

Metaparse tutorial

Agenda

- DSL embedding in C++: current practice
- Boost.Xpressive introduction
- Template metaprogramming introduction
- Embedding regular expressions

Lab

- Detailed tutorial
- These slides
- Lab environment
- Solutions

https://github.com/sabel83/metaparse_tutorial

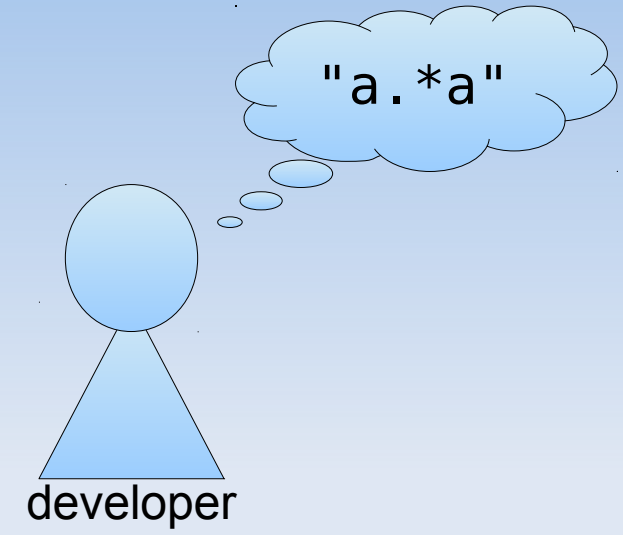
Mpllibs

- Template Metaprogramming libraries
- <http://abel.web.elte.hu/mpllibs>
 - Metaparse
 - Metamonad
 - Safe Printf
 - Xlpressive

Mpllibs

- Ábel Sinkovics
- Endre Sajó
- Zoltán Porkoláb
- István Siroki

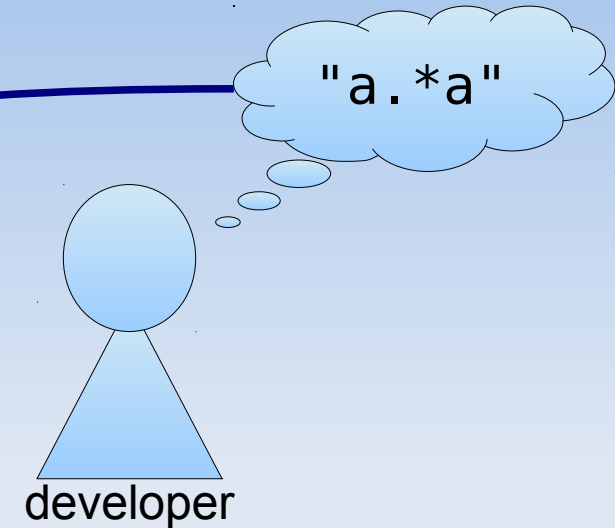
Processing DSLs



Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r = sregex::compile("a.*a");  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

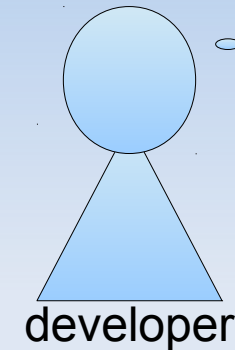


Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r = sregex::compile("a.*a");  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

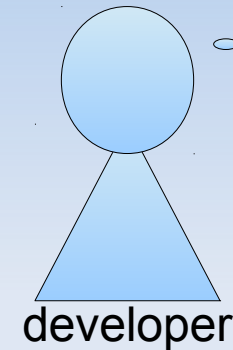
"a.*a"

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r = sregex::compile("a.*a");  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

"a.*a"

Execution

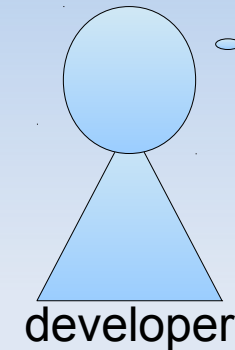
"a.*a"

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r = sregex::compile("a.*a");  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

"a.*a"

Execution

"a.*a"

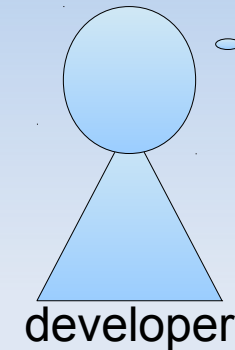
Matching
code

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r = sregex::compile("a.*a");  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

"a.*a"

Execution

"a.*a"

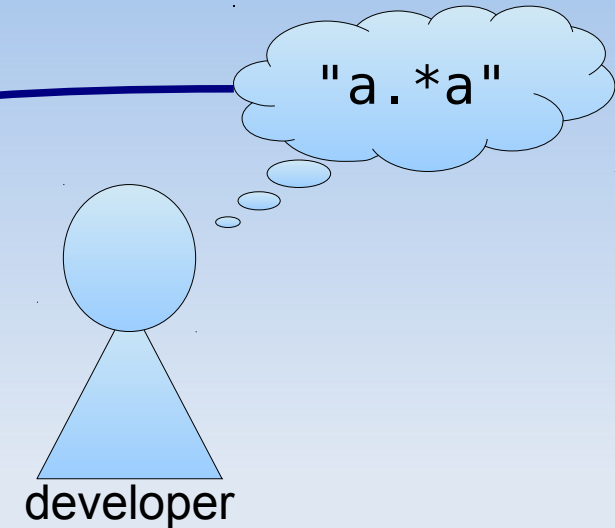
Matching
code

matching...

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r =as_xpr('a') >> *_ >> 'a';  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

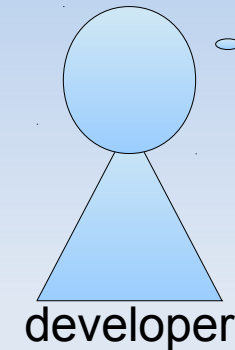


Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r =as_xpr('a') >> *_ >> 'a';  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

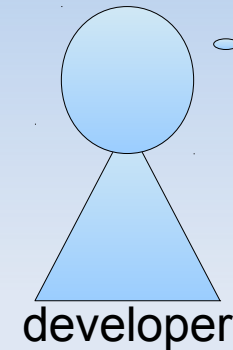
Matching
code

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r =as_xpr('a') >> *_ >> 'a';  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

Matching
code

Execution

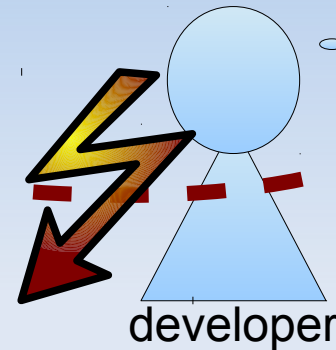
matching...

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r =as_xpr('a') >> *_ >> 'a';  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

Matching
code

Execution

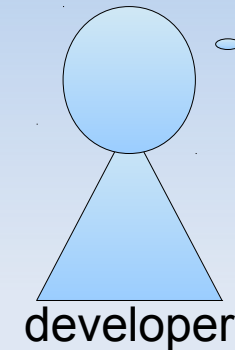
matching...

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r = MPLLIBS_REGEX("a.*a");  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

executable

Matching
code

Execution

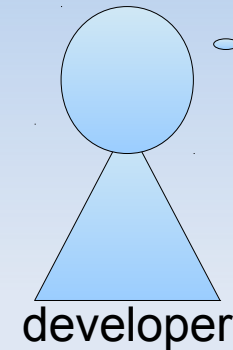
matching...

Processing DSLs

```
int main()  
{  
    string s; cin << s;  
    sregex r = MPLLIBS_REGEX("a.*a");  
    smatch w;  
    regex_search(s, w, r);  
    // ...  
}
```

main.cpp

"a.*a"



Compilation

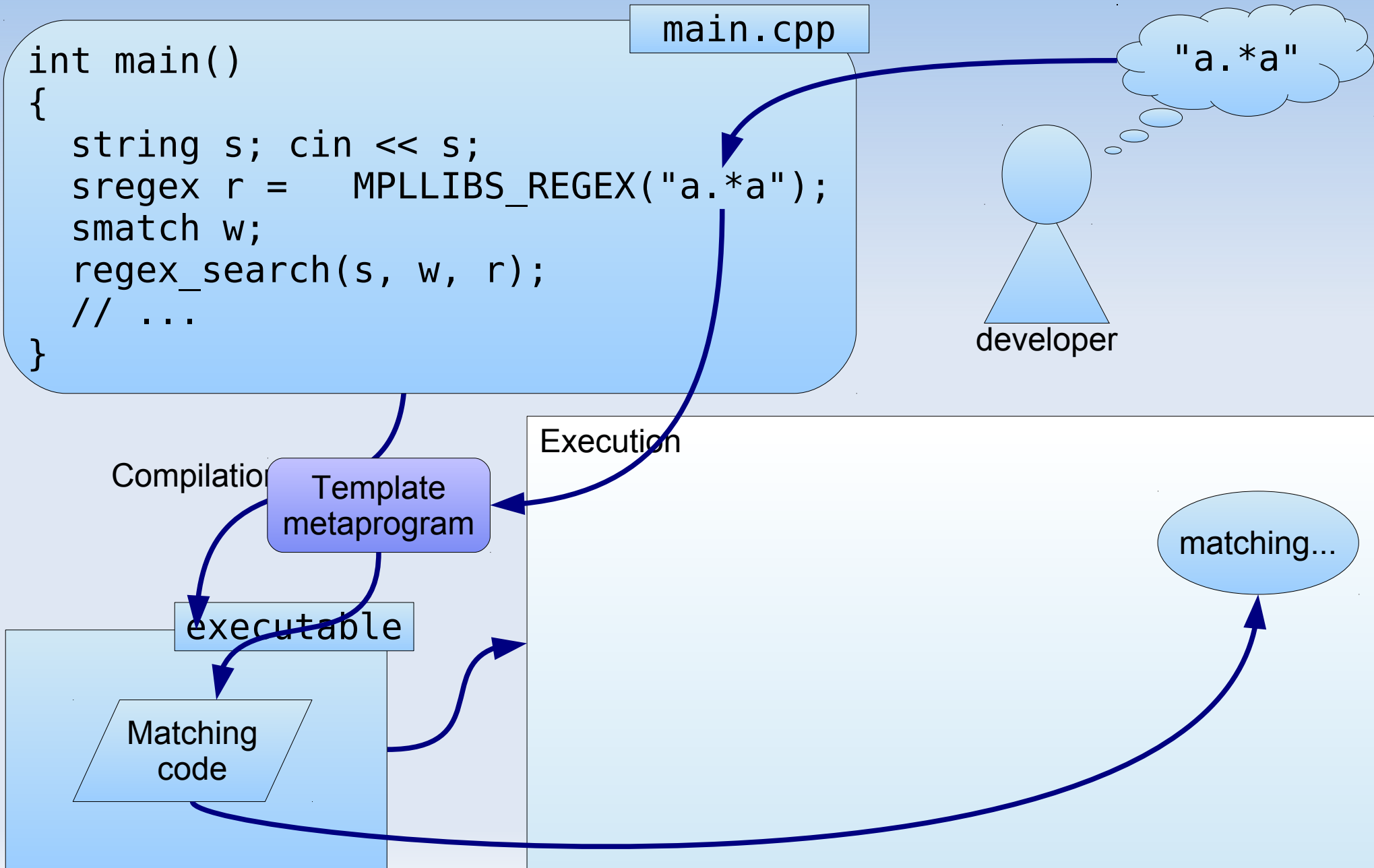
Template
metaprogram

executable

Matching
code

Execution

matching...



Lab 0

Set up your working environment

```
git clone https://github.com/sabel83/metaparse_tutorial  
cd metaparse_tutorial/lab  
make
```

Boost.Xpressive

- Include the headers

```
#include <boost/xpressive/xpressive.hpp>
```

- Create a matching object

```
sregex re = sregex::compile("ab*c");
```

- Do some matching

```
smatch what;  
regex_match("abbbbbbbbc", what, re);
```

Lab 1

- Try Xpressive yourself
- Create a number of regular expressions using Xpressive

```
" "  
"a"  
"abc"  
"b*"  
"ab*"  
"b*c"  
"ab*c"  
"a.*c"  
"a1*c"  
"(abc)*"
```

More Boost.Xpressive

```
sregex re = sregex::compile("*")
```

More Boost.Xpressive

```
sregex re = sregex::compile("*")
```



```
terminate called after throwing an instance of  
'boost::exception_detail::clone_impl<boost::xp  
ressive::regex_error>'  
  what():  quantifier not expected
```

Boost.Xpressive

- Dynamic regex

```
sregex re = sregex::compile("ab*c");
```

Boost.Xpressive

- Dynamic regex

```
sregex re = sregex::compile("ab*c");
```

- Static regex

```
sregex re =
```

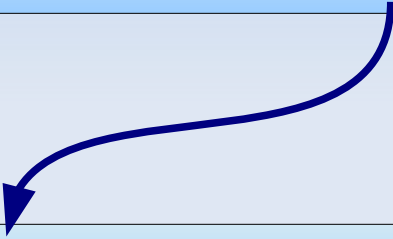

Boost.Xpressive

- Dynamic regex

```
sregex re = sregex::compile("ab*c");
```

- Static regex

```
sregex re = as_xpr('a')
```



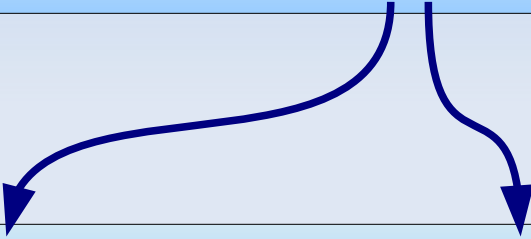
Boost.Xpressive

- Dynamic regex

```
sregex re = sregex::compile("ab*c");
```

- Static regex

```
sregex re = as_xpr('a') as_xpr('b')
```

A diagram consisting of two blue arrows. The first arrow originates from the 'ab*c' string in the code above and points to the 'a' character in the code below. The second arrow originates from the 'ab*c' string and points to the 'b' character in the code below.

Boost.Xpressive

- Dynamic regex

```
sregex re = sregex::compile("ab*c");
```

- Static regex

```
sregex re = as_xpr('a') as_xpr('b') as_xpr('c');
```

A diagram consisting of three blue arrows pointing downwards from the dynamic regex code to the static regex code. The first arrow points from the 'a' in 'ab*c' to 'as_xpr('a')'. The second arrow points from the 'b' in 'ab*c' to 'as_xpr('b')'. The third arrow points from the 'c' in 'ab*c' to 'as_xpr('c')'.

Boost.Xpressive

- Dynamic regex

```
sregex re = sregex::compile("ab*c");
```

- Static regex

```
sregex re = as_xpr('a') * as_xpr('b') * as_xpr('c');
```

The diagram illustrates the decomposition of the dynamic regex `"ab*c"` into its static components. Four blue arrows originate from the characters `a`, `b`, `*`, and `c` in the dynamic regex and point to their respective static equivalents in the static regex: `as_xpr('a')`, `as_xpr('b')`, `*`, and `as_xpr('c')`.

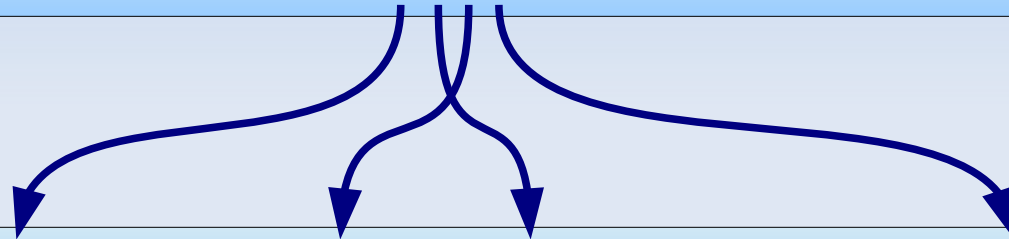
Boost.Xpressive

- Dynamic regex

```
sregex re = sregex::compile("ab*c");
```

- Static regex

```
sregex re = as_xpr('a') >> *as_xpr('b') >> as_xpr('c');
```



Lab 2

- Create the same regular expressions using static regexes of Xpressive

```
" "  
"a"  
"abc"  
"b*"  
"ab*"  
"b*c"  
"ab*c"  
"a.*c"  
"a1*c"  
"(abc)*"
```

Safe static regexes

```
sregex re = REGEX("ab*c");
```



```
sregex re = as_xpr('a') >> *as_xpr('b') >> as_xpr('c');
```

Safe static regexes

```
sregex re = REGEX("ab*c");
```



*Magic happens
here*

```
sregex re = as_xpr('a') >> *as_xpr('b') >> as_xpr('c');
```


Safe static regexes

```
sregex re = REGEX("ab*c");
```



```
graph TD; A["sregex re = REGEX(\"ab*c\");"] --> B["Template metaprogram"]; B --> C["sregex re = as_xpr('a') >> *as_xpr('b') >> as_xpr('c');"]
```

Template metaprogram

```
sregex re = as_xpr('a') >> *as_xpr('b') >> as_xpr('c');
```

Safe static regexes

```
sregex re = REGEX("ab*c");
```

Template metaprogram

type

```
sregex re = [as_xpr('a') >> *as_xpr('b') >> as_xpr('c')];
```


Safe static regexes

```
sregex re = REGEX("ab*c");
```

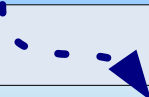


Template metaprogram

```
struct build_my_regex {  
    static sregex run() {  
        return /* ... */;  
    }  
};
```



```
sregex re = [as_xpr('a') >> *as_xpr('b') >> as_xpr('c')];
```




Safe static regexes

```
sregex re = REGEX("ab*c");
```



Template metaprogram

```
struct build_my_regex {  
    static sregex run() {  
        return as_xpr('a') >> *as_xpr('b') >> as_xpr('c');  
    }  
};
```



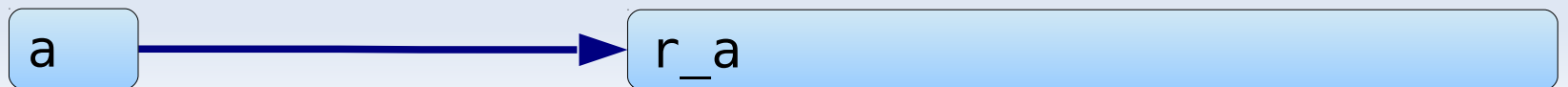
```
sregex re = {build_my_regex::run()};
```



build_my_regex

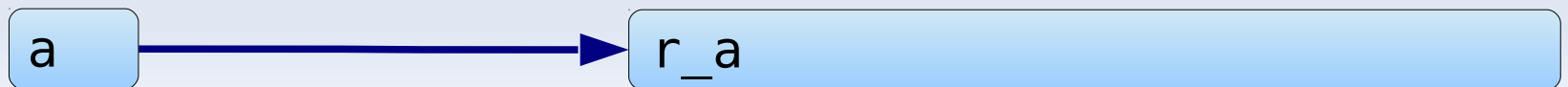
```
struct r_a {
```

```
};
```



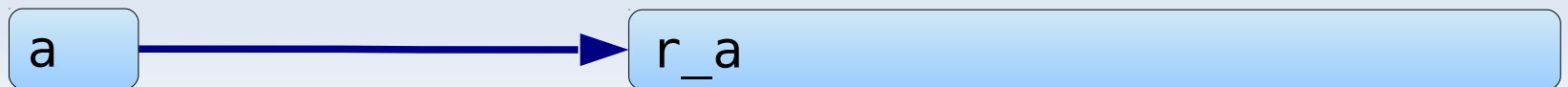
build_my_regex

```
struct r_a {  
    static /* ... */ run() {  
    }  
};
```



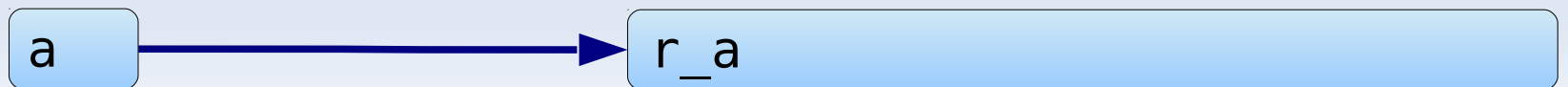
build_my_regex

```
struct r_a {  
    static /* ... */      run() {  
        return as_xpr('a');  
    }  
};
```



build_my_regex

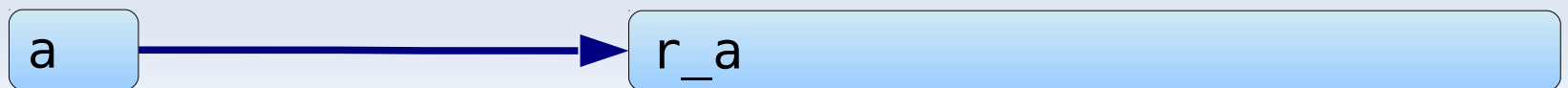
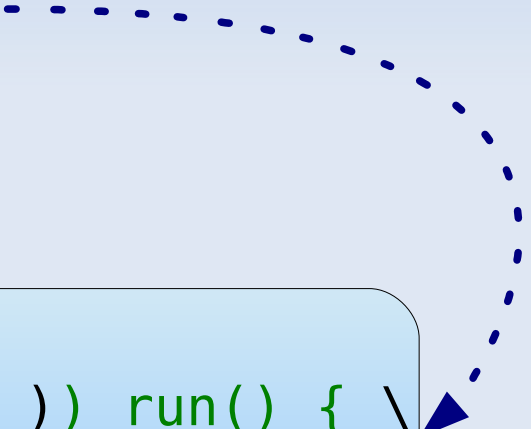
```
struct r_a {  
    static decltype(as_xpr('a')) run() {  
        return as_xpr('a');  
    }  
};
```



build_my_regex

```
struct r_a {  
  
    static decltype(as_xpr('a')) run() {  
        return as_xpr('a');  
    }  
};
```

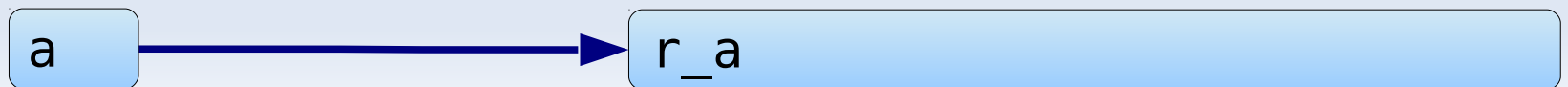
```
#define RUN(...) \  
    static decltype((__VA_ARGS__)) run() { \  
        return (__VA_ARGS__); \  
    }
```



build_my_regex

```
struct r_a {  
  
    RUN(as_xpr('a'          ))  
  
};
```

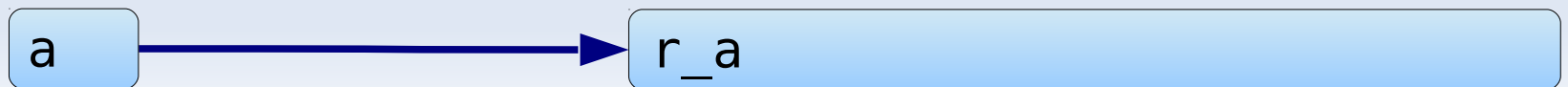
```
#define RUN(...) \  
    static decltype((__VA_ARGS__)) run() { \  
        return (__VA_ARGS__); \  
    }
```



build_my_regex

```
struct r_a {  
    typedef r_a    type;  
  
    RUN(as_xpr('a'        ))  
  
};
```

```
#define RUN(...) \  
    static decltype((__VA_ARGS__)) run() { \  
        return (__VA_ARGS__); \  
    }
```



