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`

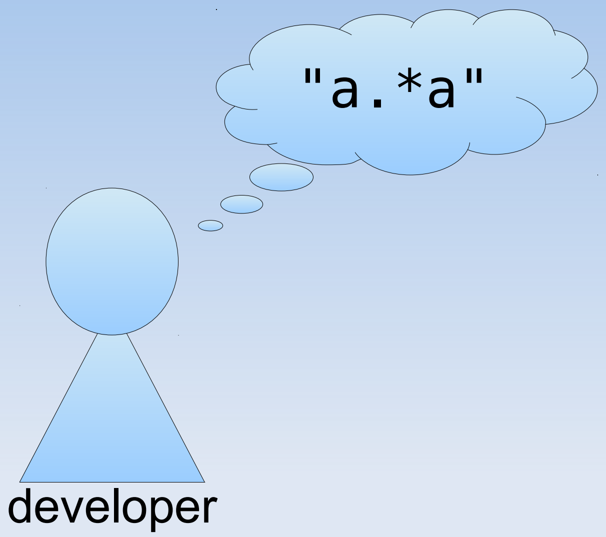
Mpllibs

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

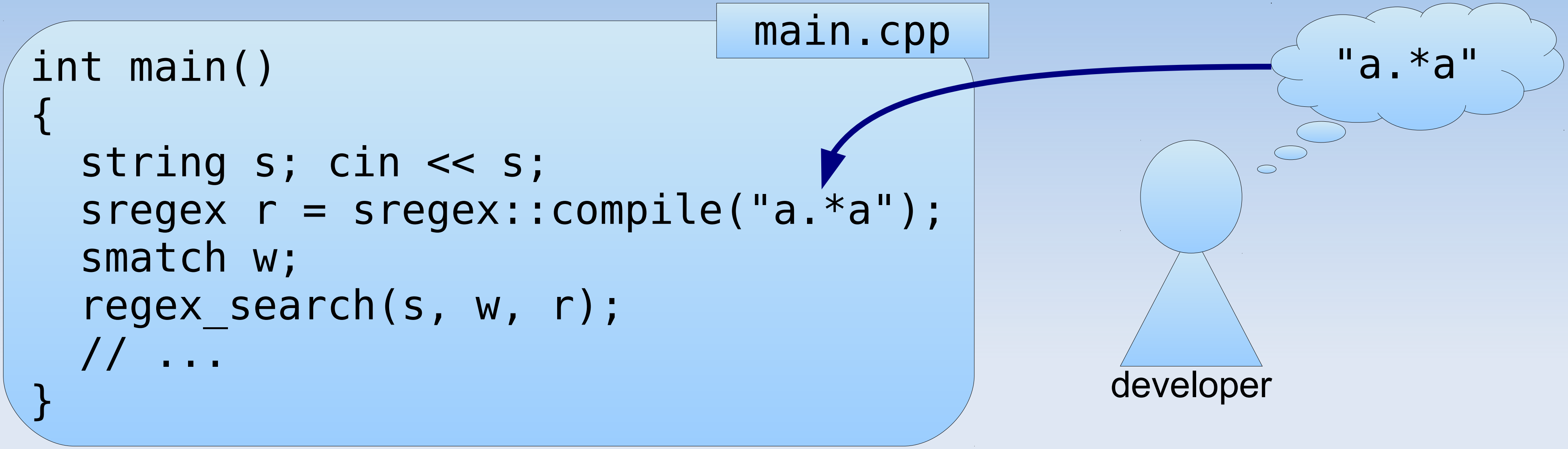
Mpllibs

