns::digit`	Numeric Digits	`ns::char_type`
`ns::graph`	Non-space Printing Characters	`ns::char_type`
`ns::lower`	Lower Case Letters	`ns::char_type`
`ns::print`	Printing Characters	`ns::char_type`
`ns::punct`	Punctuation	`ns::char_type`
`ns::space`	White Space	`ns::char_type`
`ns::upper`	Upper Case Letters	`ns::char_type`
`ns::xdigit`	Hexadecimal Digits	`ns::char_type`

String Parsers

`str`	String Literal	*Unused*
`lit(str)`		*Unused*
`ns::string("str")`	String	**Vector**`<char>`
`ns::string(L"str")`		**Vector**`<wchar_t>`

Unsigned Integer Parsers

`lit(num)`	Integer Literal	*Unused*
`ushort_`	Short	`unsigned short`
`ushort_(num)`	Short Value	`unsigned short`
`uint_`	Integer	`unsigned int`
`uint_(num)`	Integer Value	`unsigned int`
`ulong_`	Long	`unsigned long`
`ulong_(num)`	Long Value	`unsigned long`
`ulong_long`	Long Long	`unsigned long long`
`ulong_long(num)`	Long Long Value	`unsigned long long`
`bin`	Binary Integer	`unsigned int`
`bin(num)`	Binary Integer Value	`unsigned int`
`oct`	Octal Integer	`unsigned int`
`oct(num)`	Octal Integer Value	`unsigned int`
`hex`	Hexadecimal Integer	`unsigned int`
`hex(num)`	Hex Integer Value	`unsigned int`

Generalized Unsigned Integer Parser

`uint_parser<T,Radix,MinDigits,MaxDigits>{}`	`T`
`uint_parser<T,Radix,MinDigits,MaxDigits>{}(num)`	`T`

Signed Integer Parsers

`lit(num)`	Integer Literal	*Unused*
`short_`	Short	`short`
`short_(num)`	Short Value	`short`
`int_`	Integer	`int`
`int_(num)`	Integer Value	`int`
`long_`	Long	`long`
`long_(num)`	Long Value	`long`
`long_long`	Long Long	`long long`
`long_long(num)`	Long Long Value	`long long`

Generalized Signed Integer Parser

`int_parser<T,Radix,MinDigits,MaxDigits>{}`	`T`
`int_parser<T,Radix,MinDigits,MaxDigits>{}(num)`	`T`

Real Number Parsers

`lit(num)`	Real Number Literal	*Unused*
`float_`	Float	`float`
`float_(num)`	Float Value	`float`
`double_`	Double	`double`
`double_(num)`	Double Value	`double`
`long_double`	Long Double	`long double`
`long_double(num)`	Long Double Value	`long double`

Generalized Real Number Parser

`real_parser<T,RealPolicies>{}`	`T`
`real_parser<T,RealPolicies>{}(num)`	`T`

Boolean Parsers

`lit(boolean)`	Boolean Literal	*Unused*
`false_`	Match "`false`"	`bool`
`true_`	Match "`true`"	`bool`
`bool_`	Boolean	`bool`
`bool_(boolean)`	Boolean Value	`bool`

Generalized Boolean Parser

`bool_parser<T,BoolPolicies>{}`	`T`
`bool_parser<T,BoolPolicies>{}(boolean)`	`T`