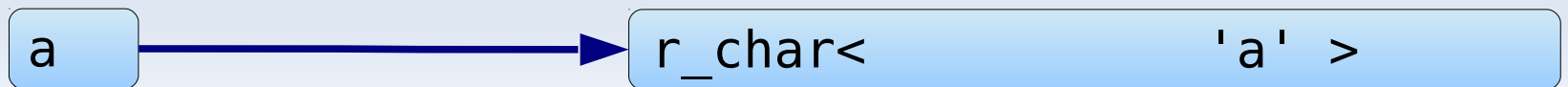
build_my_regex

```
template <char C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C          ))

};
```

```
#define RUN(...) \
    static decltype((__VA_ARGS__)) run() { \
        return (__VA_ARGS__); \
    }
```



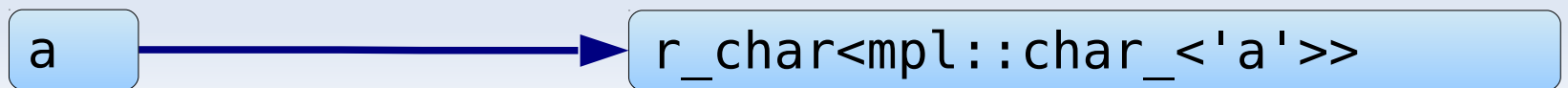
build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::value    ))

};
```

```
#define RUN(...) \
    static decltype((__VA_ARGS__)) run() { \
        return (__VA_ARGS__); \
    }
```

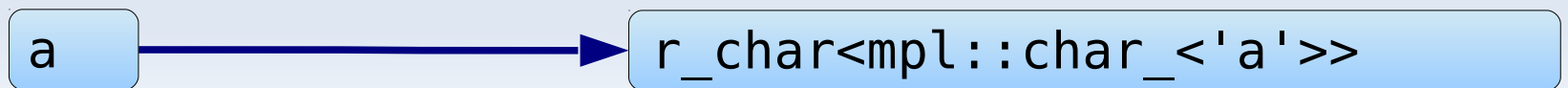


build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))
};
```

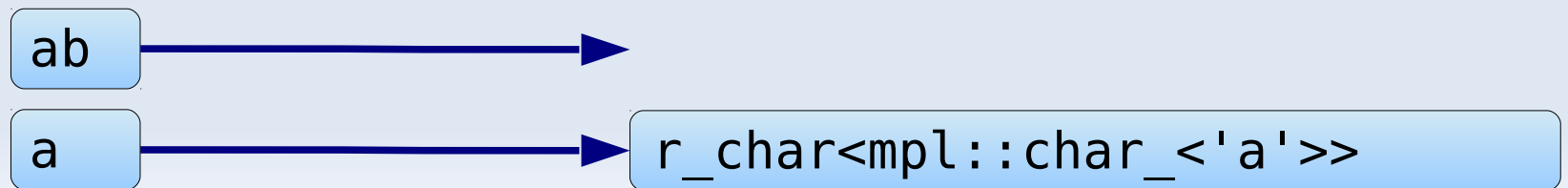
```
#define RUN(...) \
    static decltype((__VA_ARGS__)) run() { \
        return (__VA_ARGS__); \
    }
```



build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))
};
```

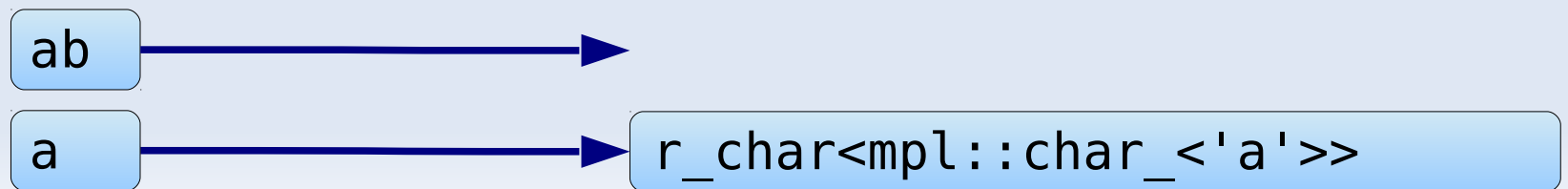


build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))
};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
};
```



build_my_regex

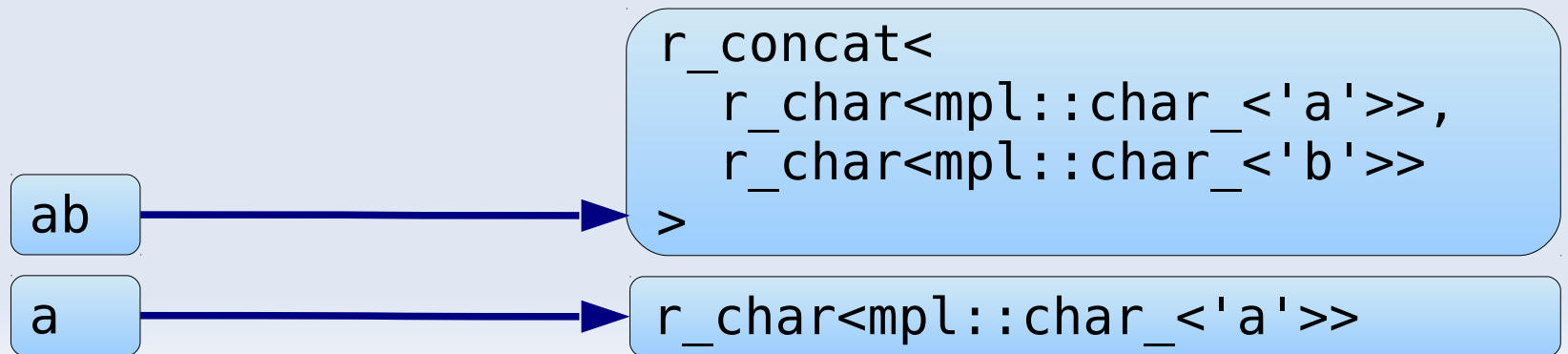
```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))

};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;

};
```



build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))

};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(
    )
};
```

ab



```
r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
>
```

a



```
r_char<mpl::char_<'a'>>
```

build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))

};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(A::run() )
};
```

ab



```
r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
>
```

a



```
r_char<mpl::char_<'a'>>
```

build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))

};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(A::run()          B::run()          )
};
```

ab



```
r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
>
```

a



```
r_char<mpl::char_<'a'>>
```

build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))

};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(A::run()          >> B::run()          )
};
```

ab



```
r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
>
```

a



```
r_char<mpl::char_<'a'>>
```

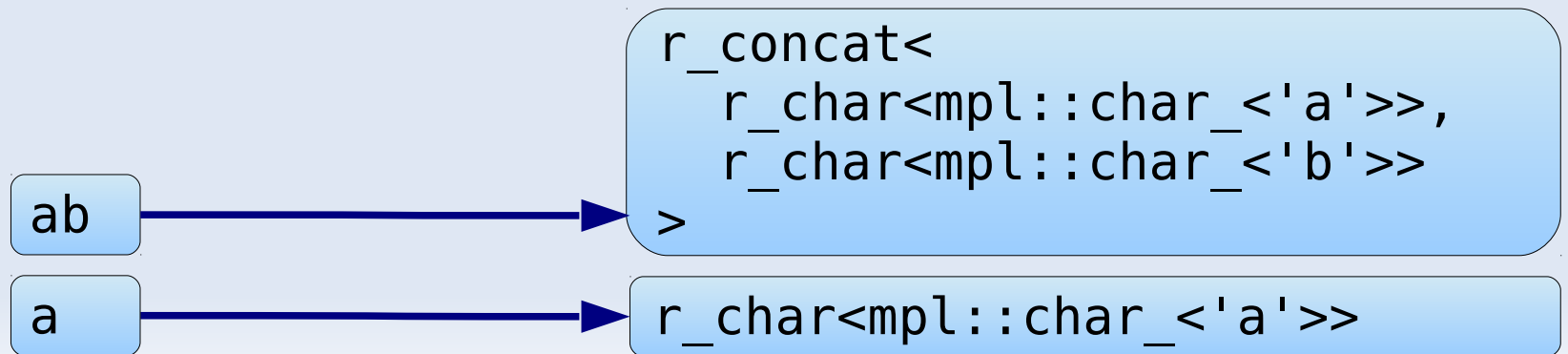
build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))

};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(A::type::run() >> B::type::run())
};
```



build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))
};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(A::type::run() >> B::type::run())
};
```

abc

ab

a

r_concat<
 r_char<mpl::char_<'a'>>,&br/> r_char<mpl::char_<'b'>>
>

r_char<mpl::char_<'a'>>

build_my_regex

```
template <class C>
struct r_char {
    typedef r_char type;

    RUN(as_xpr(C::type::value))
};
```

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(A::type::run() >> B::type::run());
};
```

abc

```
r_concat<
  r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
  >,
  r_char<mpl::char_<'c'>>
>
```

ab

```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

a

```
r_char<mpl::char_<'a'>>
```


build_my_regex

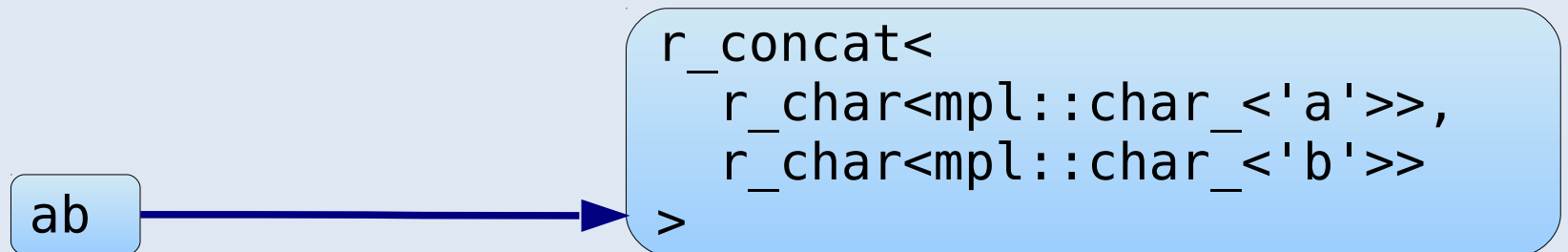
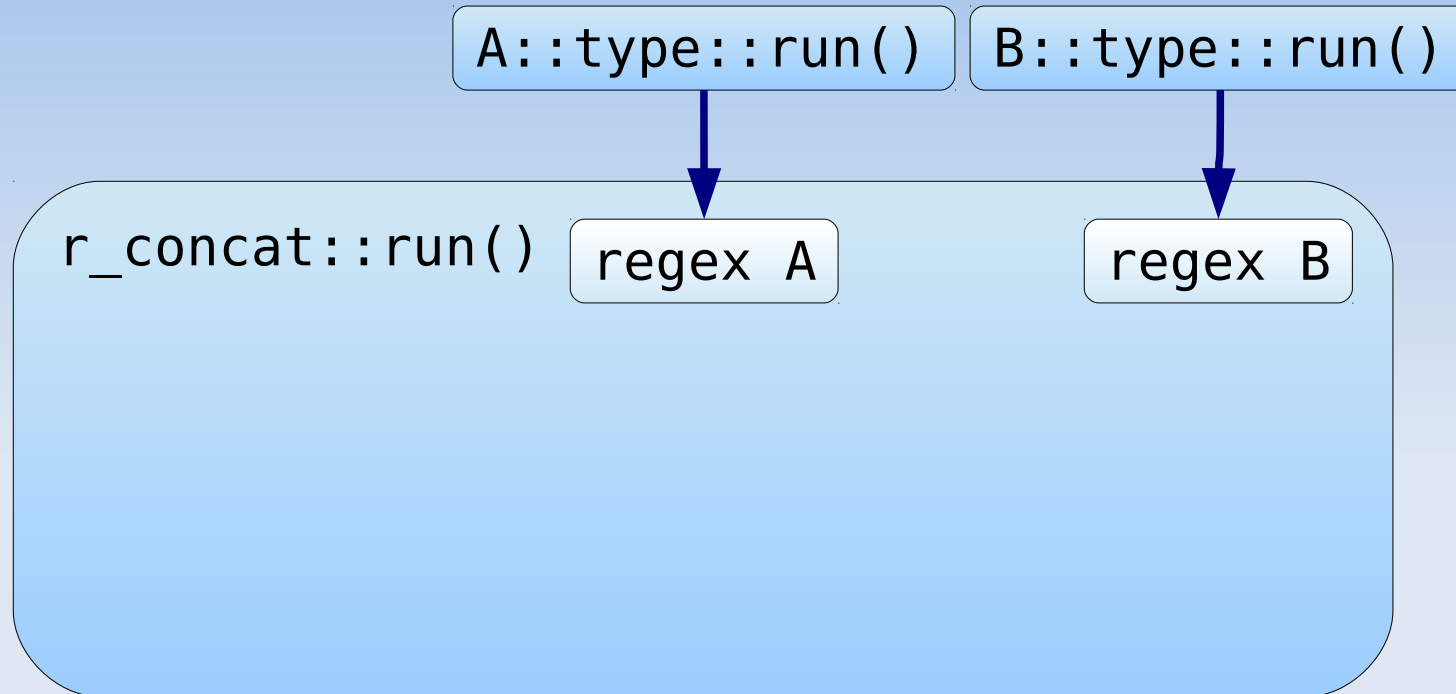
```
r_concat::run()
```

ab

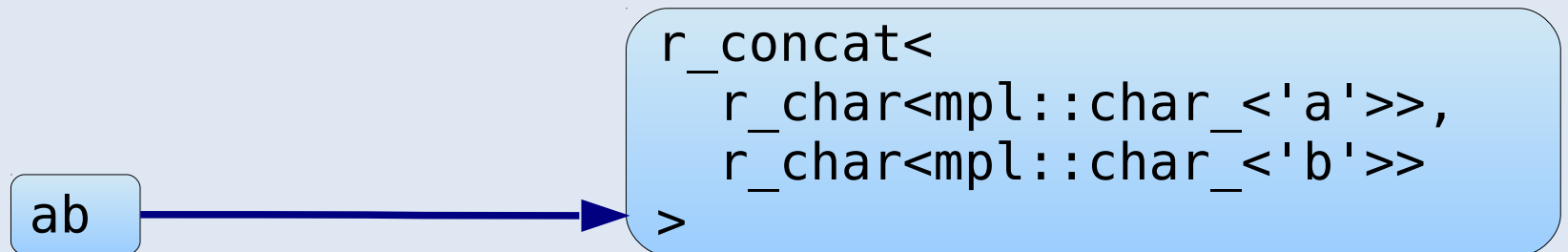
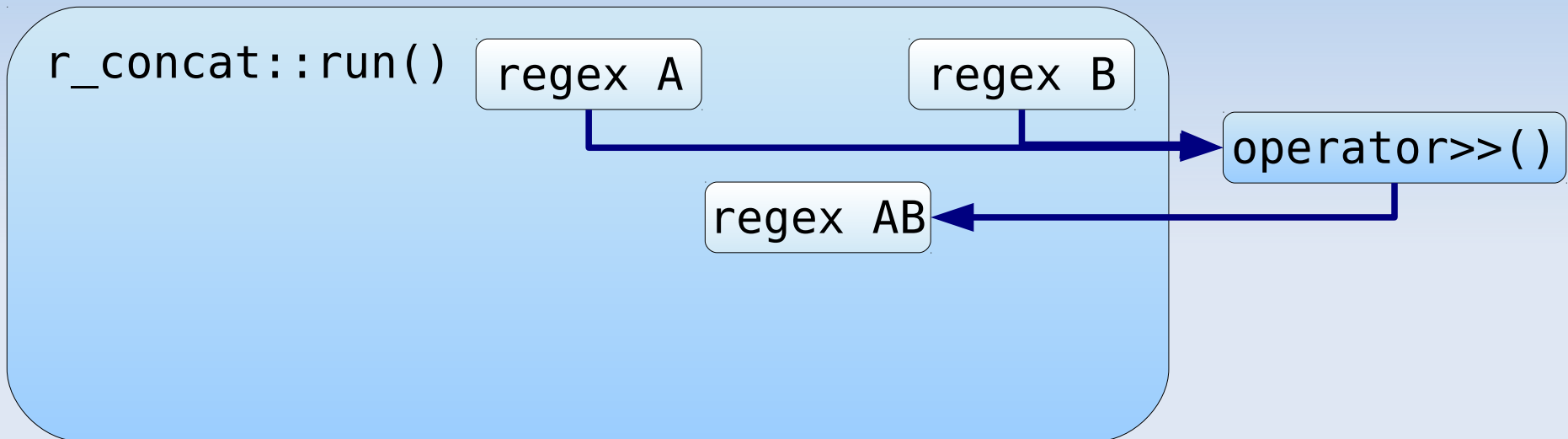


```
r_concat<  
  r_char<mpl::char_<'a'>>,&br/>  r_char<mpl::char_<'b'>>  
>
```

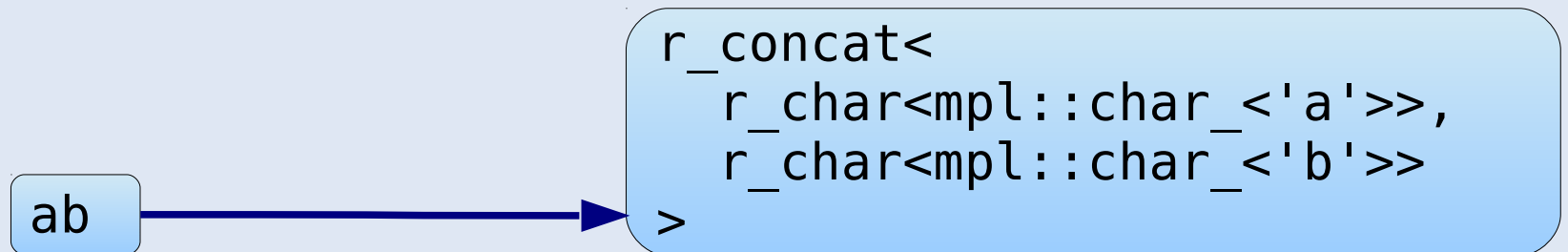
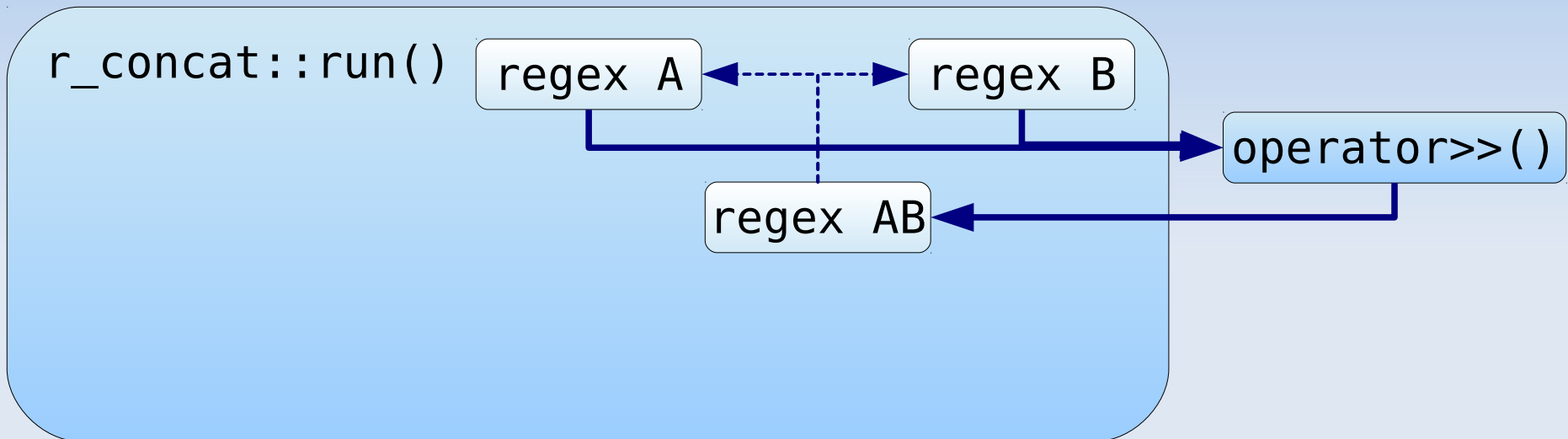
build_my_regex