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

Processing DSLs



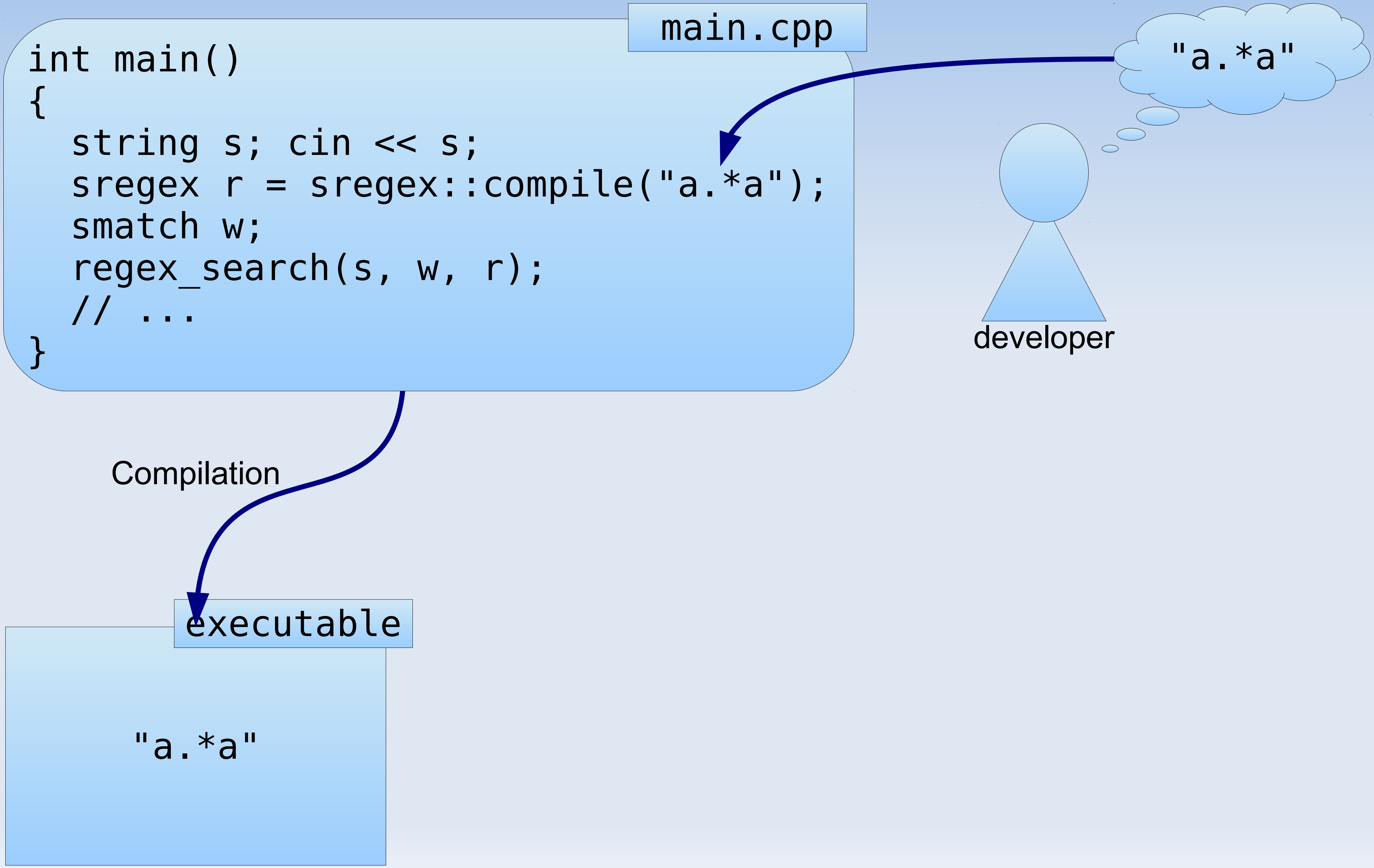
Processing DSLs



main.cpp

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

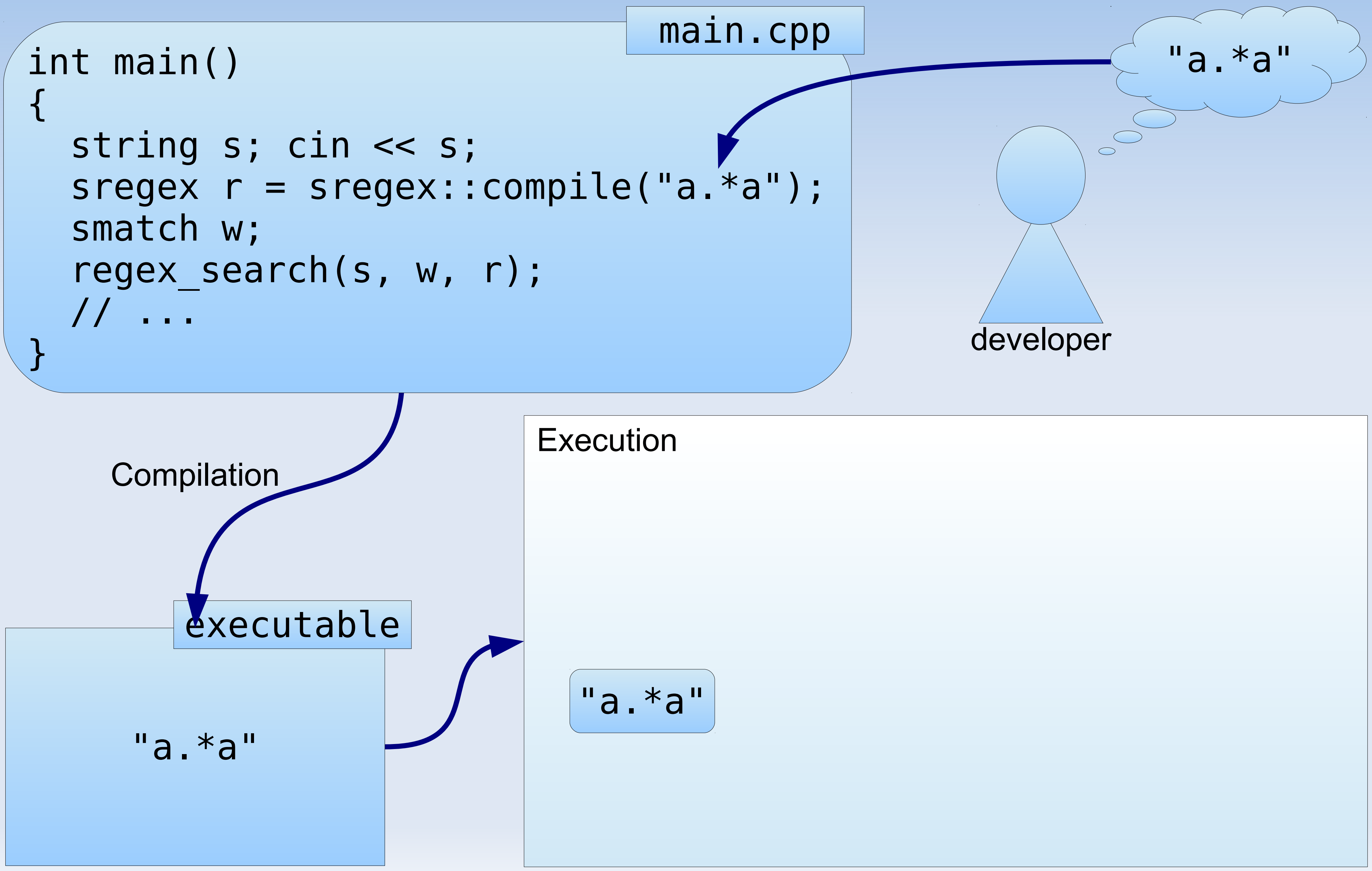
