Boost.Spirit 2.5.2 Reference Card

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>>`

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*

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);
```

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*

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

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	`basic_string<char>`
`ns::string(L"str")`		`basic_string<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`