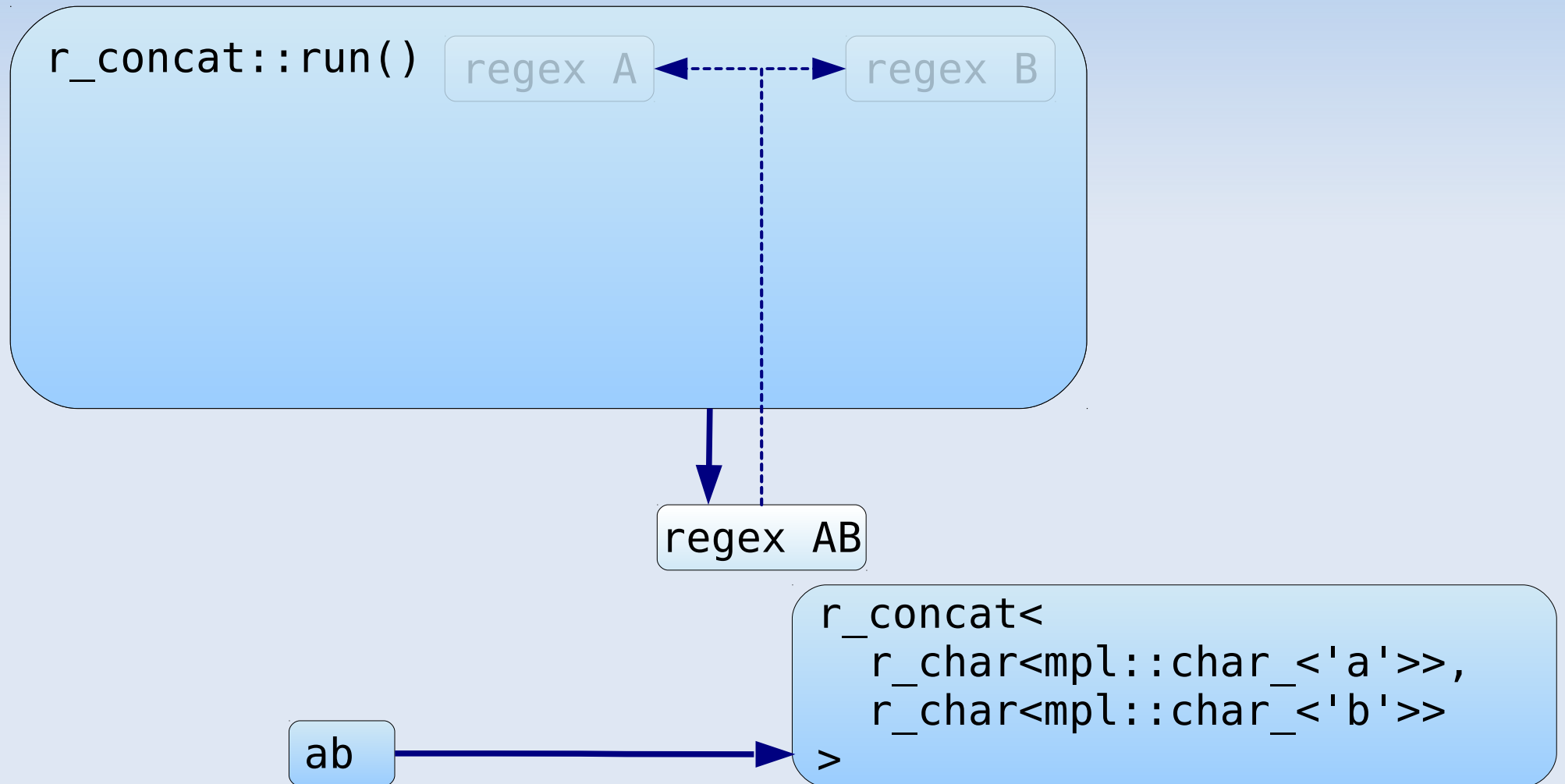
build_my_regex



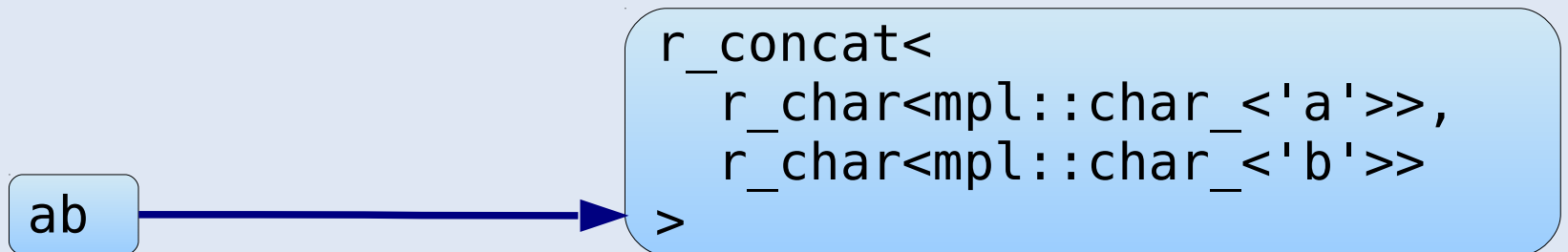
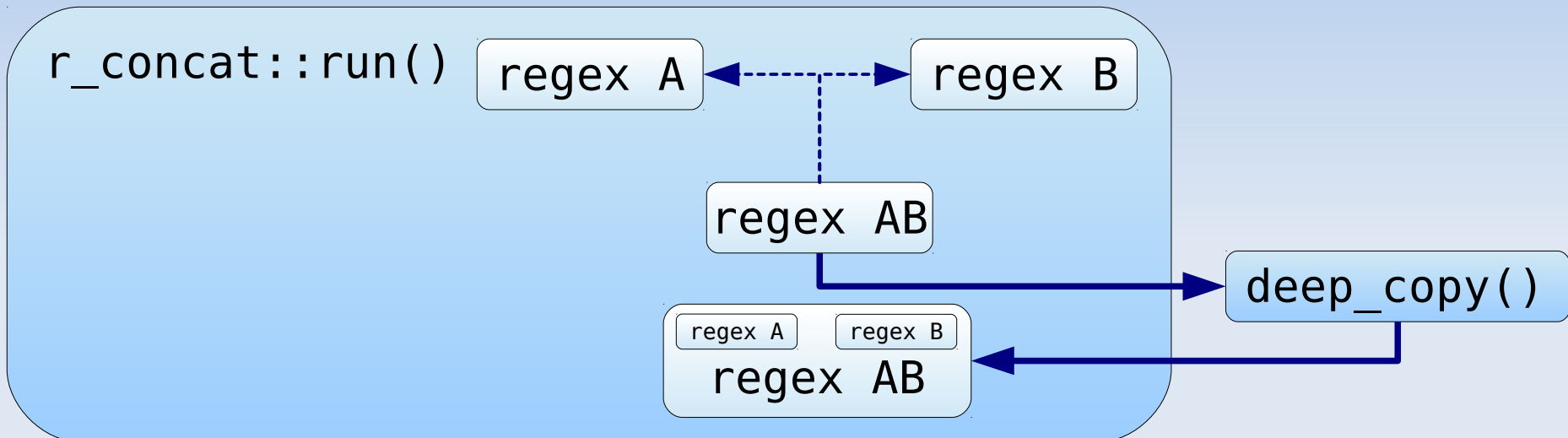
build_my_regex



build_my_regex

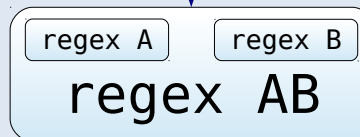


build_my_regex



build_my_regex

```
r_concat::run()
```



ab

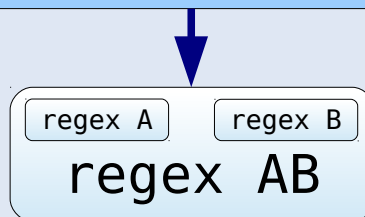


```
r_concat<  
  r_char<mpl::char_<'a'>>,  
  r_char<mpl::char_<'b'>>  
>
```

build_my_regex

`r_concat::`

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(
        A::type::run() >> B::type::run() )
};
```



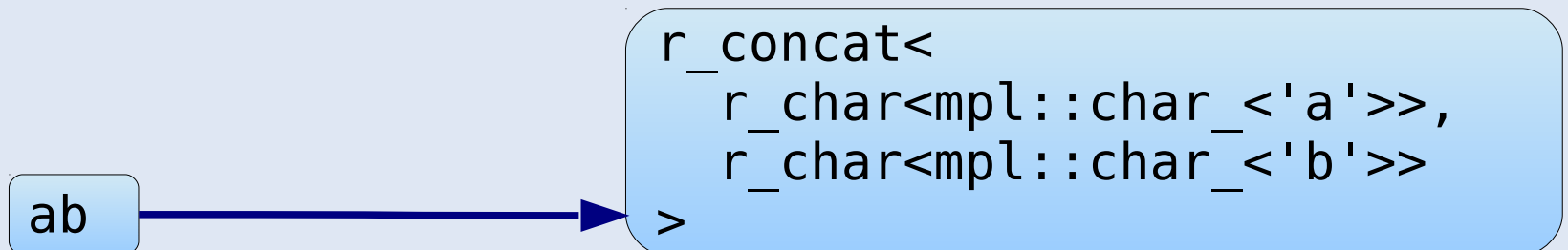
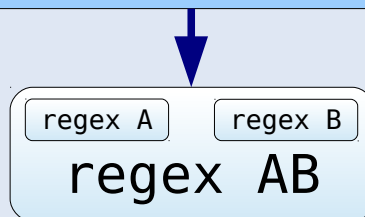
`ab`

```
r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
>
```


build_my_regex

`r_concat::`

```
template <class A, class B>
struct r_concat {
    typedef r_concat type;
    RUN(deep_copy(A::type::run() >> B::type::run()))
};
```



Lab 3

- Implement the types representing regular expressions
 - **r_empty**: empty regular expression
 - **r_dot**: the `.` regular expression
 - **r_star**: the `*` regular expression
 - **r_concat**: the concatenation of two regular expressions
 - **r_char**: match one specific character

C++ template metafunction

Argument list

Name

Body

C++ template metafunction

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

Argument list

Name

Body

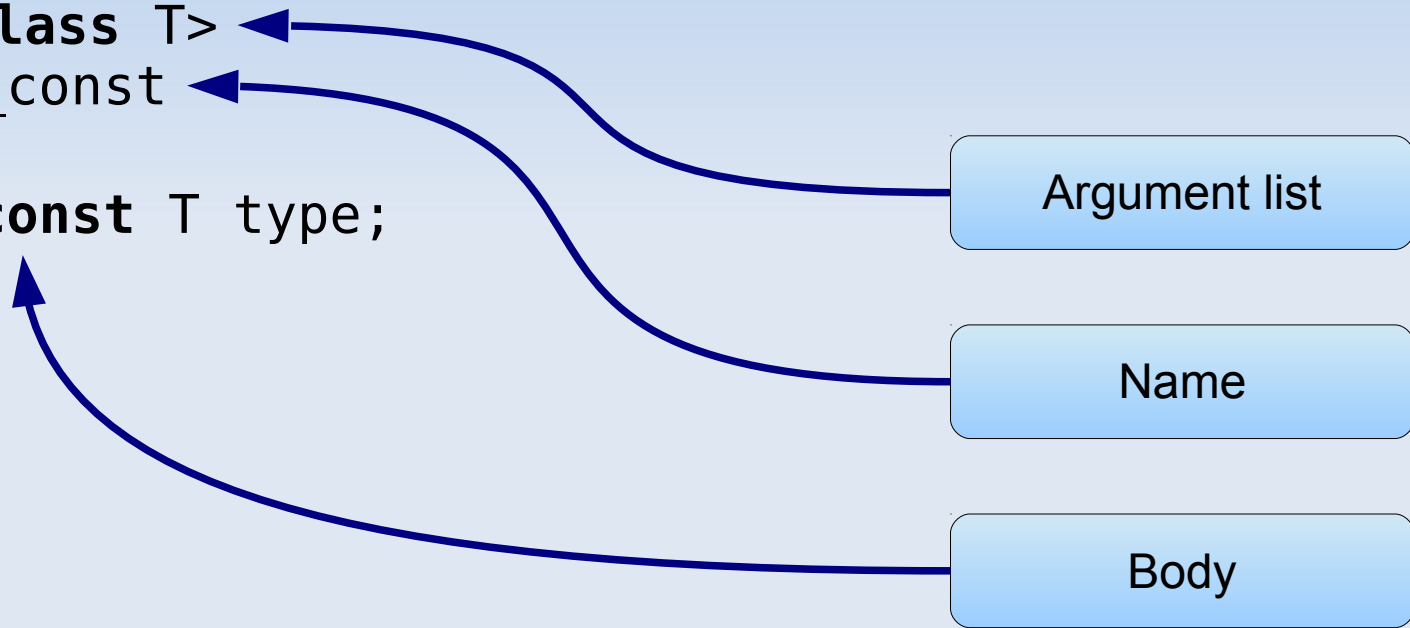
C++ template metafunction

```
template <class T>  
struct add_const  
{  
    typedef const T type;  
};
```

Argument list

Name

Body



C++ template metafunction

```
template <class T>  
struct add_const  
{  
    typedef const T type;  
};
```

Argument list

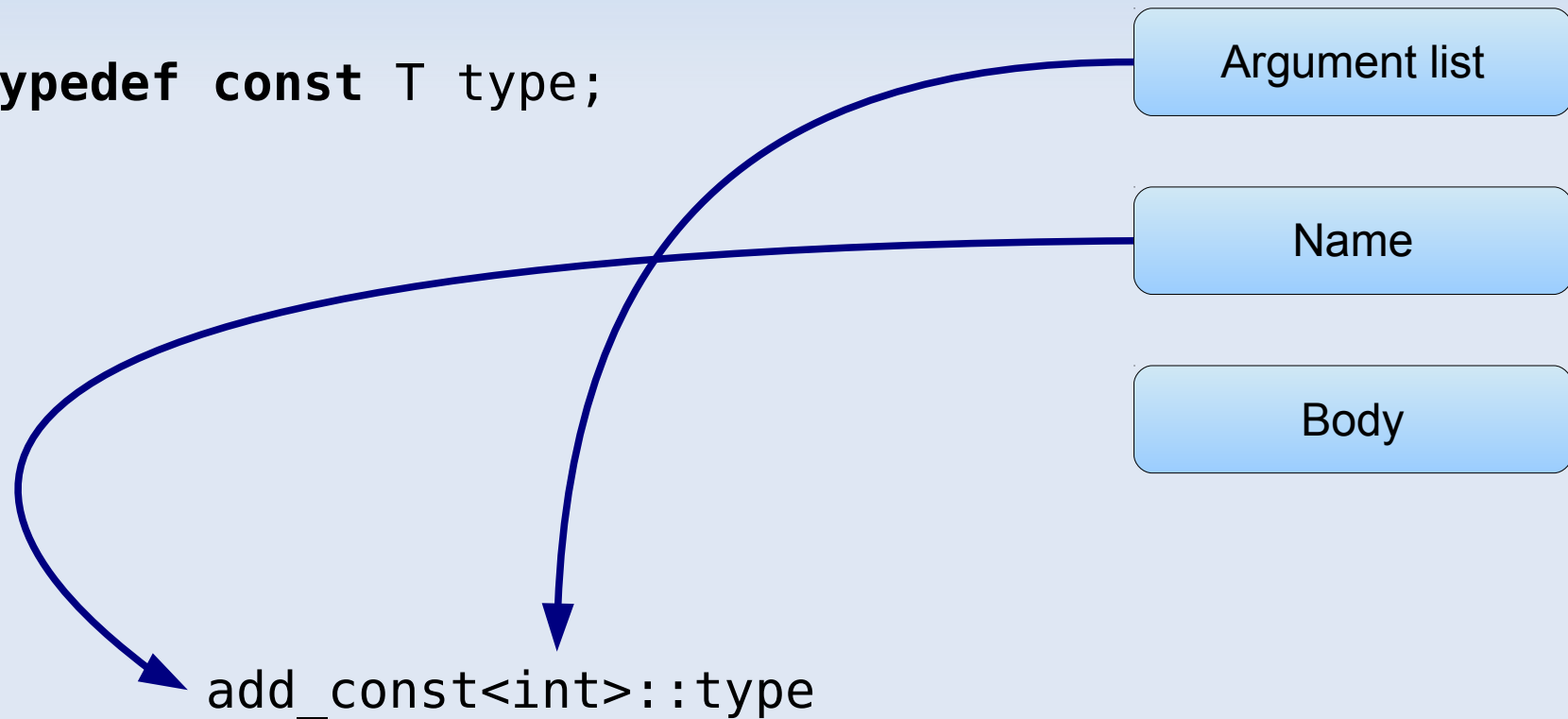
Name

Body

`add_const<int>::type`

C++ template metafunction

```
template <class T>  
struct add_const  
{  
    typedef const T type;  
};
```



C++ template metafunction

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

```
template <class T>
struct add_volatile
{
    typedef volatile T type;
};
```


C++ template metafunction

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

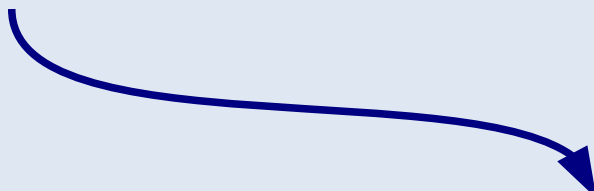
```
template <class T>
struct add_volatile
{
    typedef volatile T type;
};
```

```
template <class T>
struct add_cv
{
    typedef const volatile T type;
};
```

C++ template metafunction

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

```
template <class T>
struct add_volatile
{
    typedef volatile T type;
};
```



```
template <class T>
struct add_cv
{
    typedef

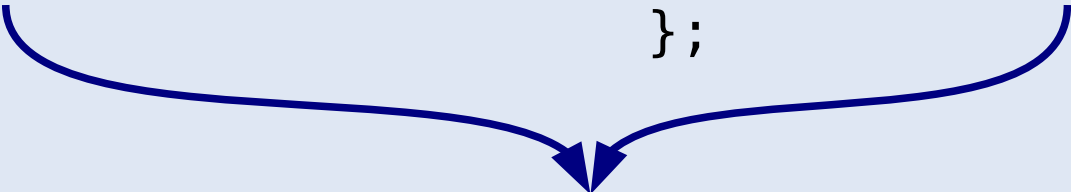
        typename add_const<T>::type

    type;
};
```

C++ template metafunction

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

```
template <class T>
struct add_volatile
{
    typedef volatile T type;
};
```



```
template <class T>
struct add_cv
{
    typedef
        typename add_volatile<
            typename add_const<T>::type
        >::type
        type;
};
```

C++ template metafunction

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

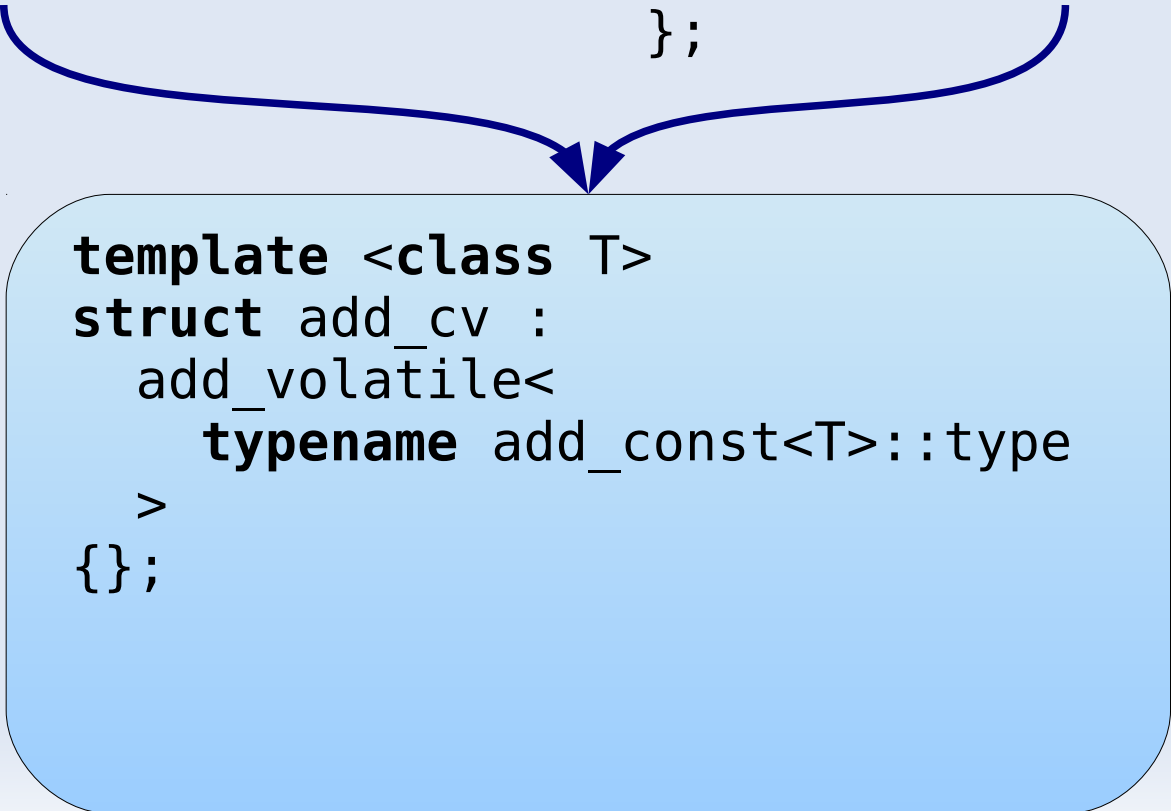
```
template <class T>
struct add_volatile
{
    typedef volatile T type;
};
```

```
template <class T>
struct add_cv :
    add_volatile<
        typename add_const<T>::type
    >
{
    typedef
    type;
};
```

C++ template metafunction

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

```
template <class T>
struct add_volatile
{
    typedef volatile T type;
};
```



The diagram illustrates the relationship between the `add_cv` template and its base templates. Two blue curved arrows originate from the `typedef const T type;` line in the `add_const` struct and the `typedef volatile T type;` line in the `add_volatile` struct. These arrows point towards the `add_cv` struct, which is highlighted in a light blue rounded rectangle. The `add_cv` struct is defined as a specialization of `add_volatile` with a `typedef` that inherits from `add_const`.