Processing DSLs



main.cpp

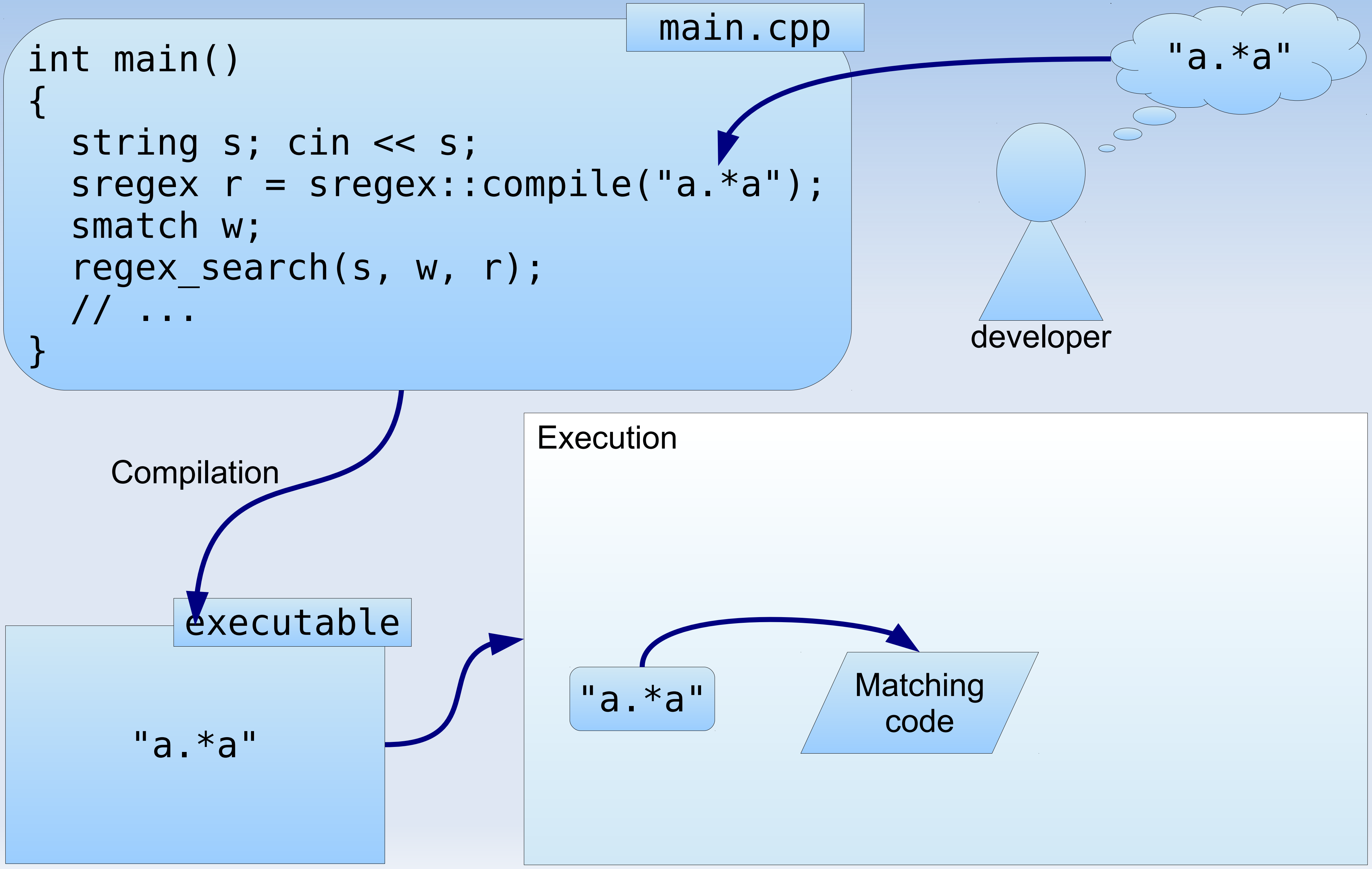
```
int main()
{
  string s; cin << s;
  sregex r = sregex::compile("a.*a");
  smatch w;
  regex_search(s, w, r);
  // ...
}
```