```
template <class T>
struct add_cv :
    add_volatile<
        typename add_const<T>::type
    >
{};
```

Lab 4

- Write a template metafunction called `beginning_and_end`
 - It has one argument (which is expected to be a string)
 - Returns a pair of characters: the first and the last character of the string
 - Eg. "Hello" \rightarrow ('h', 'o')
- Make use of Boost.MPL
 - `boost::mpl::pair`
 - `boost::mpl::front`
 - `boost::mpl::back`

Higher order functions

```
template <class T>
struct add_const
{
    typedef const T type;
};
```

add_const<int>::type

Higher order functions

```
struct add_const
{
    template <class T>
    struct add_const
    {
        typedef const T type;
    };
};
```

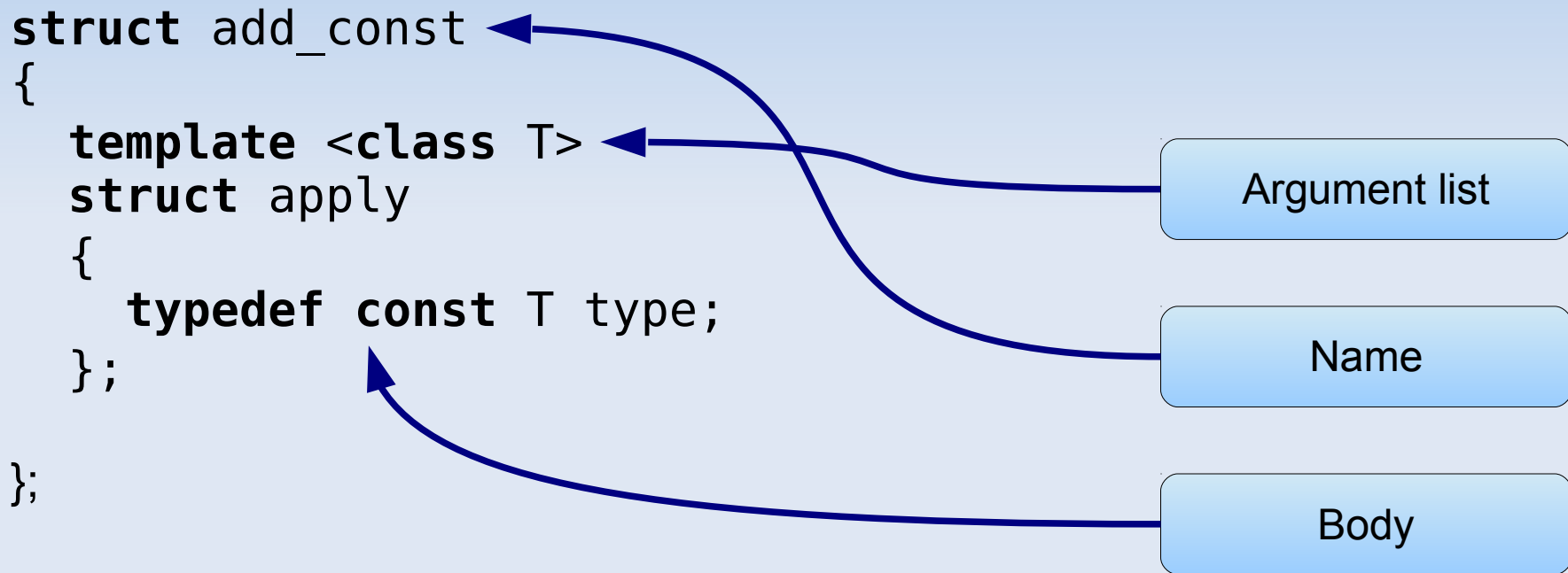
add_const::add_const<int>::type

Higher order functions

```
struct add_const
{
    template <class T>
    struct apply
    {
        typedef const T type;
    };
};
```

add_const::apply<int>::type

Template metafunction class



`add_const::apply<int>::type`

Template metafunction class

```
struct add_const  
{  
    template <class T>  
    struct apply  
    {  
        typedef const T type;  
    };  
};
```

Argument list

Name

Body

`add_const::apply<int>::type`

Template metafunction class

```
struct add_const
{
    template <class T>
    struct apply
    {
        typedef const T type;
    };
    typedef add_const type;
};
```

Argument list

Name

Body

`add_const::apply<int>::type`

```
graph LR; Name[Name] --- AC[add_const]; ArgList[Argument list] --- Int[<int>]; Body[Body] --- Type[type];
```

Lab 5

- Turn `beginning_and_end` into a template metafunction class

The grammar

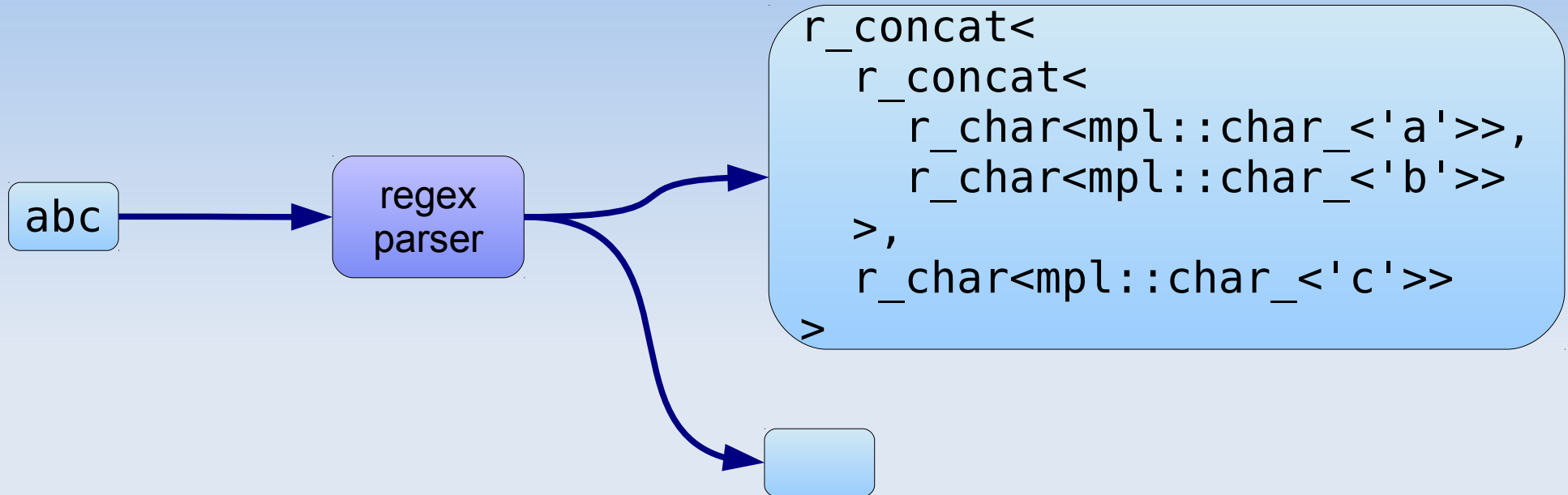
- We will support
 - letters and numbers (eg. **abc123**)
 - **.**
 - *****
 - brackets (eg. **(abc)***)

The grammar

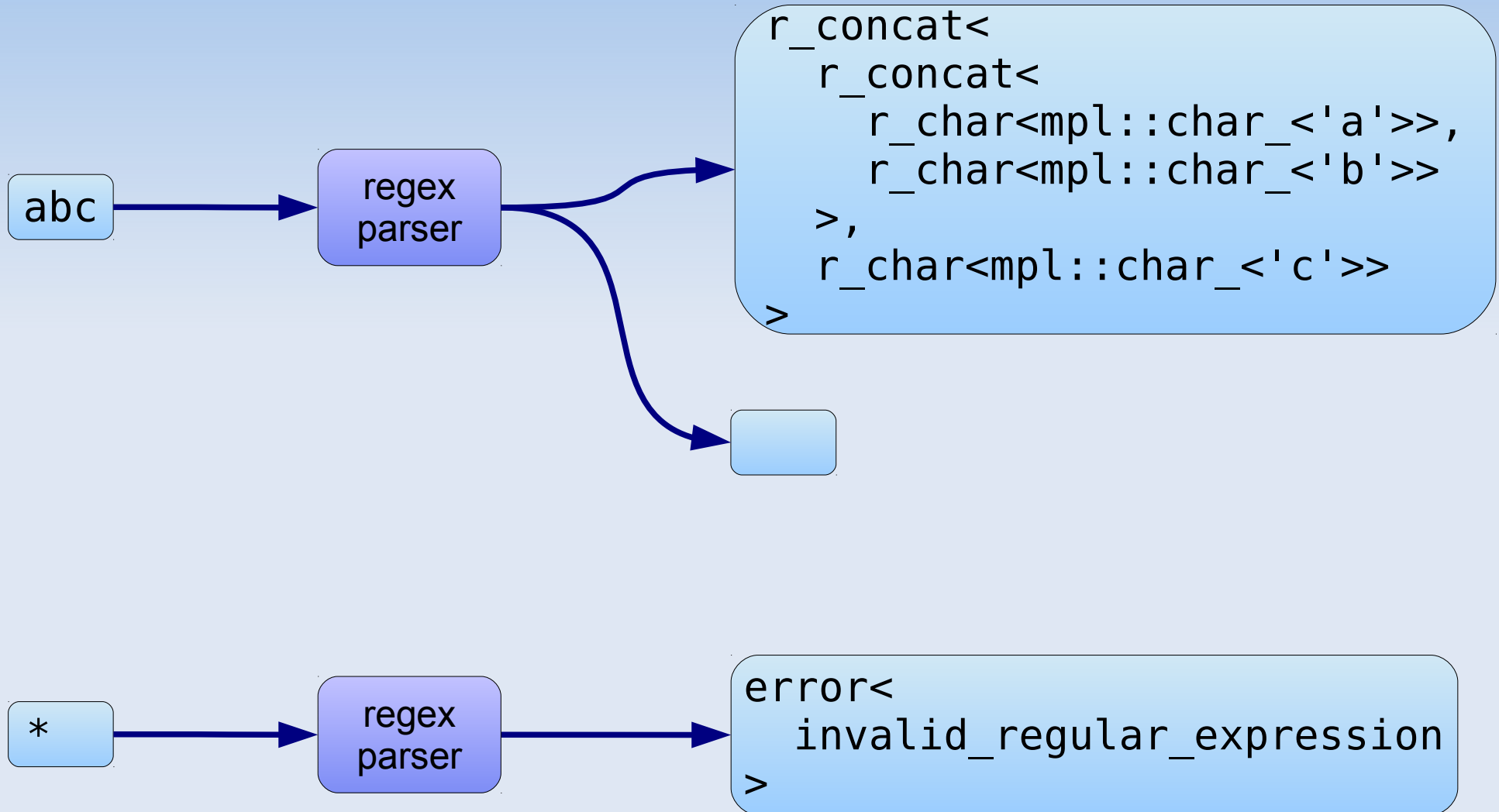
- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

```
reg_exp      ::= unary_item*
unary_item   ::= item '*'?
item         ::= any | bracket_exp | char_
any          ::= '.'
bracket_exp  ::= '(' reg_exp ')'
char_       ::= number | letter
number      ::= '0'..'9'
letter      ::= 'a'..'z' | 'A'..'Z'
```

Parsing regular expressions



Parsing regular expressions



letter ::= 'a'..'z' | 'A'..'Z'

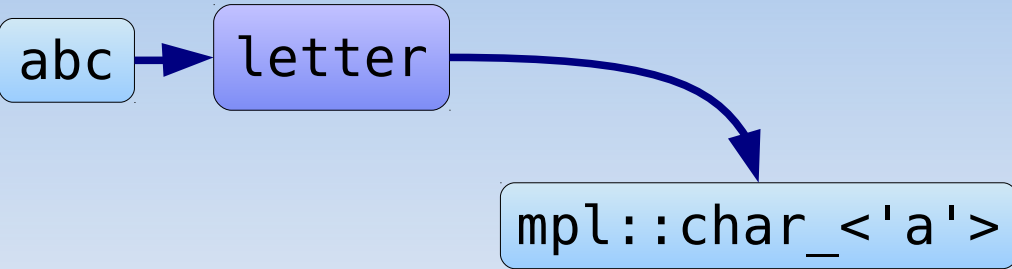
abc



letter

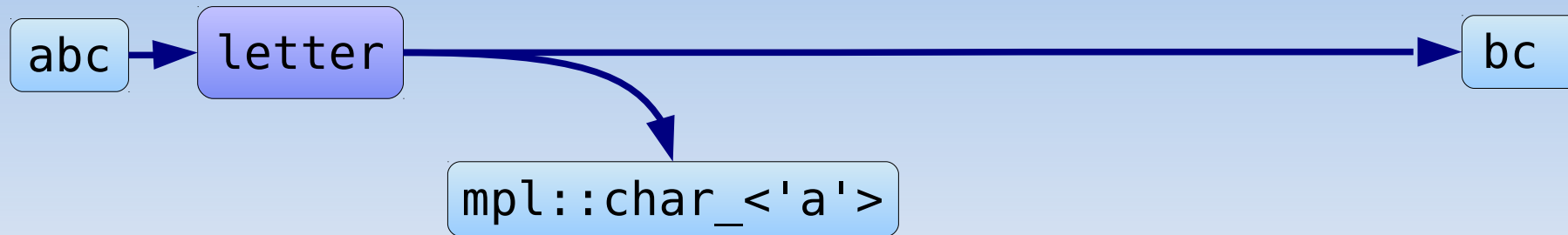
letter

letter ::= 'a'..'z' | 'A'..'Z'



letter

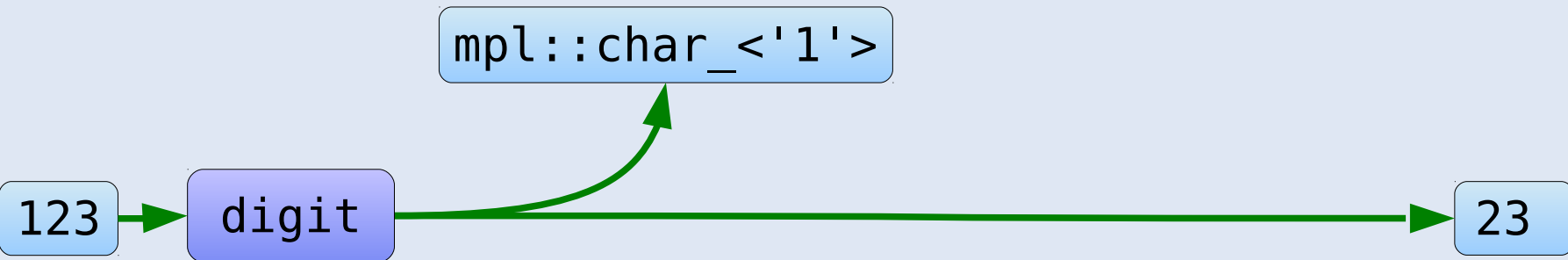
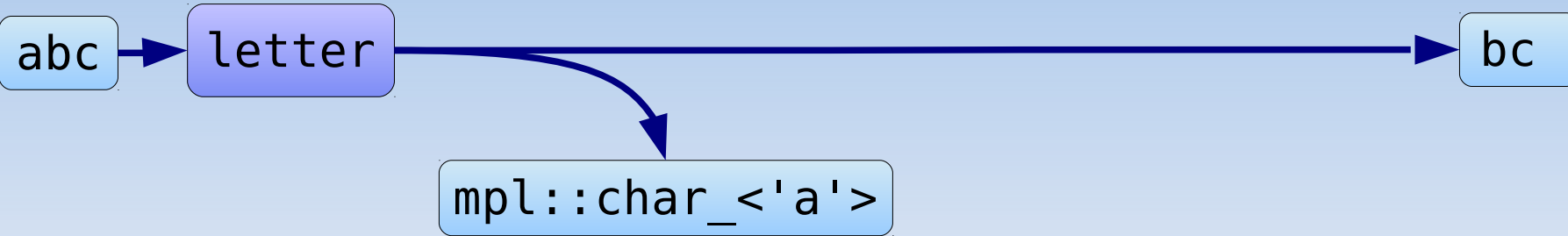
letter ::= 'a'..'z' | 'A'..'Z'



letter

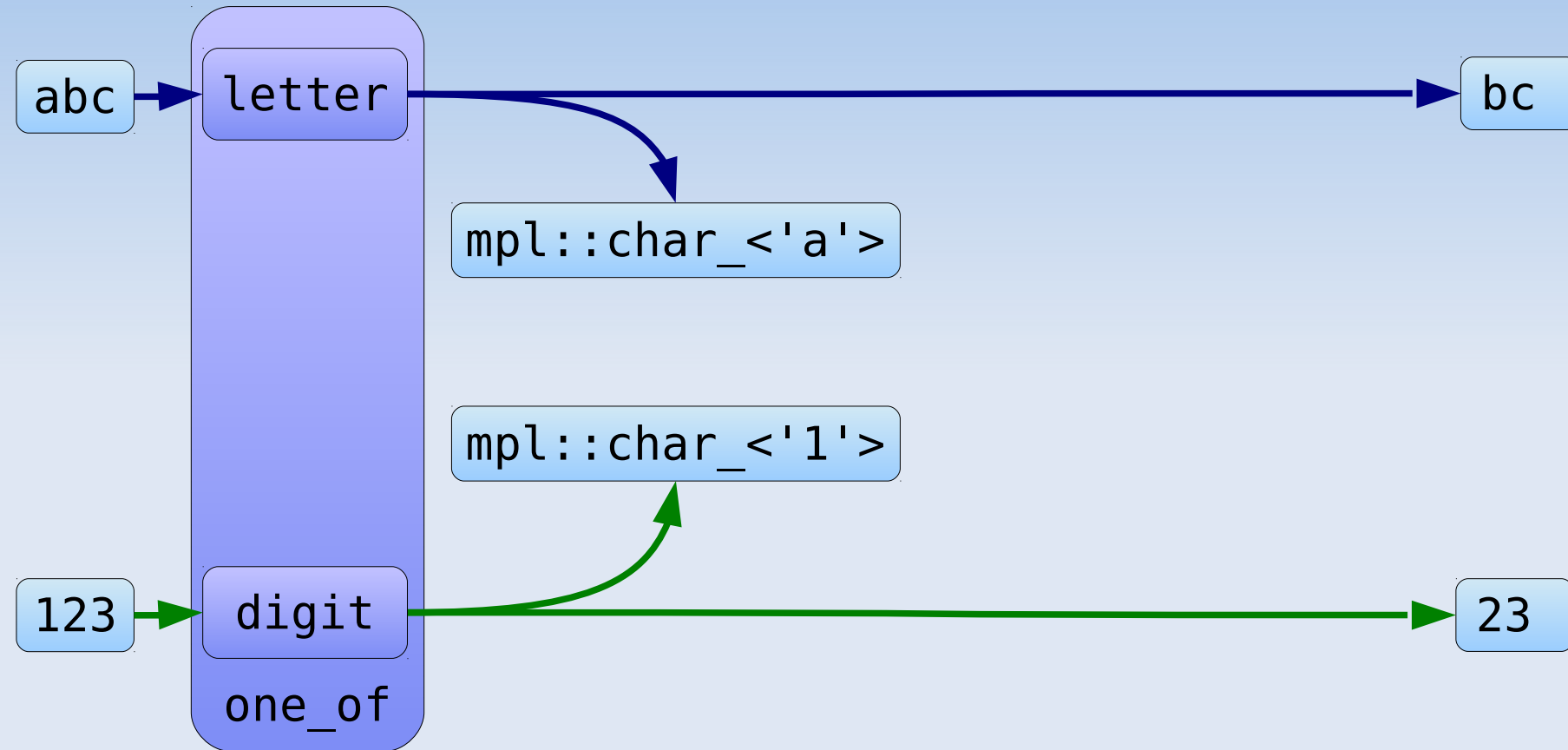
number ::= '0'..'9'

letter ::= 'a'..'z' | 'A'..'Z'



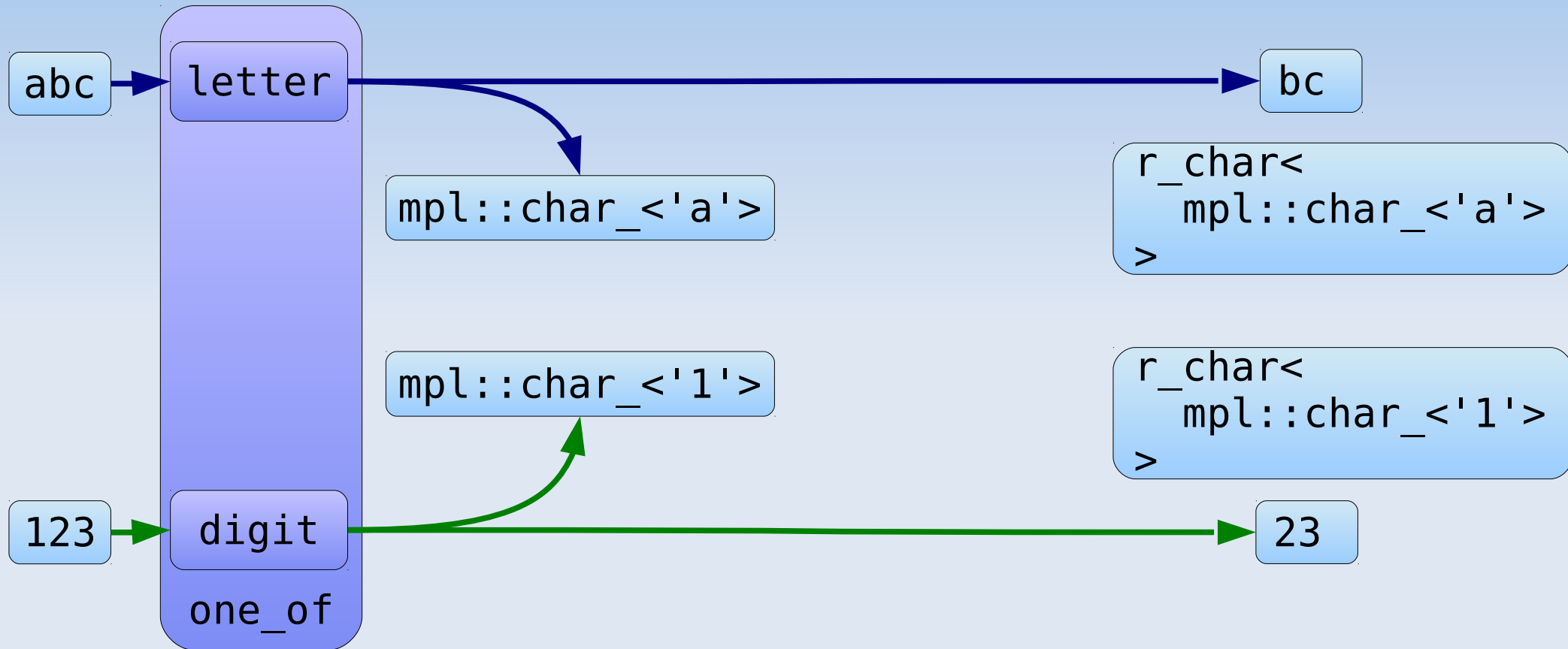
letter digit

`char_ ::= number | letter`
`number ::= '0'..'9'`
`letter ::= 'a'..'z' | 'A'..'Z'`



`one_of<letter, digit>`

`char_ ::= number | letter`
`number ::= '0'..'9'`
`letter ::= 'a'..'z' | 'A'..'Z'`

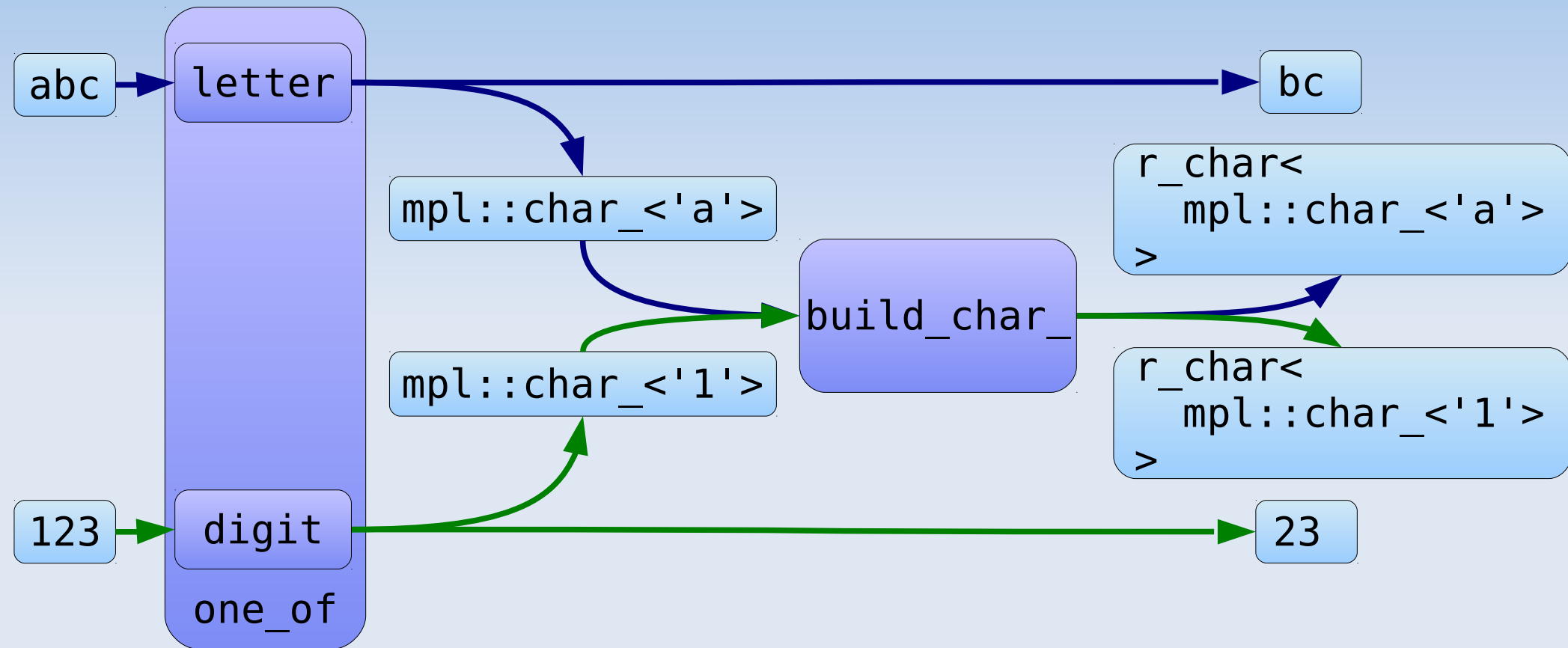


`one_of<letter, digit>`