Processing DSLs



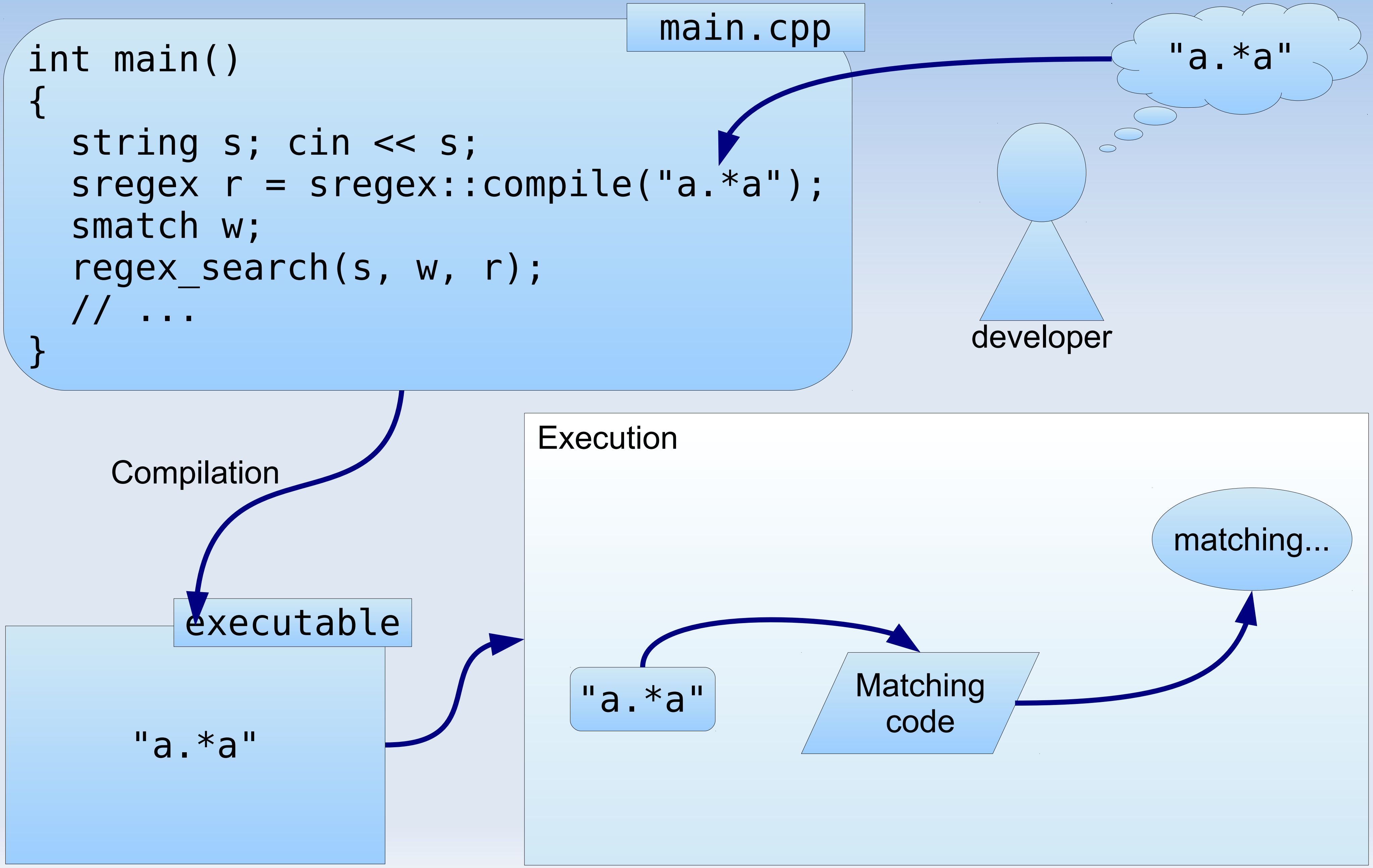
```
int main()
{
  string s; cin << s;
  sregex r = sregex::compile("a.*a");
  smatch w;
  regex_search(s, w, r);
  // ...
}
```

Processing DSLs



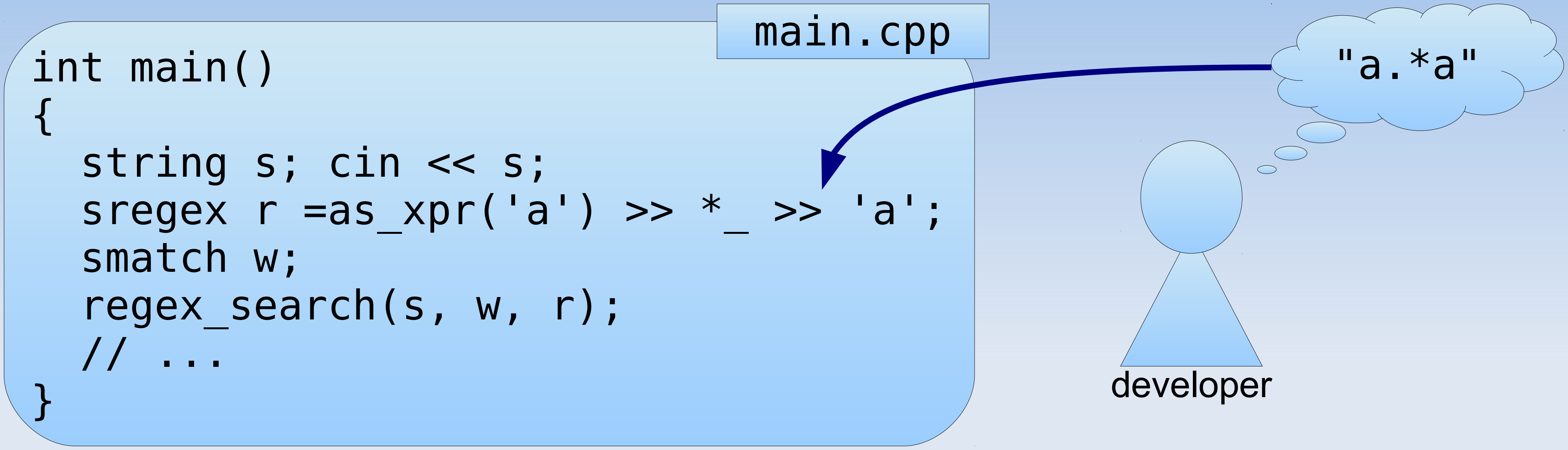
```
int main()
{
  string s; cin << s;
  sregex r = sregex::compile("a.*a");
  smatch w;
  regex_search(s, w, r);
  // ...
}
```

Processing DSLs



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

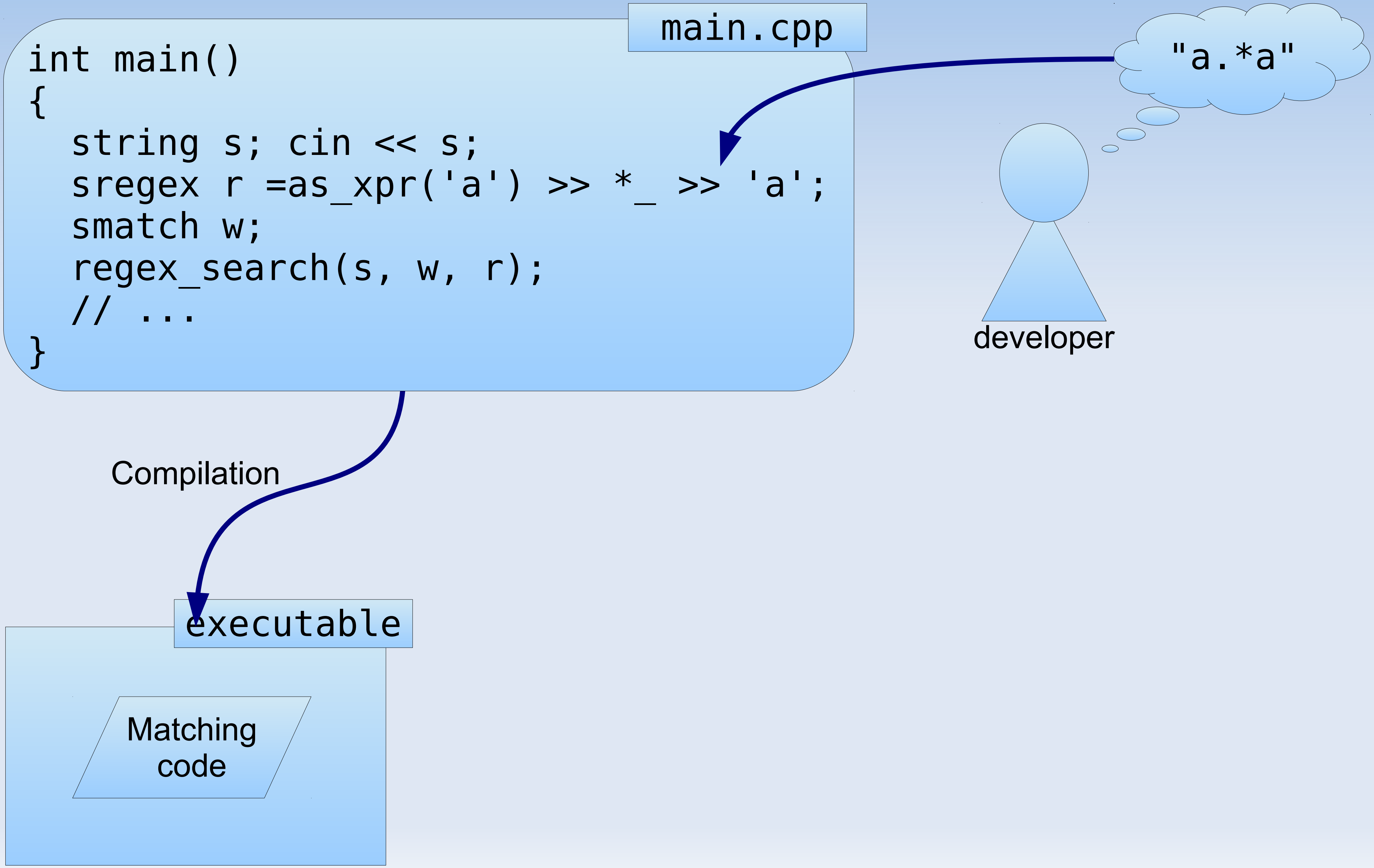
Processing DSLs



main.cpp

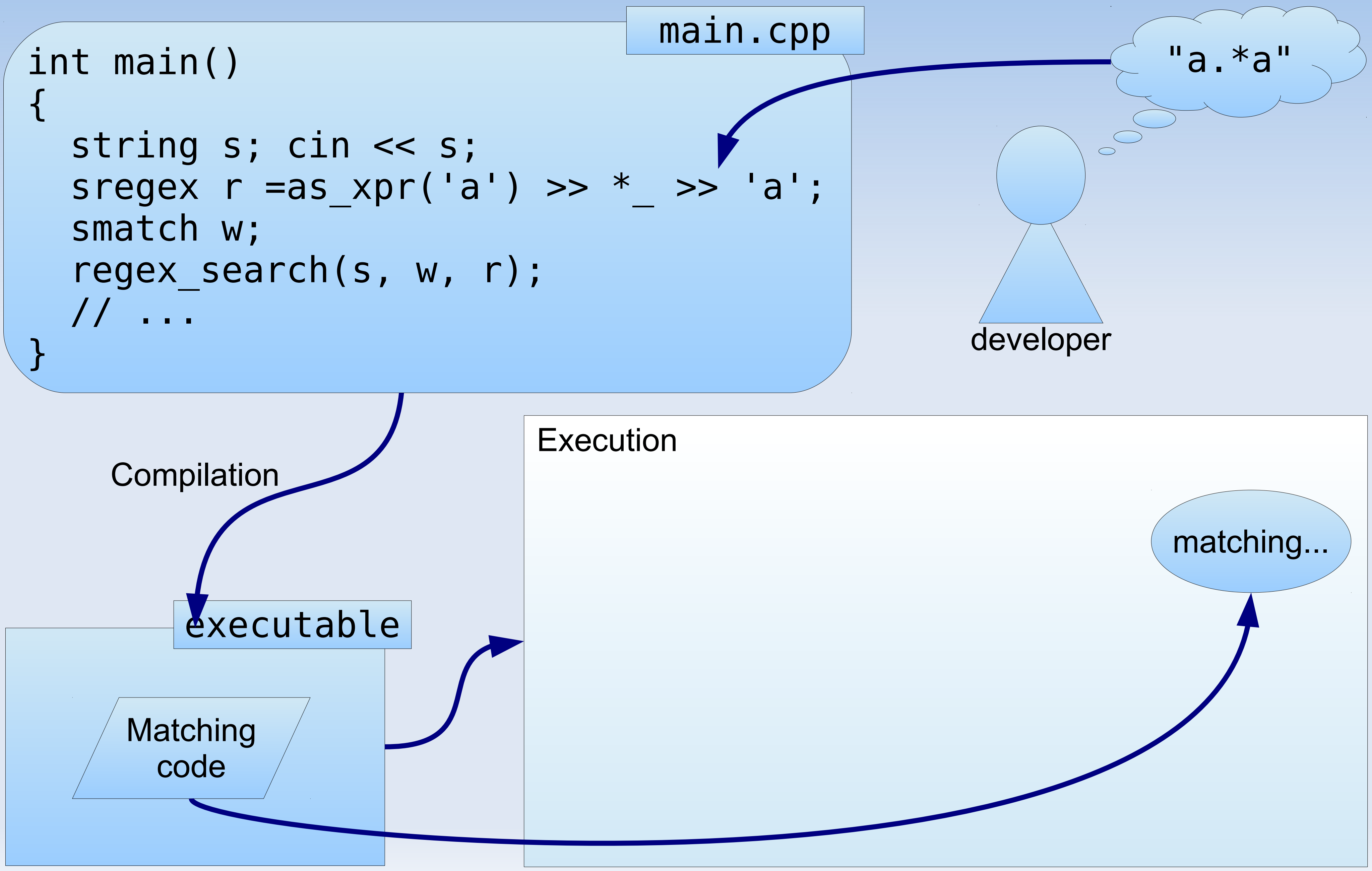
```
int main()
{
  string s; cin << s;
  sregex r =as_xpr('a') >> *_ >> 'a';
  smatch w;
  regex_search(s, w, r);
  // ...
}
```