```

char_ ::= number | letter
number ::= '0'..'9'
letter ::= 'a'..'z' | 'A'..'Z'

```



```

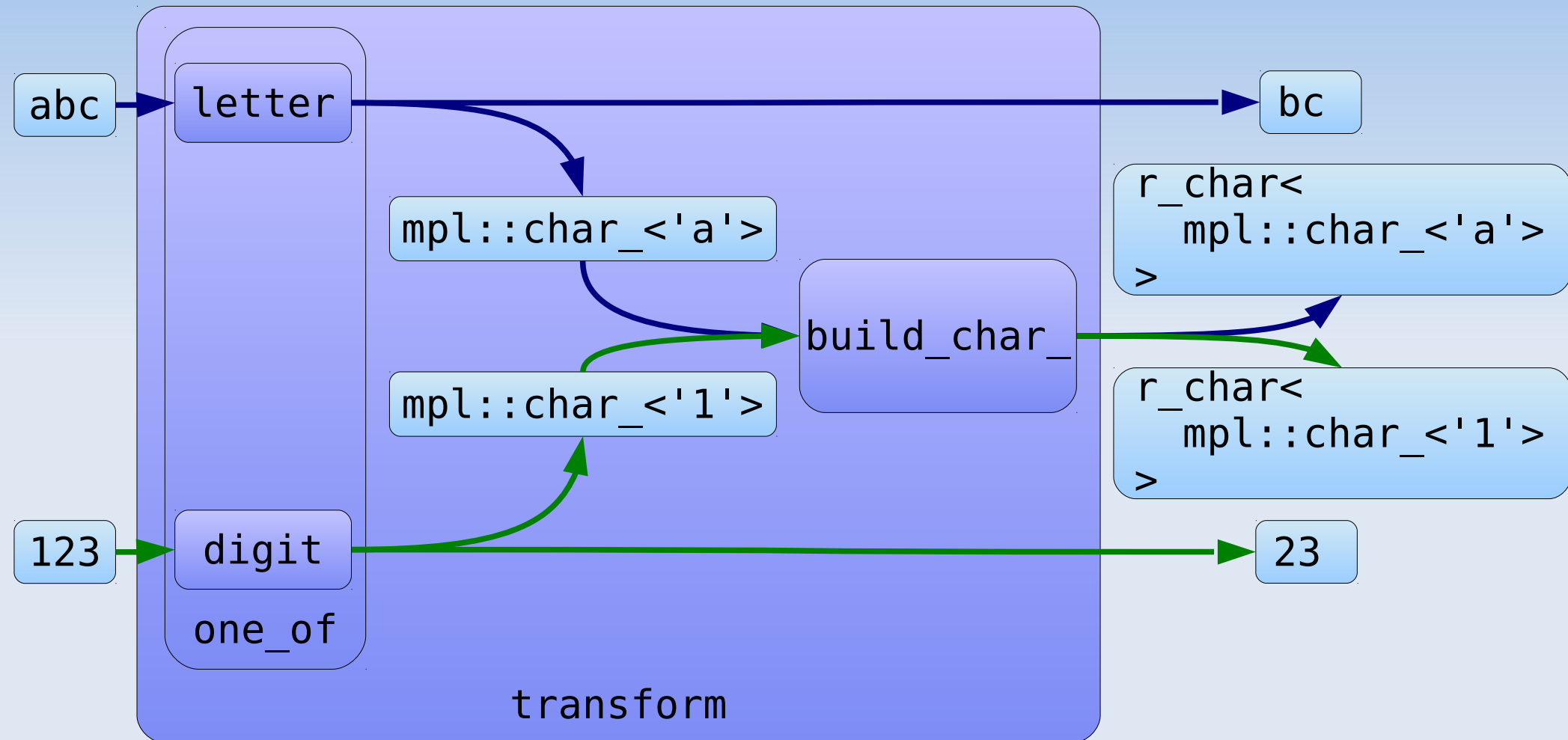
struct build_char_ {
    typedef build_char_ type;

    template <class T>
    struct apply : r_char<T> {};
};

```

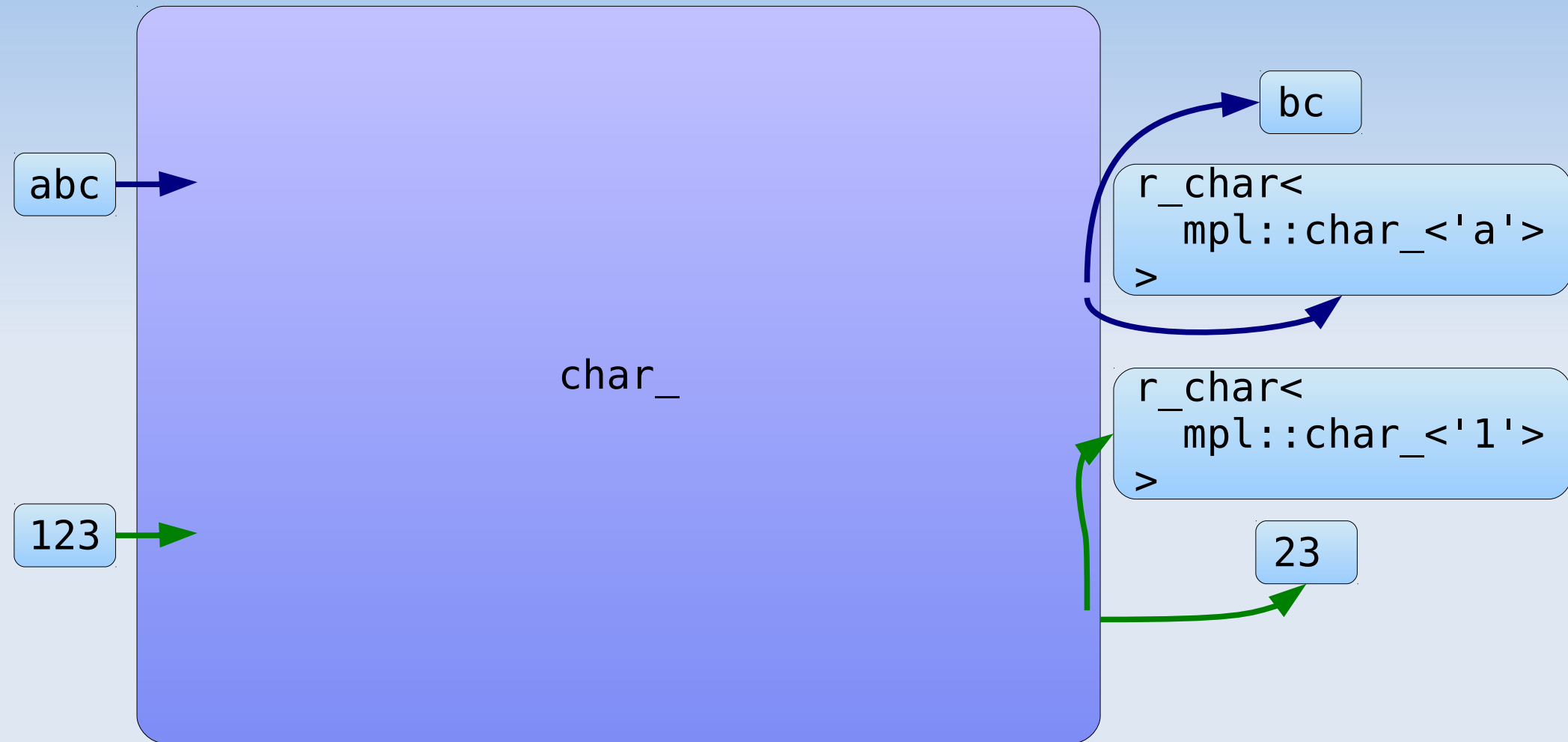
one_of<letter>

`char_ ::= number | letter`
`number ::= '0'..'9'`
`letter ::= 'a'..'z' | 'A'..'Z'`



`transform<one_of<letter, digit>, build_char_>`

```
char_ ::= number | letter  
number ::= '0'..'9'  
letter ::= 'a'..'z' | 'A'..'Z'
```



```
typedef transform<one_of<letter, digit>, build_char_> char_;
```

The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

```
reg_exp      ::= unary_item*  
unary_item   ::= item '*'?  
item         ::= any | bracket_exp | char_  
any          ::= '.'  
bracket_exp  ::= '(' reg_exp ')'  
char_        ::= number | letter ✓  
number       ::= '0'..'9' ✓  
letter       ::= 'a'..'z' | 'A'..'Z' ✓
```

The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

```
reg_exp      ::= unary_item*  
unary_item   ::= item '*'?  
item         ::= any | bracket_exp | char_  
any          ::= '.' ✓  
bracket_exp  ::= '(' reg_exp ')'  
char_        ::= number | letter ✓  
number       ::= '0'..'9' ✓  
letter       ::= 'a'..'z' | 'A'..'Z' ✓
```

The grammar

```
typedef one_of<any, char_> item;
```

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

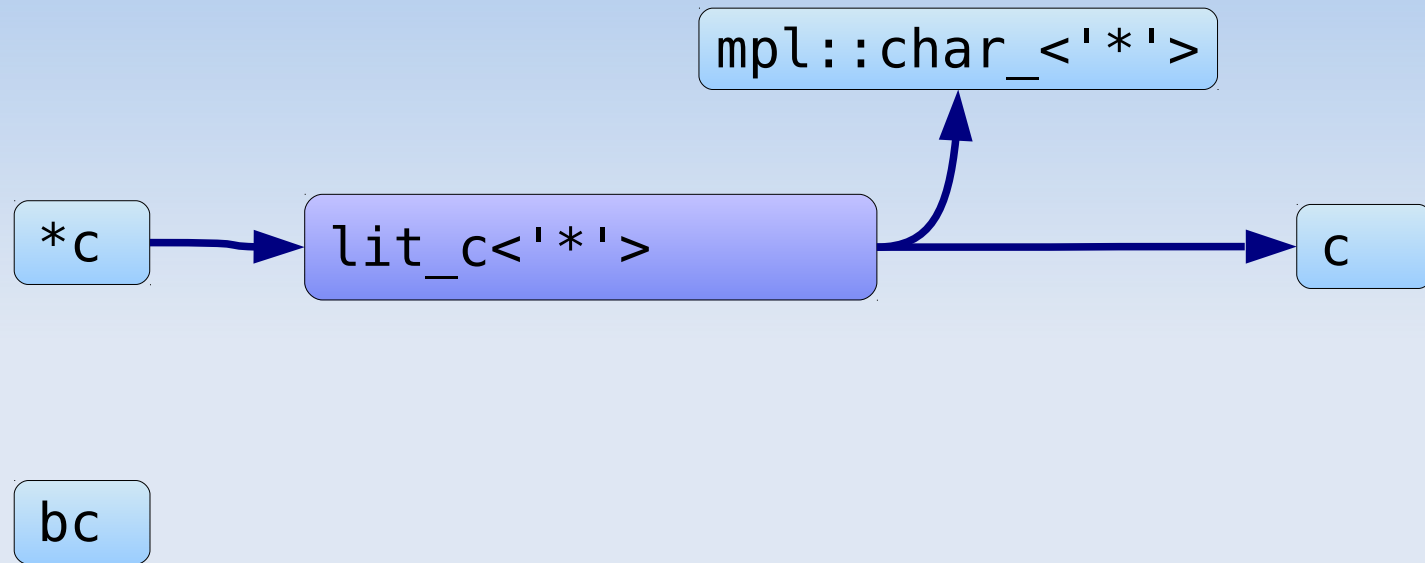
```
reg_exp      ::= unary_item*  
unary_item   ::= item '*'?  
item         ::= any | bracket_exp | char_  
any          ::= '.' ✓  
bracket_exp  ::= '(' reg_exp ')' ✓  
char_        ::= number | letter ✓  
number       ::= '0'..'9' ✓  
letter       ::= 'a'..'z' | 'A'..'Z' ✓
```

unary_item ::= item '*'?



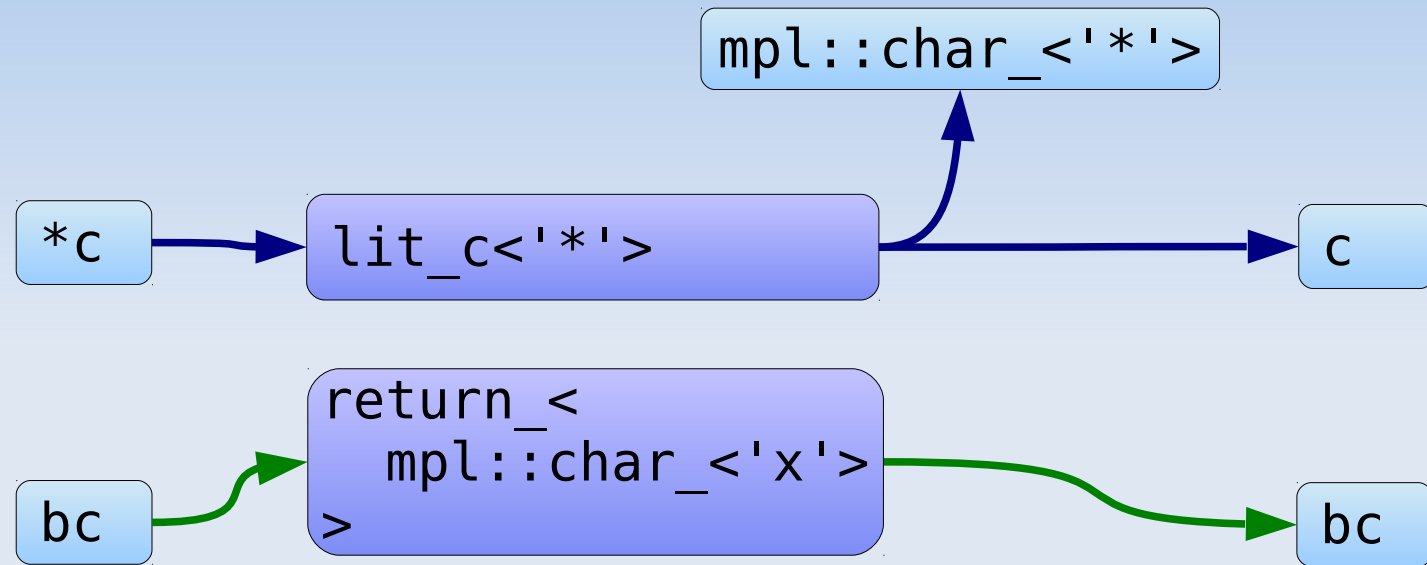
`lit_c<'*>`

unary_item ::= item '*'?



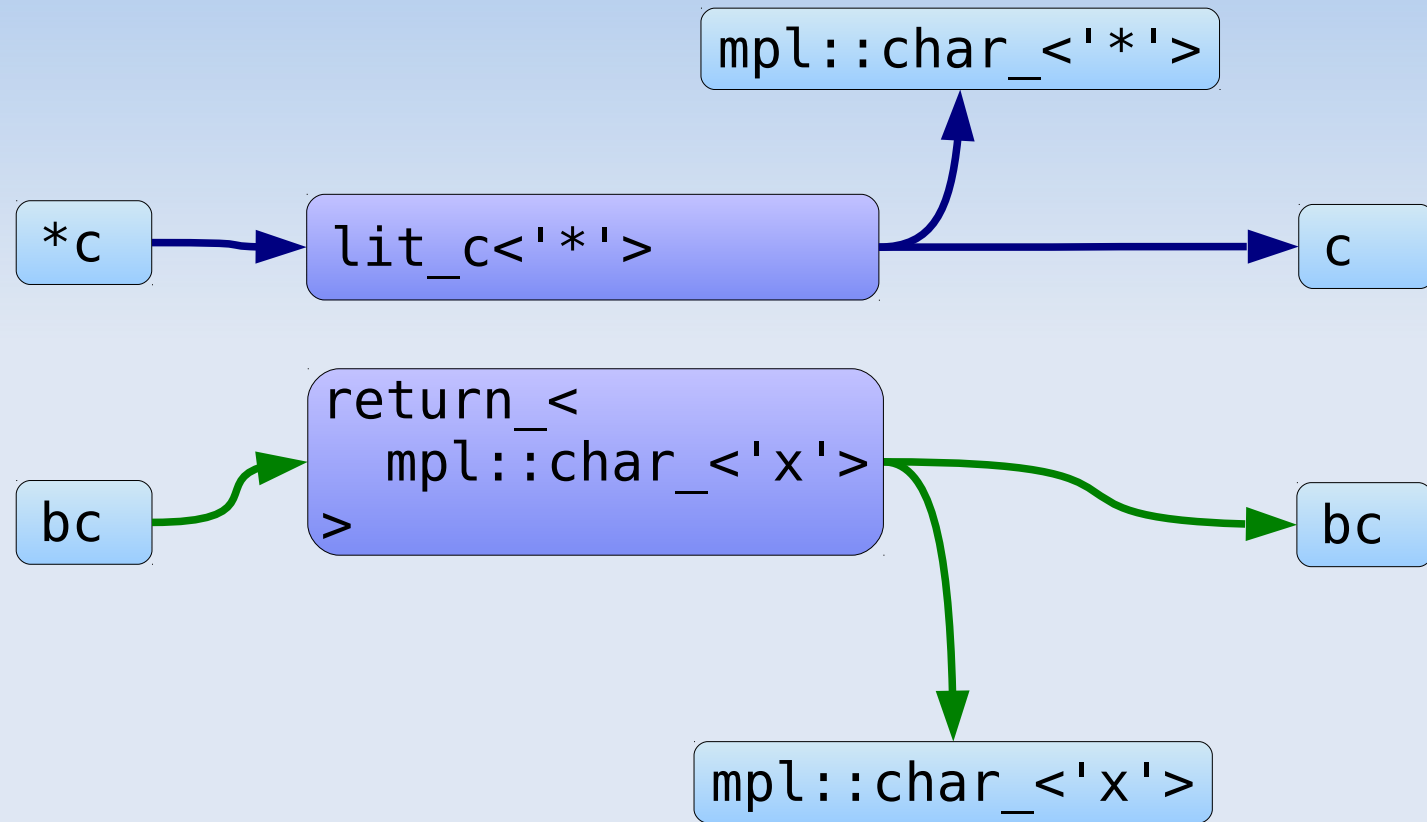
`lit_c<'*>`

unary_item ::= item '*'?



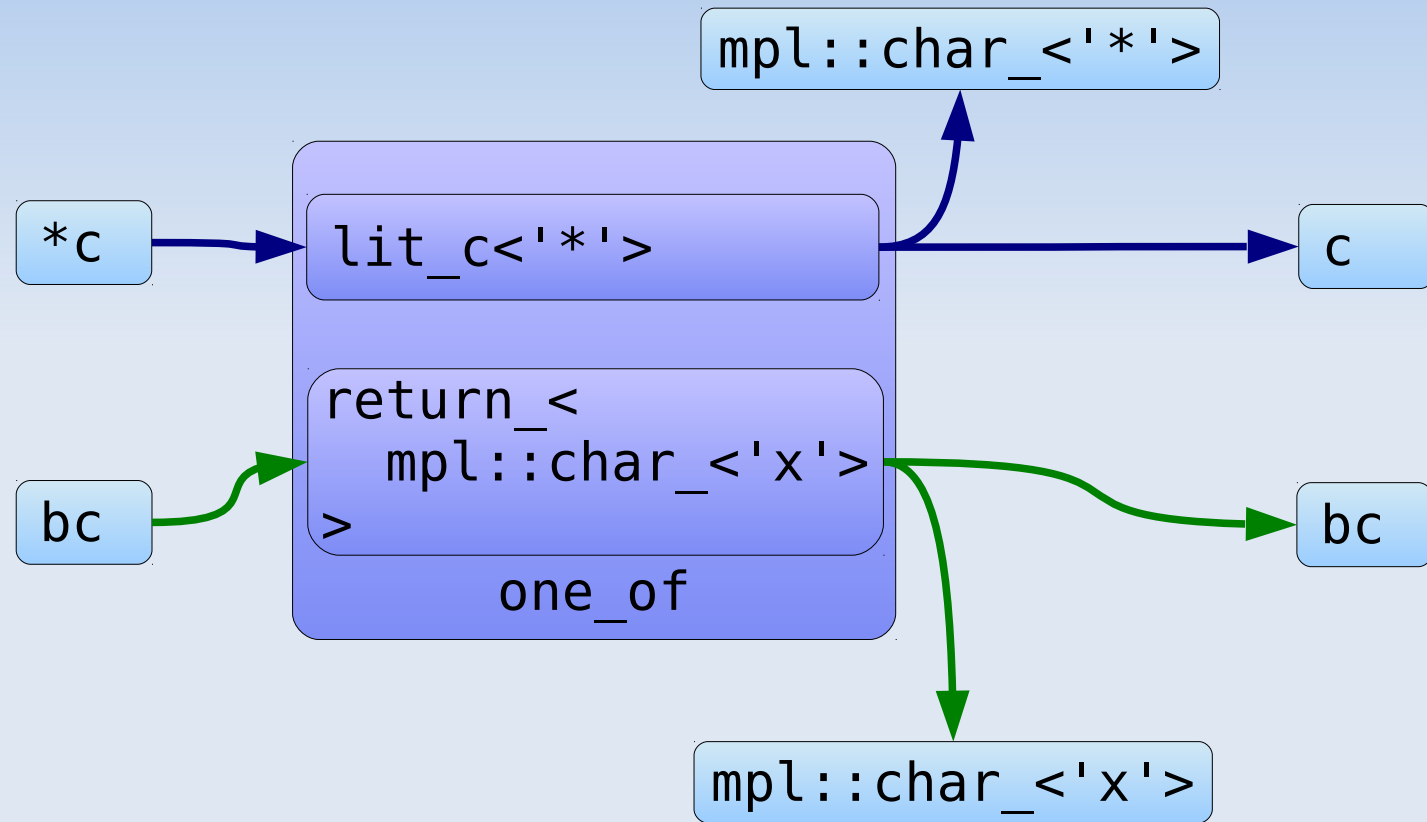
`lit_c<'*>` `return_<mpl::char_<'x'>>`

unary_item ::= item '*'?



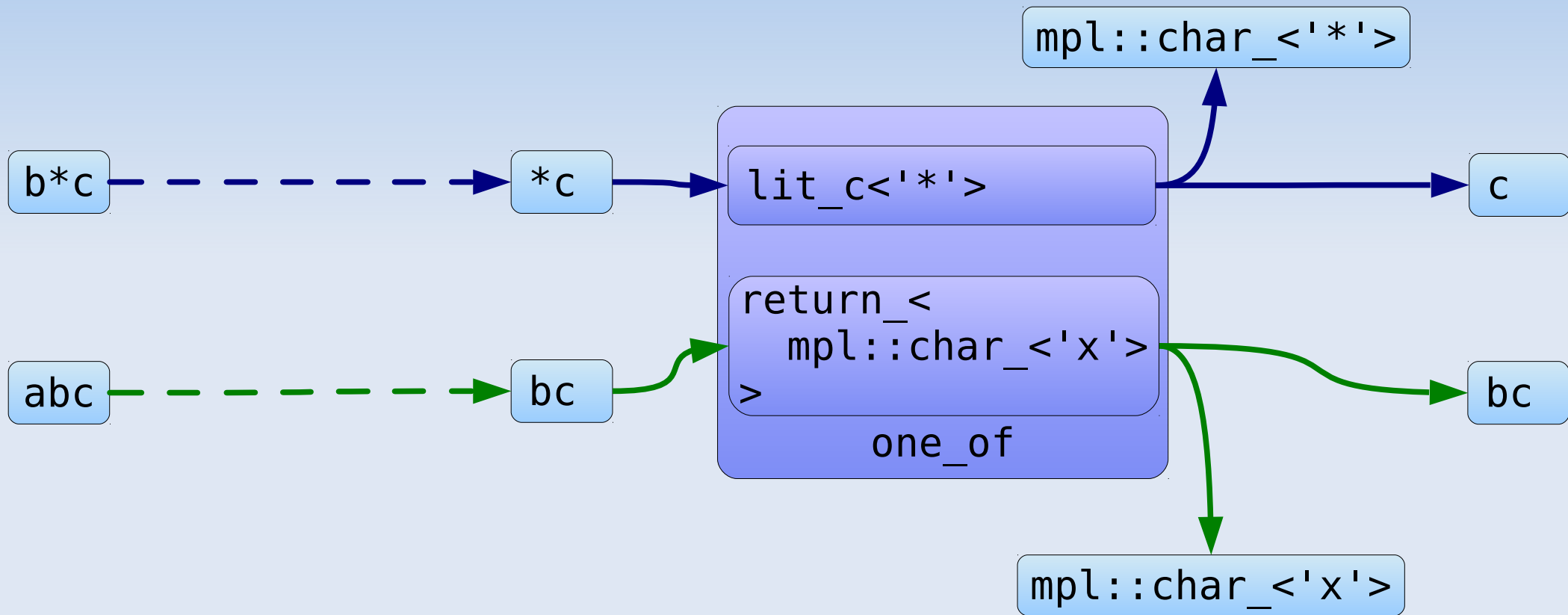
`lit_c<'*>` `return_<mpl::char_<'x'>>`

unary_item ::= item '*'?



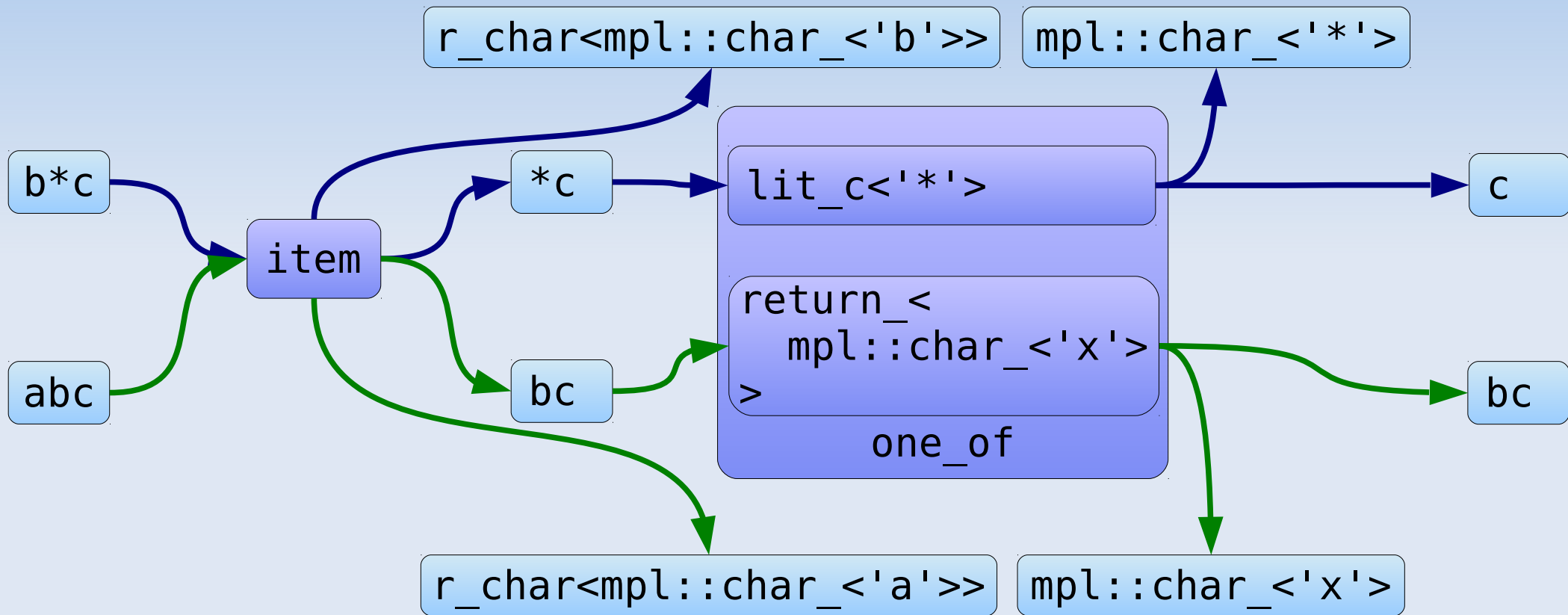
`one_of<lit_c<'*>, return_<mpl::char_<'x'>>>`

unary_item ::= item '*'?



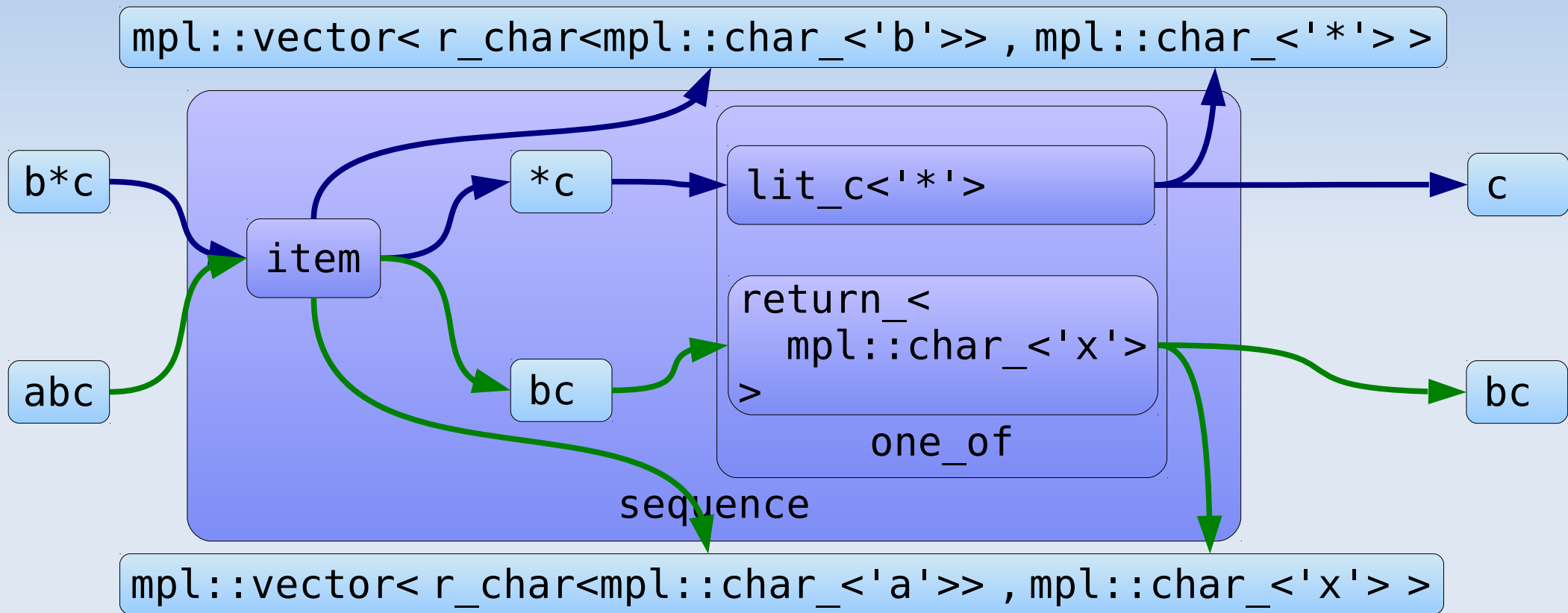
`one_of<lit_c<'*>, return_<mpl::char_<'x'>>>`

unary_item ::= item '*'?



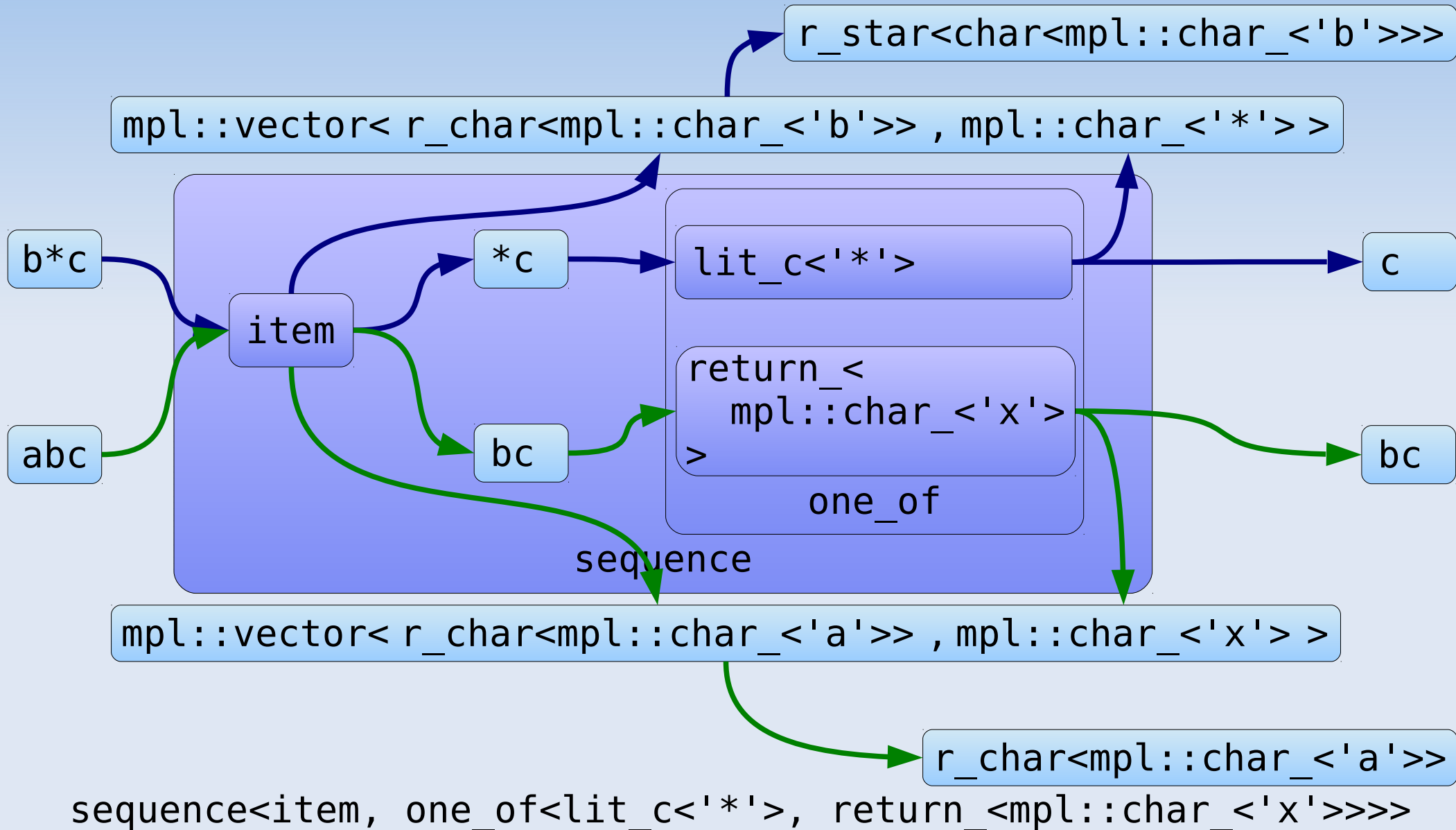
item one_of<lit_c<'*>, return_<mpl::char_<'x'>>>

unary_item ::= item '*'?




`sequence<item, one_of<lit_c<'<*>'>, return_<mpl::char_<'x'>>>>>`

unary_item ::= item '*'?



unary_item ::= item '*'?

`r_star<char<mpl::char_<'b'>>>`