Processing DSLs



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

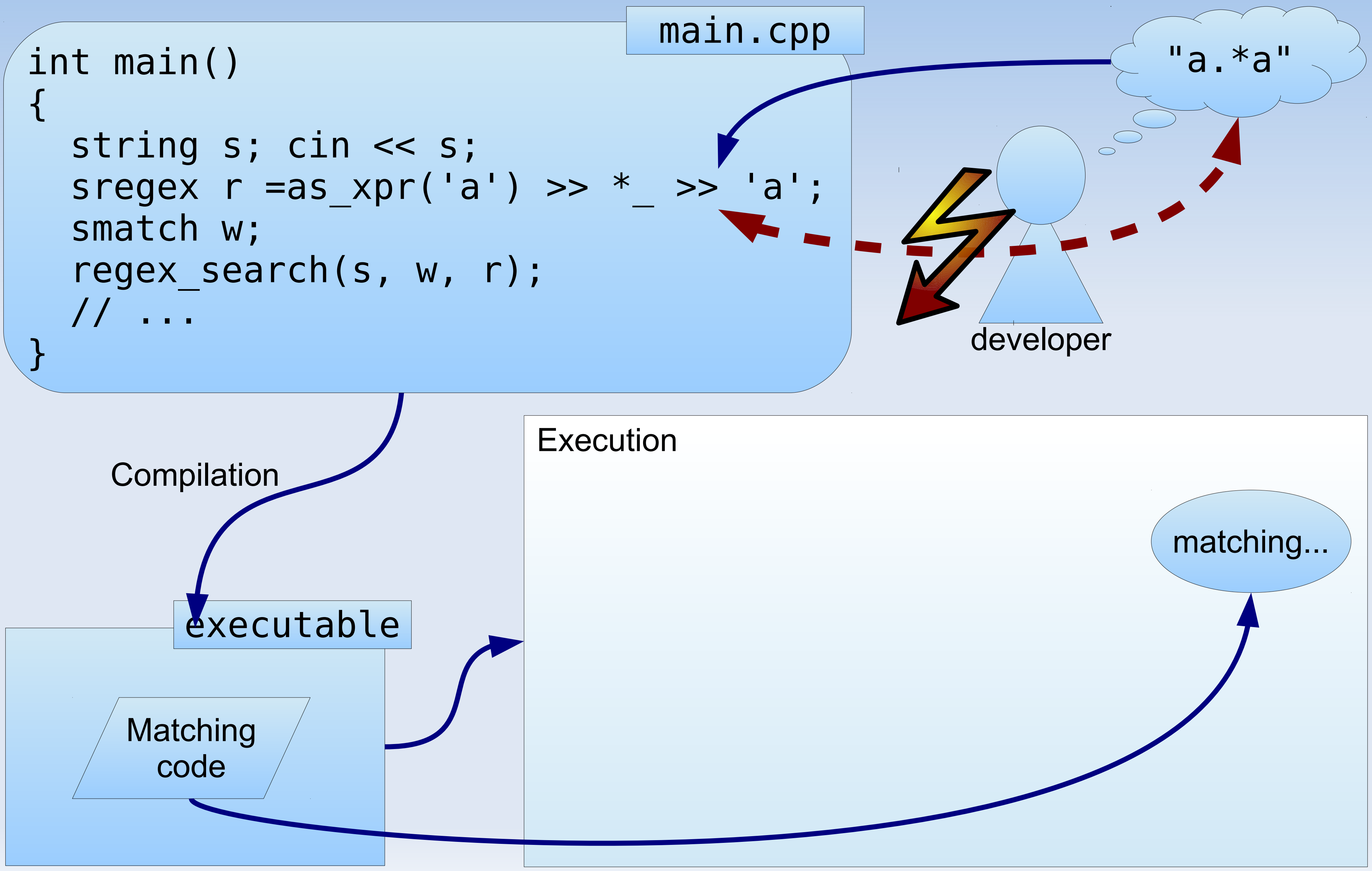
Processing DSLs



main.cpp