```
template <class RegExp, char Repeat>  
struct impl;
```

`r_char<mpl::char_<'a'>>`



```
sequence<item, one_of<lit_c<'*','>, return_<mpl::char_<'x'>>>>
```

unary_item ::= item '*'?

`r_star<char<mpl::char_<'b'>>>`

```
template <class RegExp, char Repeat>
struct impl;
```

```
template <class RegExp>
struct impl<RegExp, '*'> : r_star<RegExp> {};
```

`r_char<mpl::char_<'a'>>`

`sequence<item, one_of<lit_c<'*','>, return_<mpl::char_<'x'>>>>`

unary_item ::= item '*'?

`r_star<char<mpl::char_<'b'>>>`

```
template <class RegExp, char Repeat>
struct impl;
```

b*c
template <class RegExp>
struct impl<RegExp, '*'> : r_star<RegExp> {};

abc
template <class RegExp>
struct impl<RegExp, 'x'> : RegExp {};

`r_char<mpl::char_<'a'>>`

`sequence<item, one_of<lit_c<'*','x'>, return_<mpl::char_<'x'>>>>`

unary_item ::= item '*'?

`r_star<char<mpl::char_<'b'>>>`

```
template <class RegExp, char Repeat>
struct impl;
```

b*c

```
template <class RegExp>
struct impl<RegExp, '*'> : r_star<RegExp> {};
```

abc

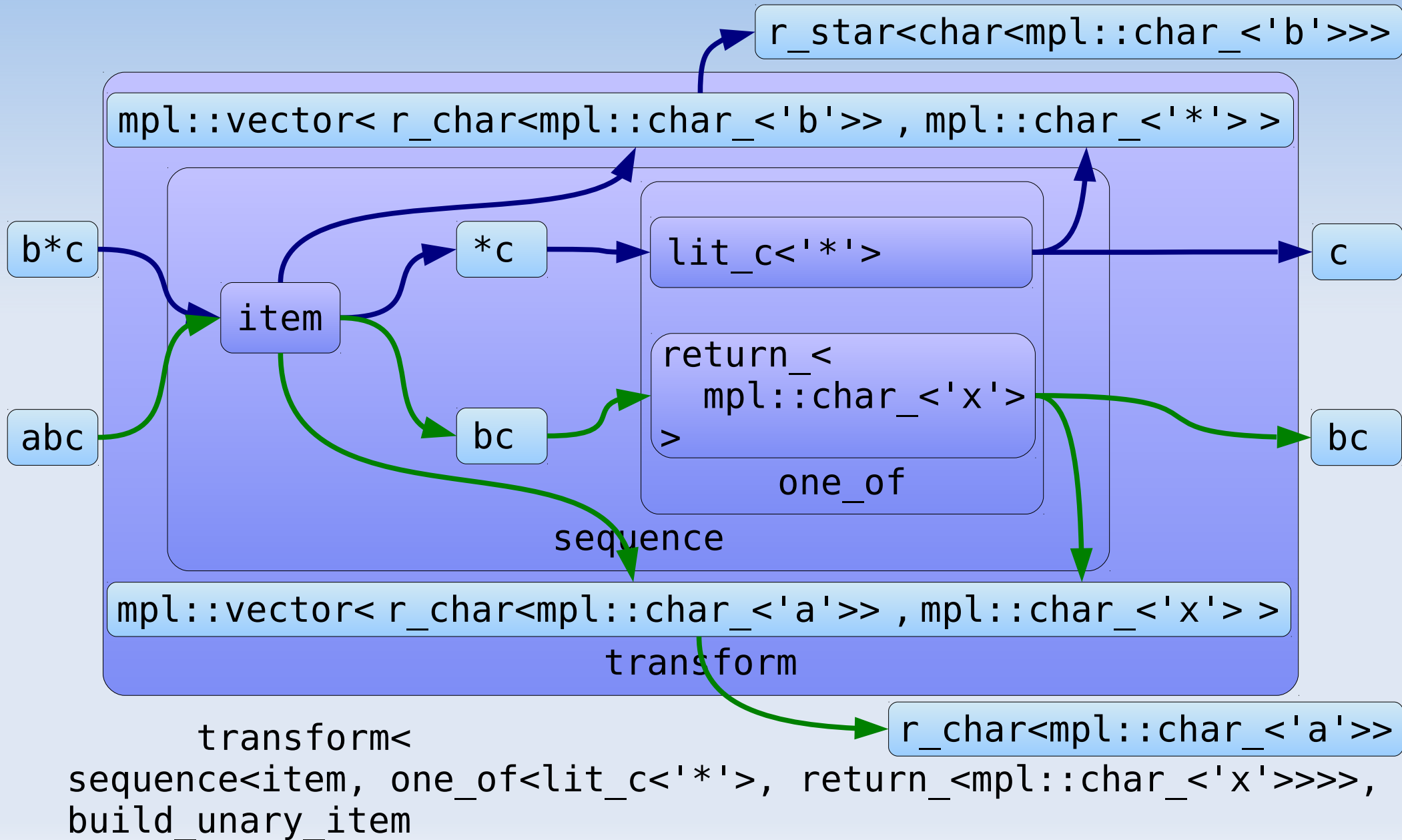
```
template <class RegExp>
struct impl<RegExp, 'x'> : RegExp {};
```

```
struct build_unary_item {
    template <class V>
    struct apply : impl<front<V>, back<V>::type::value> {};
};
```

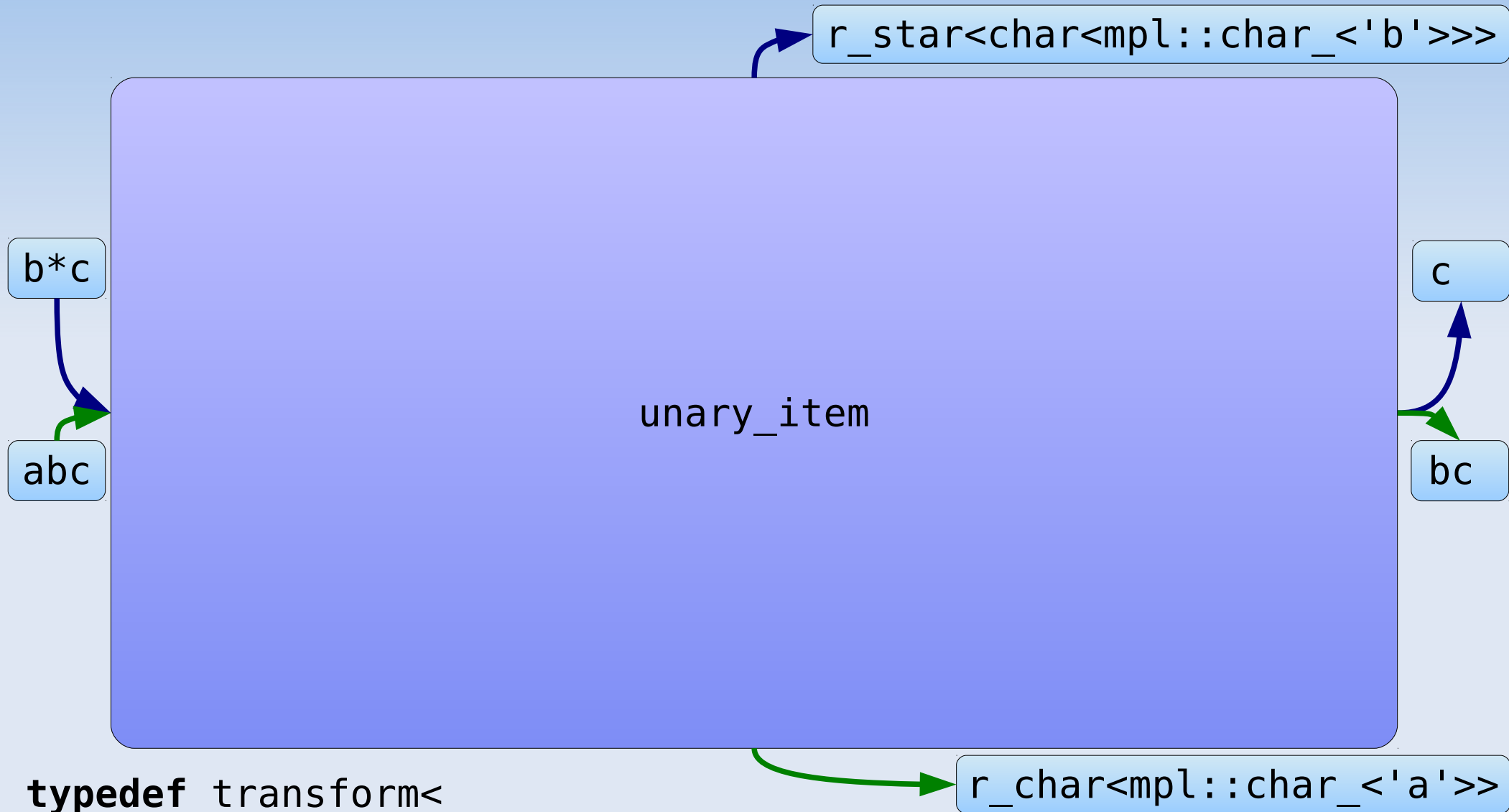
`r_char<mpl::char_<'a'>>`

`sequence<item, one_of<lit_c<'*','x'>, return_<mpl::char_<'x'>>>>`

unary_item ::= item '*'?



unary_item ::= item '*'?