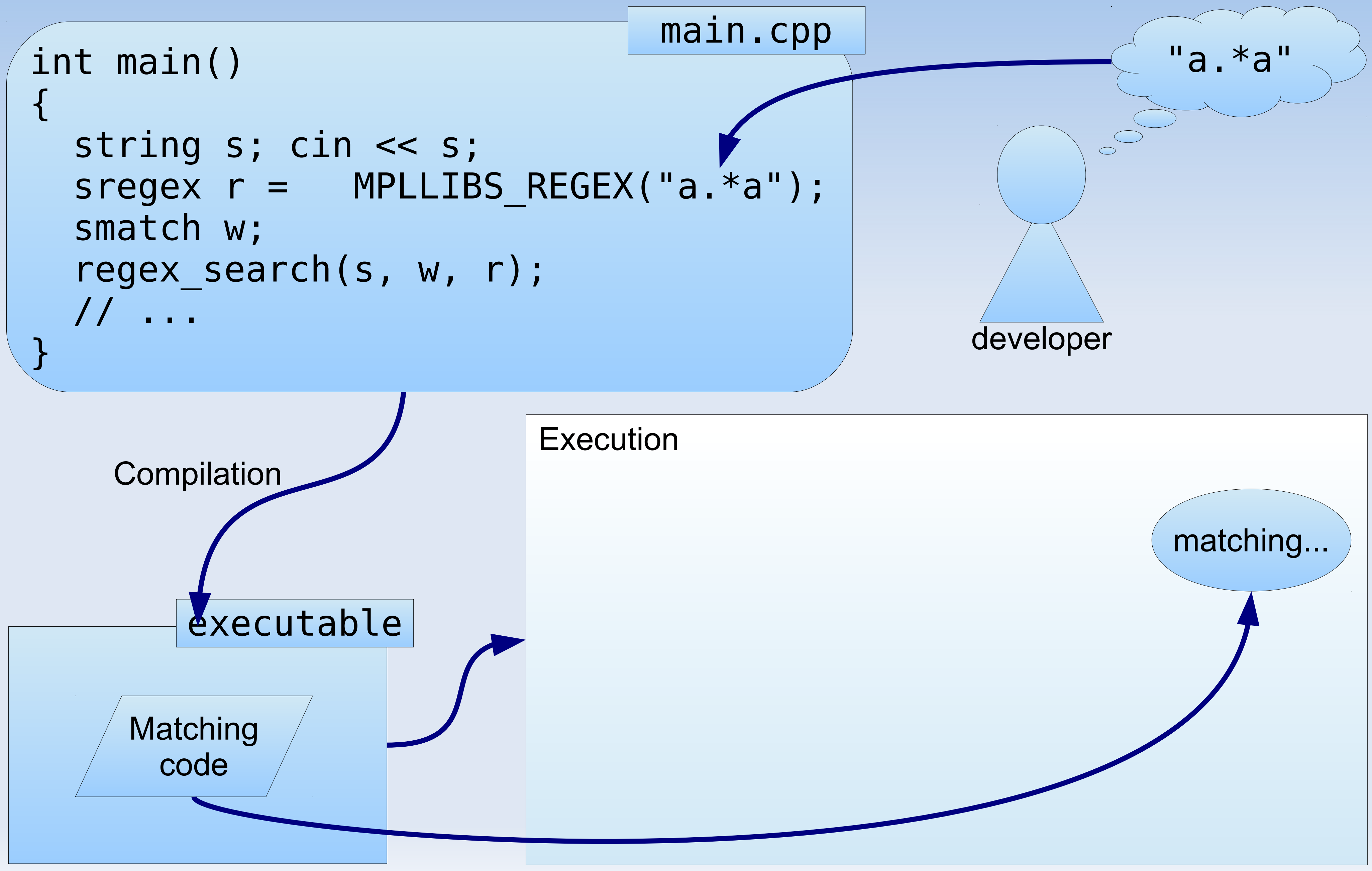
```
int main()
{
  string s; cin << s;
  sregex r =as_xpr('a') >> *_ >> 'a';
  smatch w;
  regex_search(s, w, r);
  // ...
}
```