```
typedef transform<  
    sequence<item, one_of<lit_c<'* '>, return_<mpl::char_<'x'>>>>,  
    build_unary_item  
> unary_item;
```

The grammar

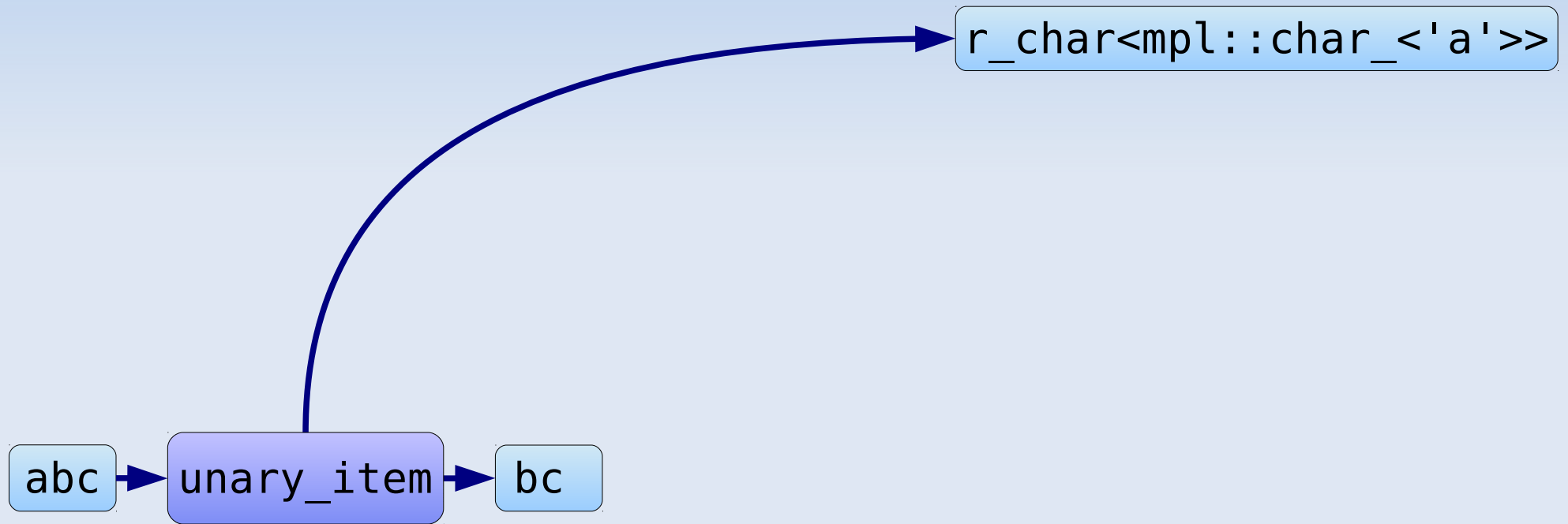
- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

```
reg_exp      ::= unary_item*  
unary_item   ::= item '*'? ✓  
item         ::= any | bracket_exp | char_  
any          ::= '.' ✓  
bracket_exp  ::= '(' reg_exp ')' ✓  
char_        ::= number | letter ✓  
number       ::= '0'..'9' ✓  
letter       ::= 'a'..'z' | 'A'..'Z' ✓
```

reg_exp ::= unary_item*

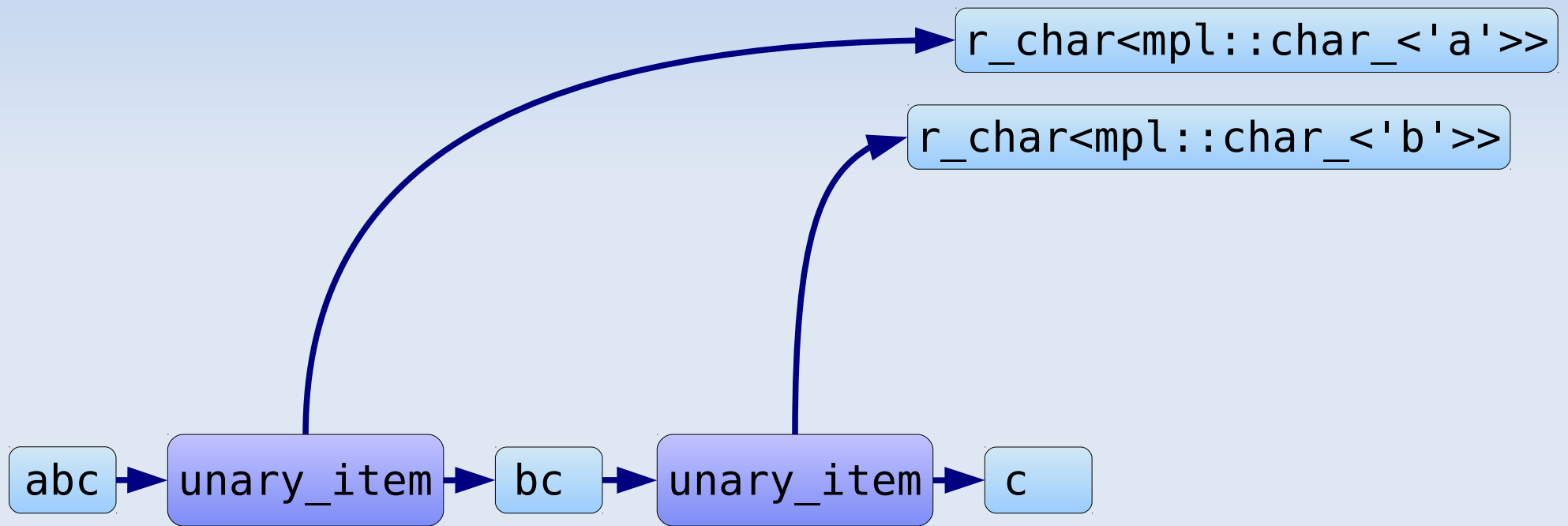
abc

`reg_exp ::= unary_item*`



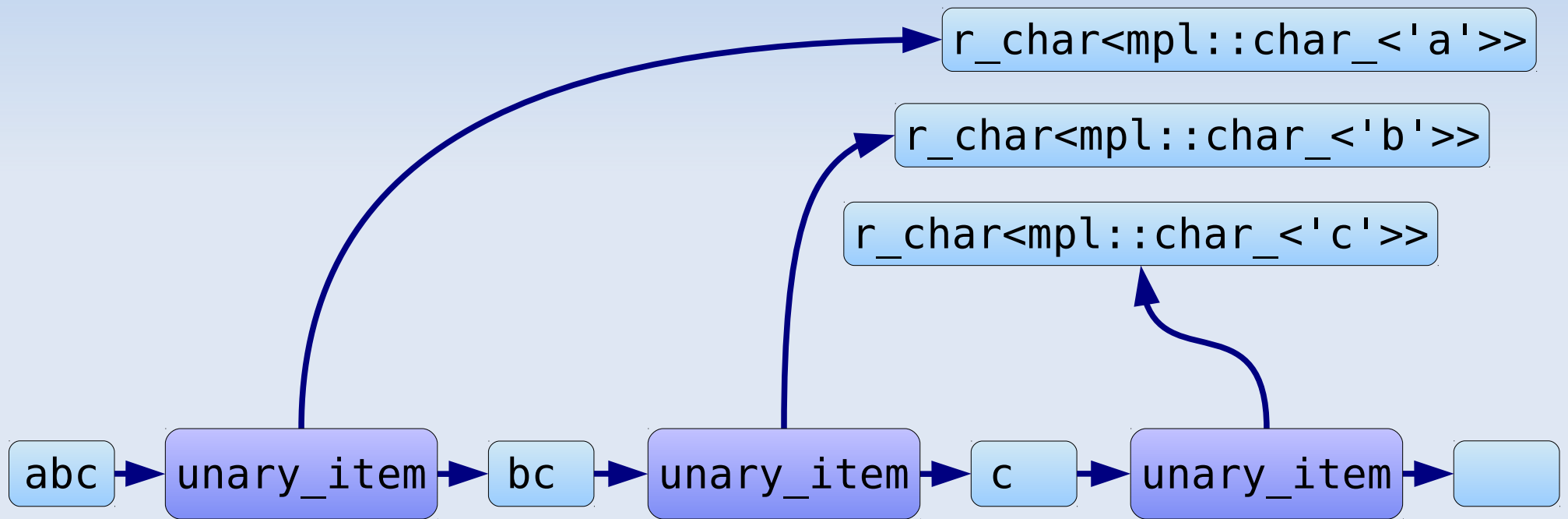
`unary_item`

reg_exp ::= unary_item*



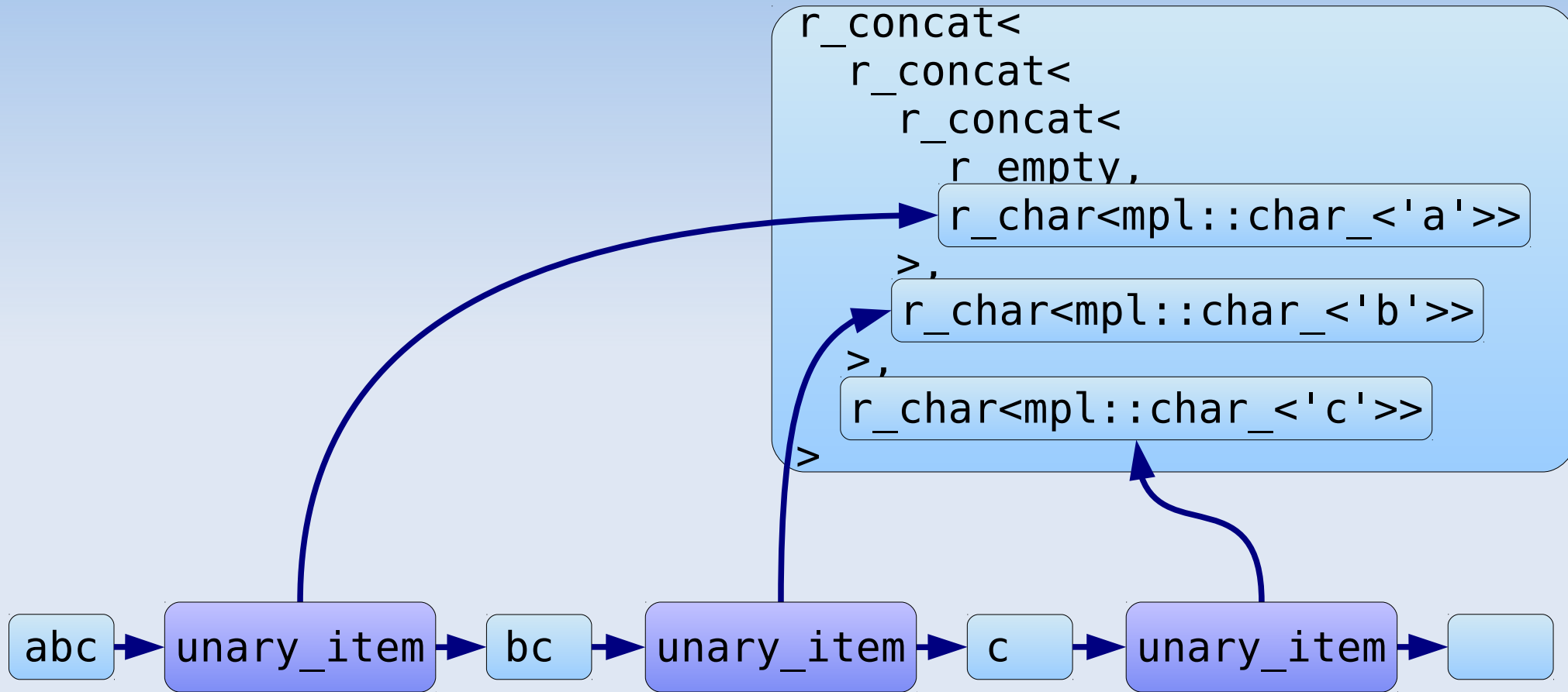
unary_item

reg_exp ::= unary_item*



unary_item

reg_exp ::= unary_item*



unary_item

`reg_exp ::= unary_item*`

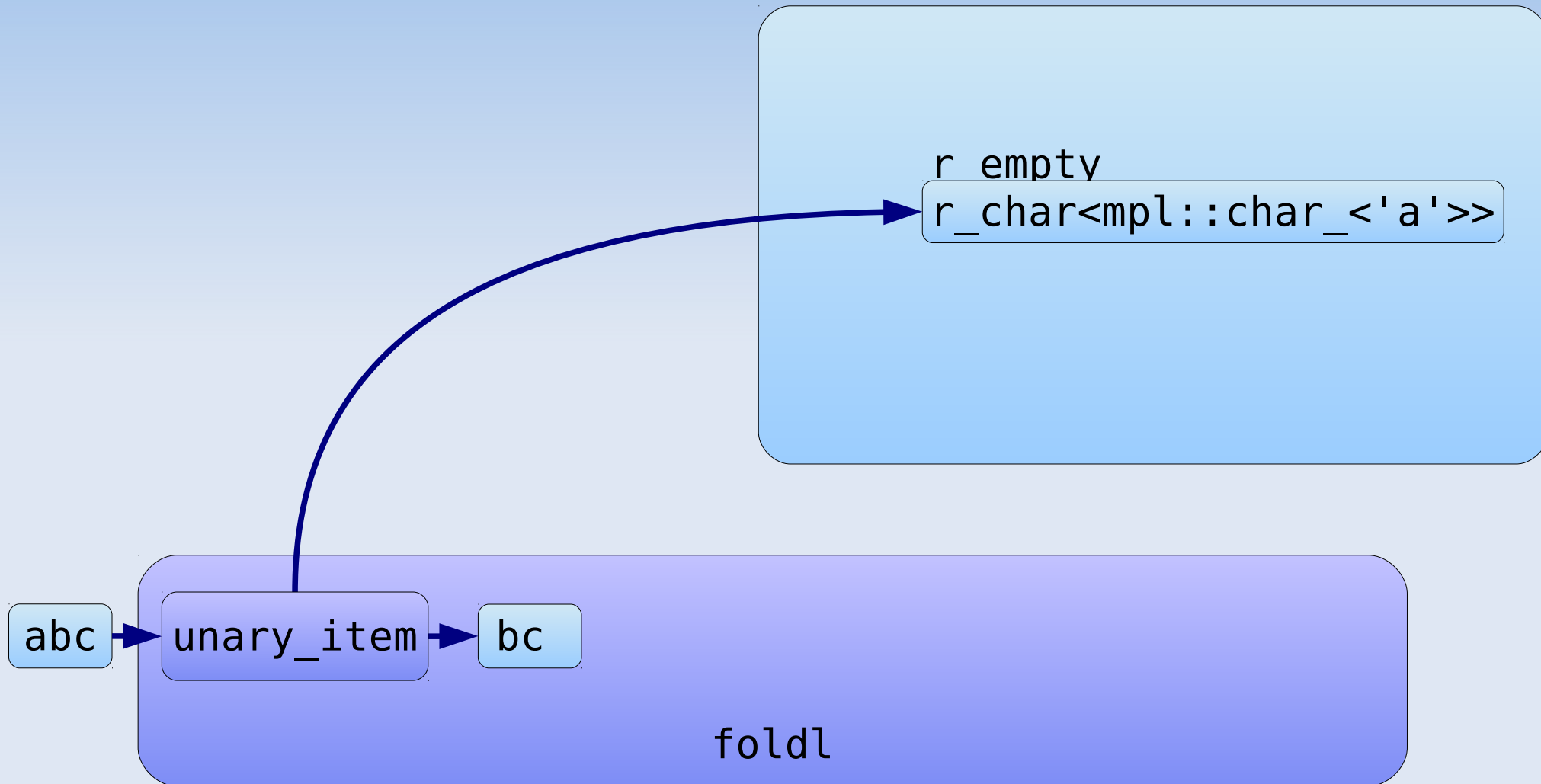
`r_empty`

`abc` →

`foldl`

`foldl<unary_item, r_empty, build_reg_exp>`

reg_exp ::= unary_item*



`foldl<unary_item, r_empty, build_reg_exp>`

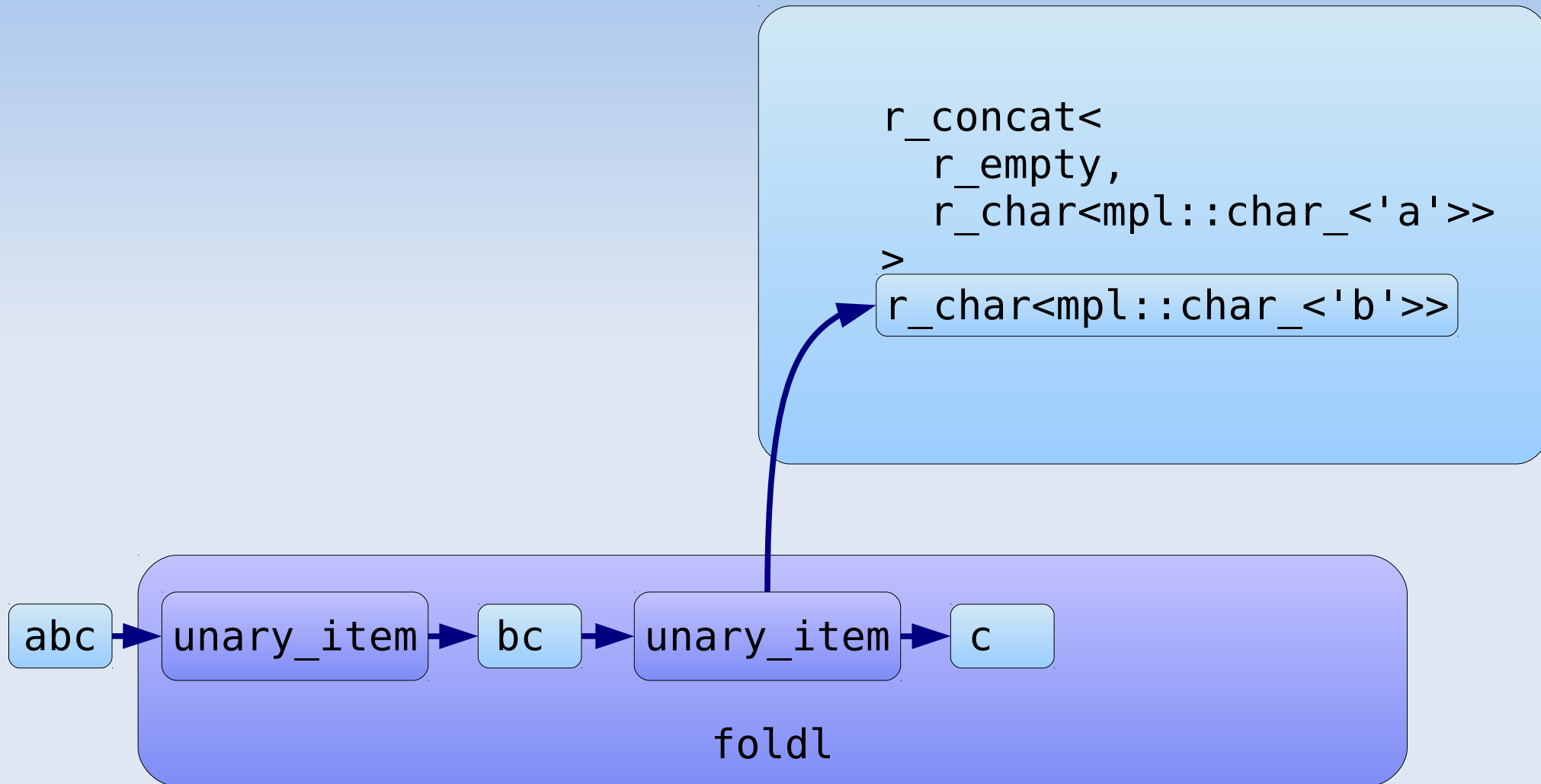
reg_exp ::= unary_item*

```
r_concat<  
  r_empty,  
  r_char<mpl::char_<'a'>>  
>
```



```
foldl<unary_item, r_empty, build_reg_exp>
```

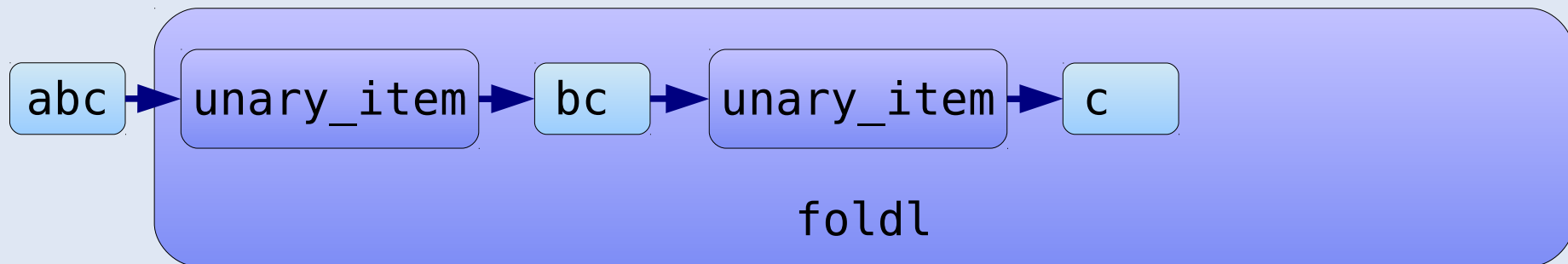
reg_exp ::= unary_item*



`foldl<unary_item, r_empty, build_reg_exp>`

reg_exp ::= unary_item*

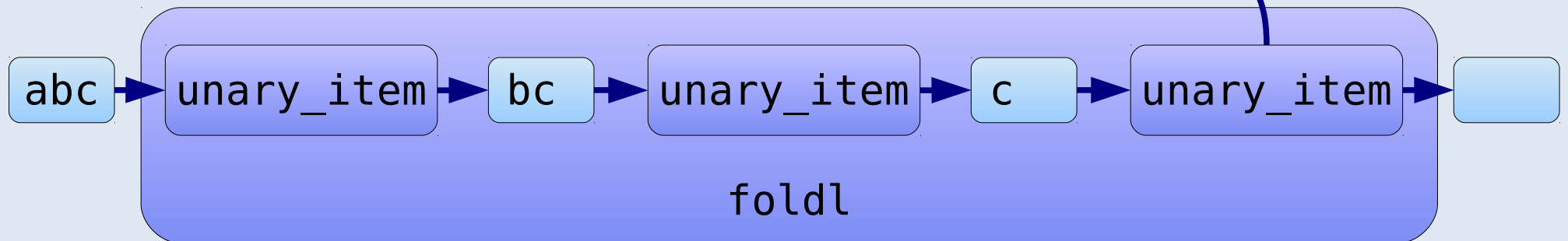
```
r_concat<
  r_concat<
    r_empty,
    r_char<mpl::char_<'a'>>
  >,
  r_char<mpl::char_<'b'>>
>
```



```
foldl<unary_item, r_empty, build_reg_exp>
```

reg_exp ::= unary_item*

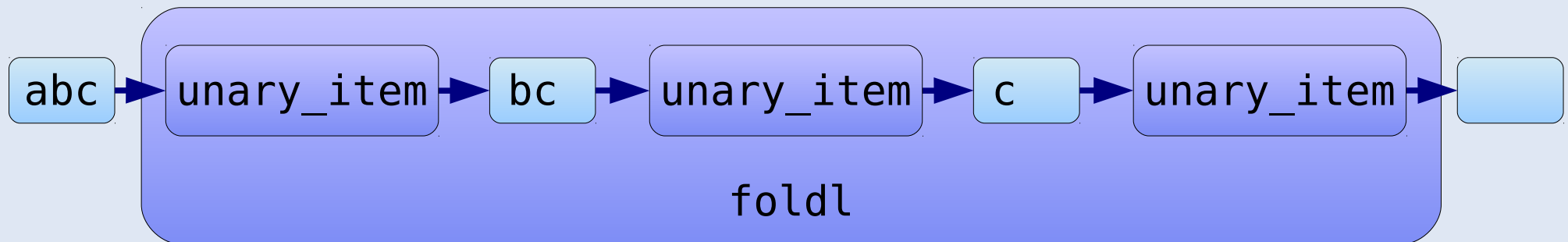
```
r_concat<
  r_concat<
    r_empty,
    r_char<mpl::char_<'a'>>
  >,
  r_char<mpl::char_<'b'>>
>
r_char<mpl::char_<'c'>>
```



`foldl<unary_item, r_empty, build_reg_exp>`

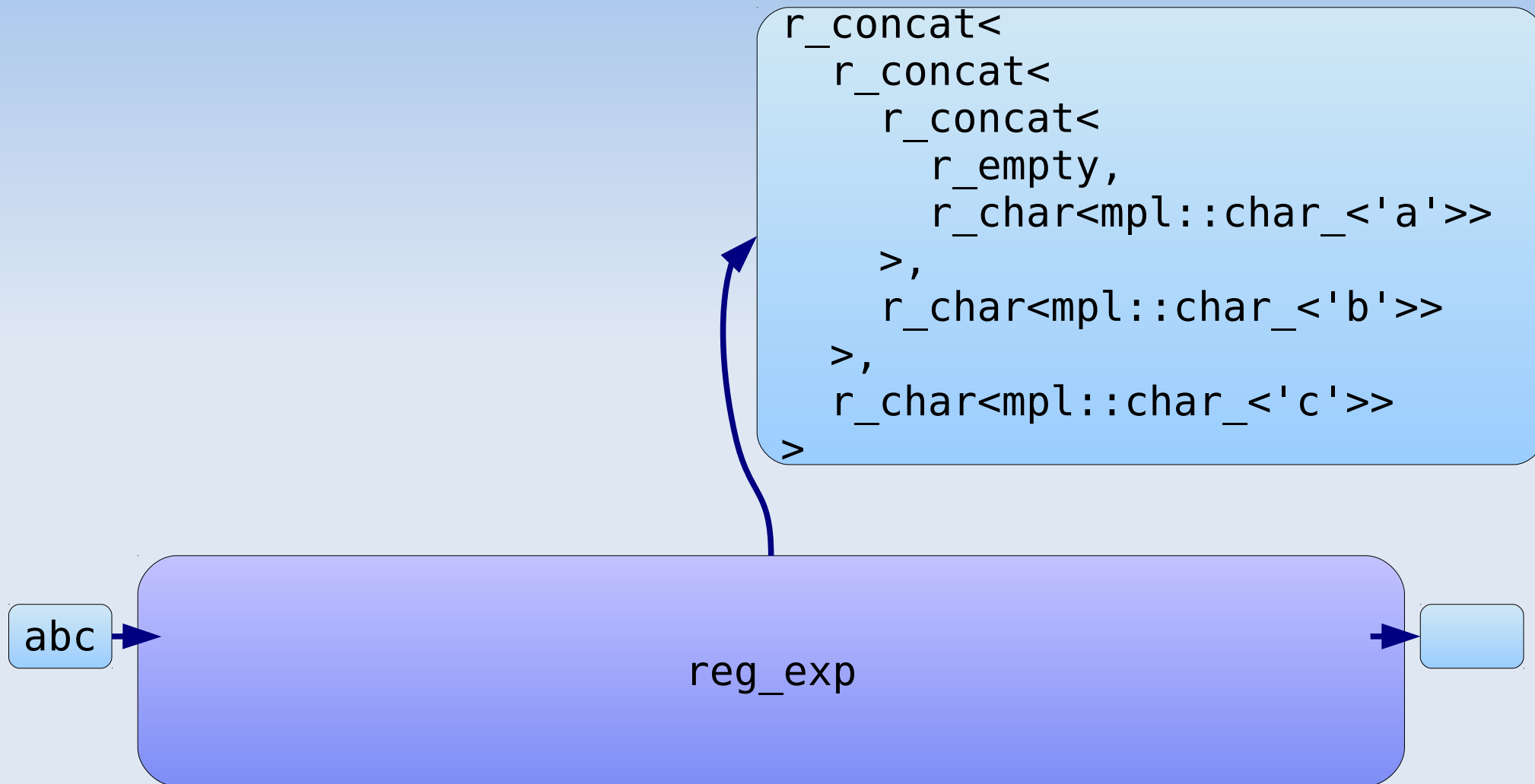
reg_exp ::= unary_item*

```
r_concat<
  r_concat<
    r_concat<
      r_empty,
      r_char<mpl::char_<'a'>>
    >,
    r_char<mpl::char_<'b'>>
  >,
  r_char<mpl::char_<'c'>>
>
```



`foldl<unary_item, r_empty, build_reg_exp>`

reg_exp ::= unary_item*



```
typedef foldl<unary_item, r_empty, build_reg_exp> reg_exp;
```

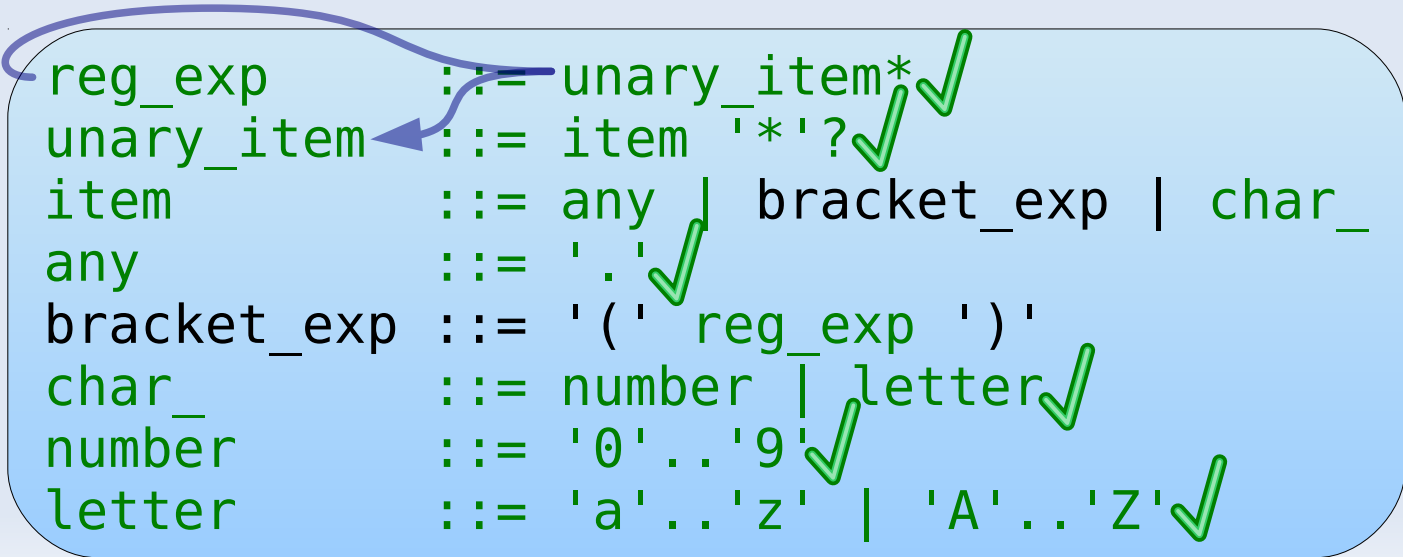
The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

```
reg_exp      ::= unary_item* ✓
unary_item   ::= item '*'? ✓
item         ::= any | bracket_exp | char_
any          ::= '.' ✓
bracket_exp  ::= '(' reg_exp ')'
char_        ::= number | letter ✓
number       ::= '0'..'9' ✓
letter       ::= 'a'..'z' | 'A'..'Z' ✓
```

The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)



A light blue rounded rectangle contains the following grammar rules. Green checkmarks and arrows are drawn over the rules. A purple arrow points from the `reg_exp` rule to the `unary_item` rule. Another purple arrow points from the `unary_item` rule to the `item` rule. Green checkmarks are placed at the end of each rule line.

```
reg_exp ::= unary_item* ✓  
unary_item ::= item '*'? ✓  
item ::= any | bracket_exp | char_ ✓  
any ::= '.' ✓  
bracket_exp ::= '(' reg_exp ')' ✓  
char_ ::= number | letter ✓  
number ::= '0'..'9' ✓  
letter ::= 'a'..'z' | 'A'..'Z' ✓
```

The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

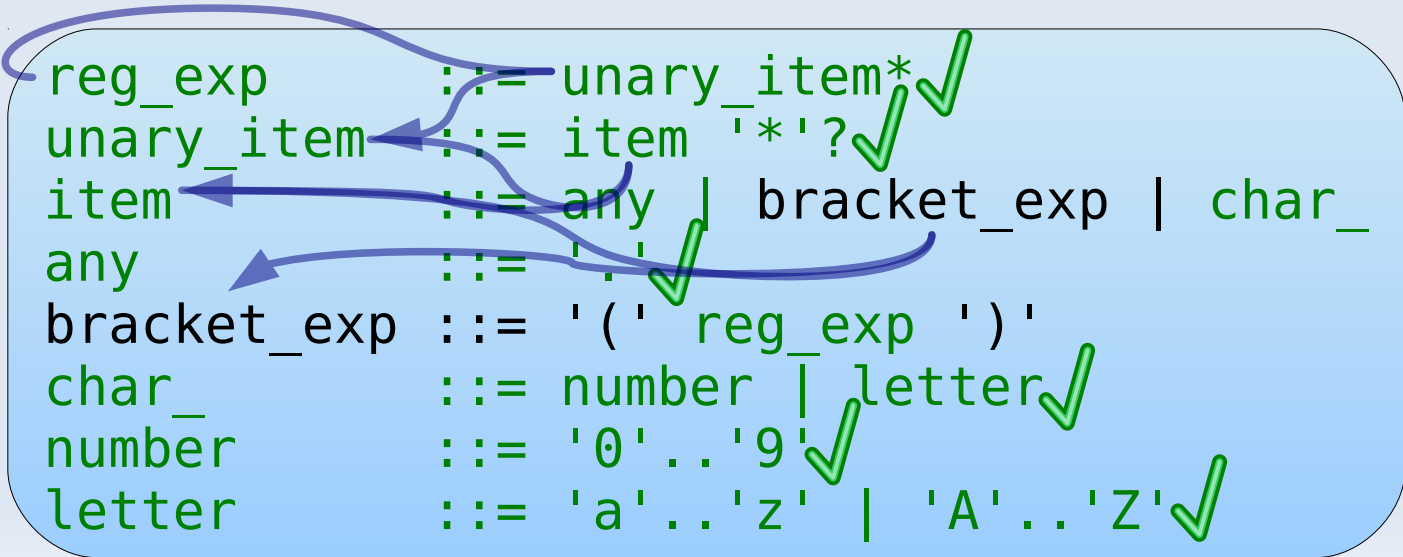
A diagram showing a grammar with the following rules. The rules are enclosed in a light blue rounded rectangle. Handwritten green checkmarks and arrows are used to highlight and connect parts of the grammar.

```
reg_exp ::= unary_item* ✓  
unary_item ::= item '*'? ✓  
item ::= any | bracket_exp | char_ ✓  
any ::= '.' ✓  
bracket_exp ::= '(' reg_exp ')' ✓  
char_ ::= number | letter ✓  
number ::= '0'..'9' ✓  
letter ::= 'a'..'z' | 'A'..'Z' ✓
```

Handwritten annotations include green checkmarks at the end of each rule and green arrows pointing from the right-hand side of a rule to the left-hand side of another rule, indicating dependencies or derivations. For example, an arrow points from 'unary_item' in the first rule to 'unary_item' in the second rule, and another from 'any' in the third rule to 'any' in the fourth rule.

The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)



The diagram shows a grammar with the following rules, each followed by a green checkmark:

- `reg_exp ::= unary_item*`
- `unary_item ::= item '*'?`
- `item ::= any | bracket_exp | char_`
- `any ::= '.'`
- `bracket_exp ::= '(' reg_exp ')'`
- `char_ ::= number | letter`
- `number ::= '0'..'9'`
- `letter ::= 'a'..'z' | 'A'..'Z'`

Blue arrows indicate recursive dependencies: from `reg_exp` to `unary_item`, from `unary_item` to `item`, from `item` to `any`, and from `any` to `bracket_exp`. The `any` rule is crossed out with a blue line.

The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

```
reg_exp ::= unary_item* ✓
unary_item ::= item '*'? ✓
item ::= any | bracket_exp | char_ ✓
any ::= '.' ✓
bracket_exp ::= '(' reg_exp ')' ✓
char_ ::= number | letter ✓
number ::= '0'..'9' ✓
letter ::= 'a'..'z' | 'A'..'Z' ✓
```

bracket_exp ::= ' (' reg_exp ') '

// ...

typedef

bracket_exp;

// ...

typedef foldl<unary_item, r_empty, build_reg_exp> reg_exp;

bracket_exp ::= ' (' reg_exp ') '

// ...

typedef

bracket_exp;

// ...

typedef foldl<unary_item, r_empty, build_reg_exp> reg_exp;



struct reg_exp : foldl<unary_item, r_empty, build_reg_exp> {};

bracket_exp ::= ' (' reg_exp ') '

```
struct reg_exp;
```

```
// ...
```

```
typedef
```

```
    bracket_exp;
```

```
// ...
```

```
typedef foldl<unary_item, r_empty, build_reg_exp> reg_exp;
```



```
struct reg_exp : foldl<unary_item, r_empty, build_reg_exp> {};
```

bracket_exp ::= ' (' reg_exp ') '

```
struct reg_exp;
```

```
// ...
```

```
typedef
```

```
reg_exp
```

```
bracket_exp;
```

```
// ...
```

```
typedef foldl<unary_item, r_empty, build_reg_exp> reg_exp;
```



```
struct reg_exp : foldl<unary_item, r_empty, build_reg_exp> {};
```

bracket_exp ::= '(' reg_exp ')'

```
struct reg_exp;
```

```
// ...
```

```
typedef
```

```
    lit_c<'('> reg_exp lit_c<')'>  
bracket_exp;
```

```
// ...
```

```
typedef foldl<unary_item, r_empty, build_reg_exp> reg_exp;
```



```
struct reg_exp : foldl<unary_item, r_empty, build_reg_exp> {};
```

bracket_exp ::= '(' reg_exp ')'

```
struct reg_exp;
```

```
// ...
```

```
typedef
```

```
    middle_of<lit_c<'('>, reg_exp, lit_c<')'>>  
    bracket_exp;
```

```
// ...
```

```
typedef foldl<unary_item, r_empty, build_reg_exp> reg_exp;
```



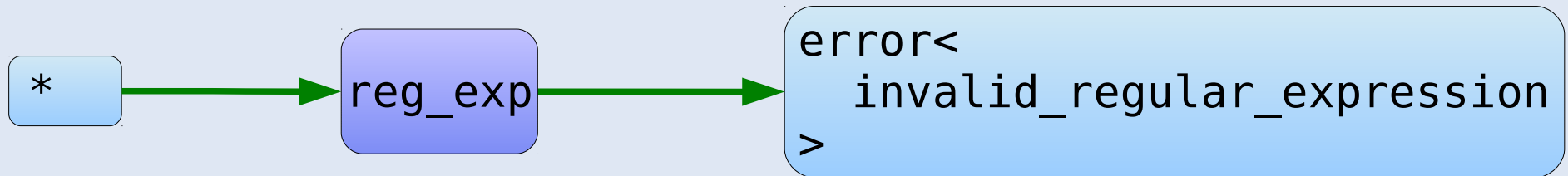
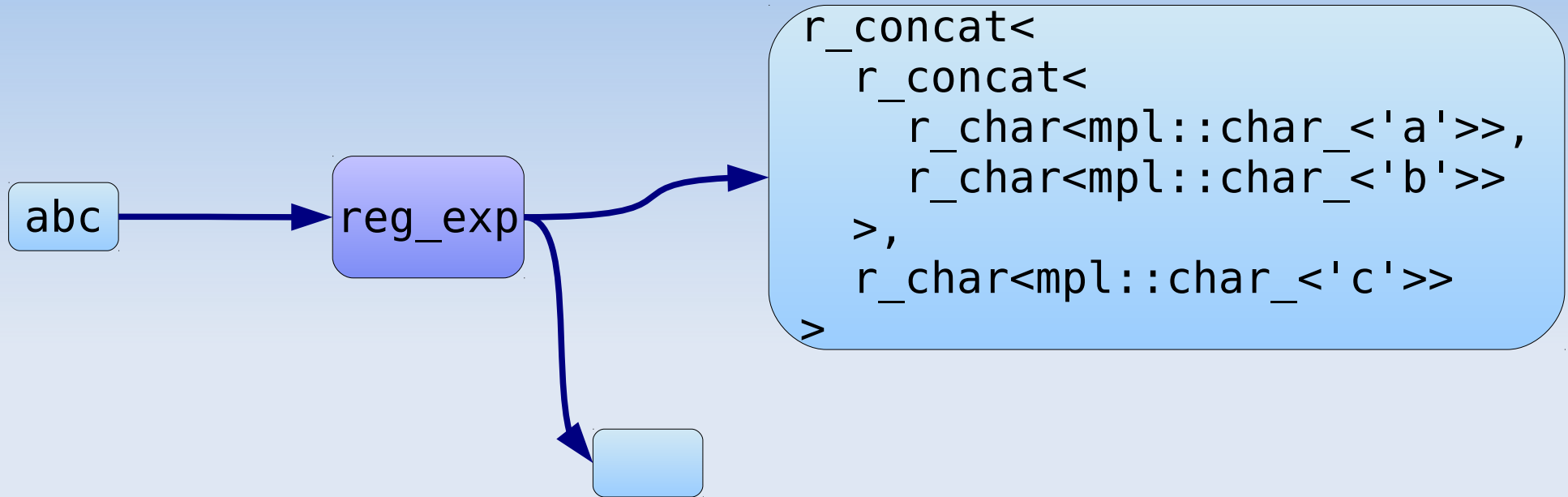
```
struct reg_exp : foldl<unary_item, r_empty, build_reg_exp> {};
```

The grammar

- We will support
 - letters and numbers (eg. **abc123**)
 - .
 - *
 - brackets (eg. **(abc)***)

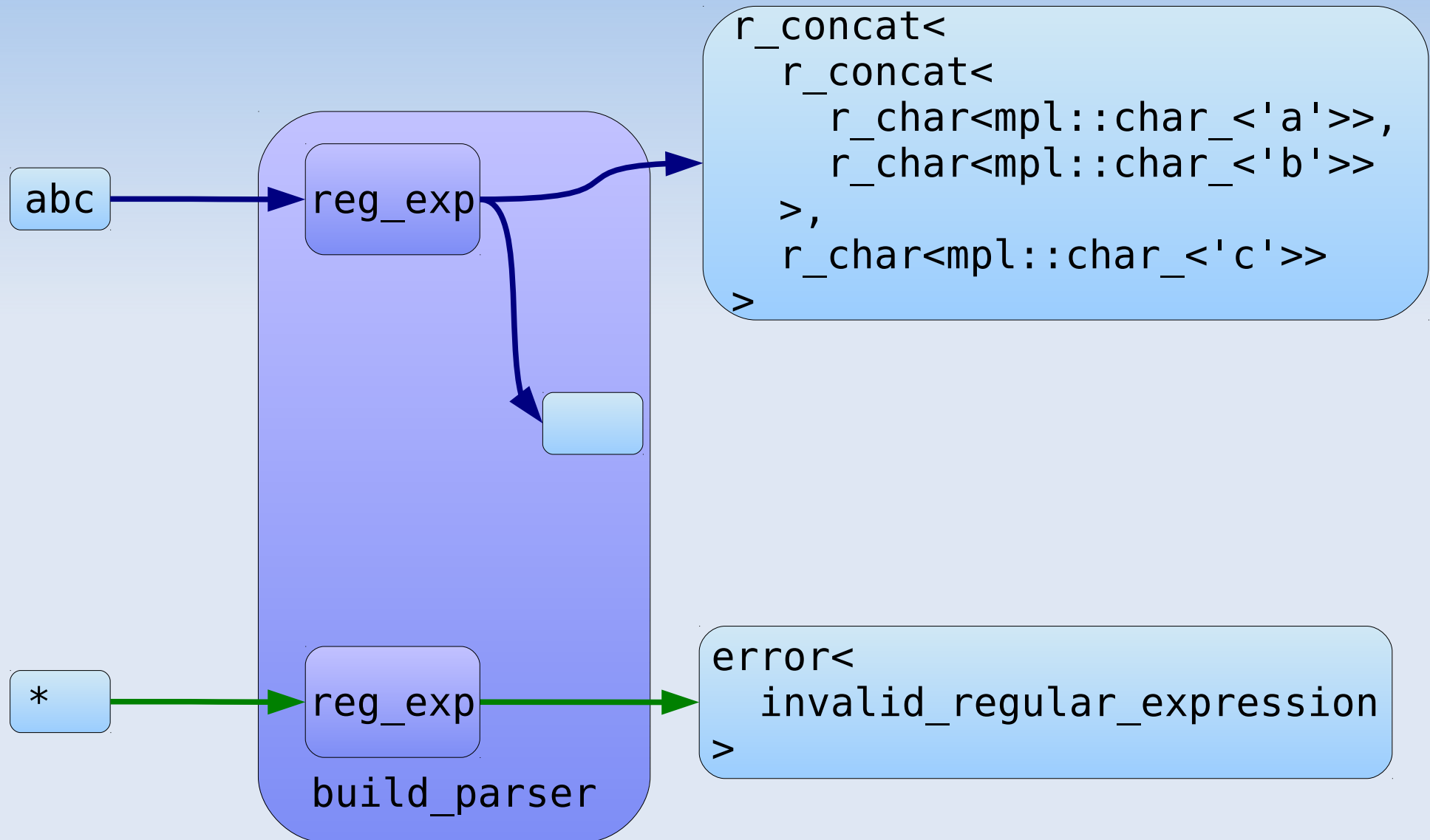
```
reg_exp      ::= unary_item* ✓  
unary_item   ::= item '*'? ✓  
item         ::= any | bracket_exp | char_ ✓  
any          ::= '.' ✓  
bracket_exp  ::= '(' reg_exp ')' ✓  
char_        ::= number | letter ✓  
number       ::= '0'..'9' ✓  
letter       ::= 'a'..'z' | 'A'..'Z' ✓
```

Parsing regular expressions



reg_exp

Parsing regular expressions



```
typedef build_parser<reg_exp> reg_exp_parser;
```


Using the parser

```
sregex re =  
    regex_parser::apply<MPLLIBS_STRING("abc")>::type::run();
```

Using the parser

```
sregex re =  
    regex_parser::apply<MPLLIBS_STRING("abc")>::type::run()
```



```
#define REGEX(s) \  
    (regex_parser::apply<MPLLIBS_STRING(s)>::type::run())
```

Using the parser

```
sregex re = REGEX("abc");
```

```
#define REGEX(s) \  
    (regex_parser::apply<MPLLIBS_STRING(s)>::type::run())
```

Lab 6

- Build the regular expression parser

Summary

- DSL embedding into C++
 - Early validation and error reporting
 - Efficient implementation
 - Keeping the common syntax
- Improving the interface of existing libraries

Q & A

Mpllibs.Metaparse

<http://abel.web.elte.hu/mpllibs>

https://github.com/sabel83/metaparse_tutorial