Processing DSLs



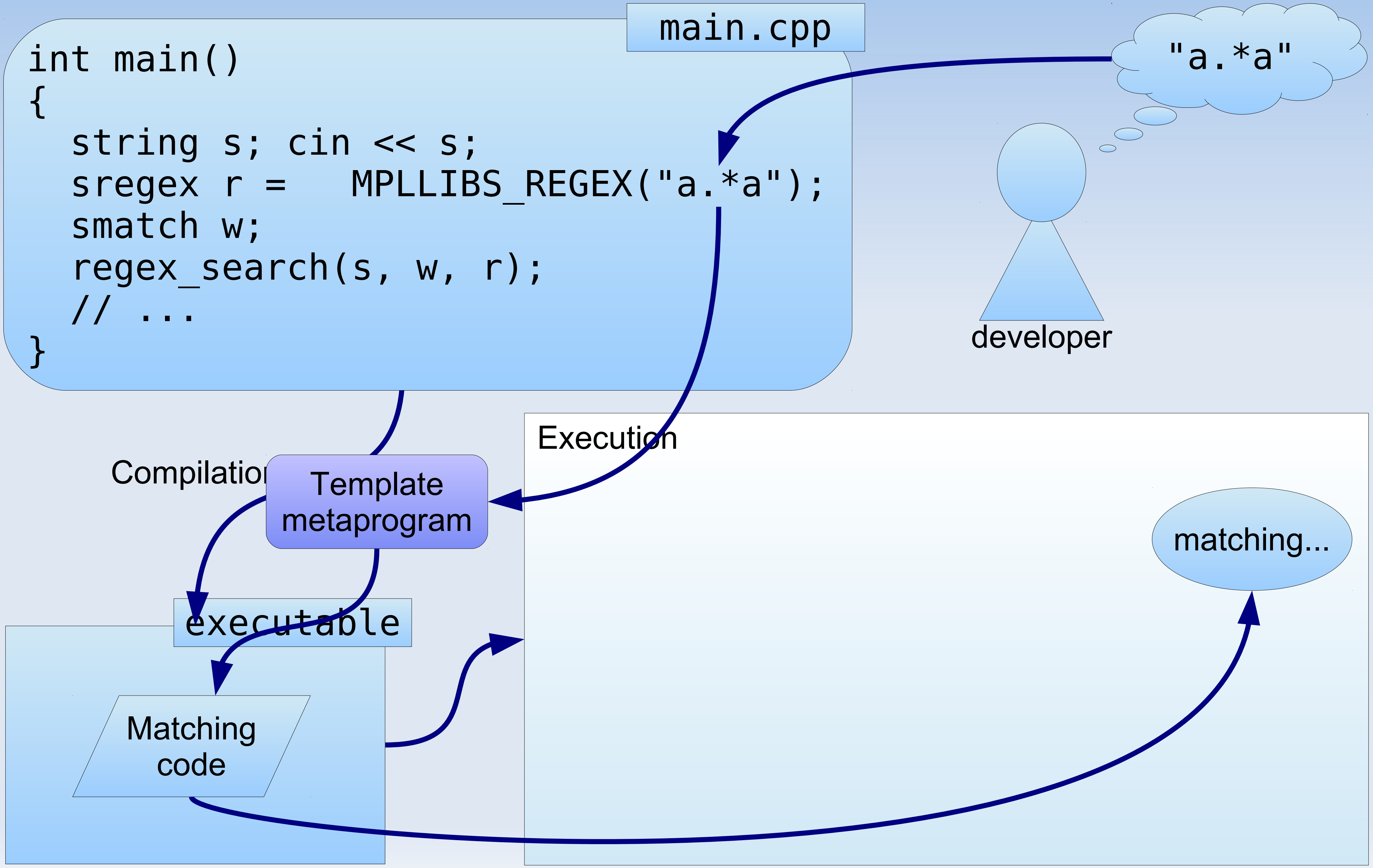
```
int main()
{
  string s; cin << s;
  sregex r =as_xpr('a') >> *_ >> 'a';
  smatch w;
  regex_search(s, w, r);
  // ...
}
```

Processing DSLs



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

Processing DSLs



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

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("abbbbbbbc", 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')
```

Boost.Xpressive

- Dynamic regex

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

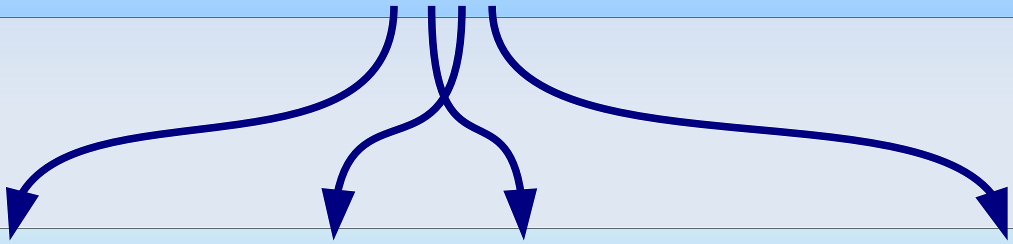
- Static regex

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

Boost.Xpressive

- Dynamic regex

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



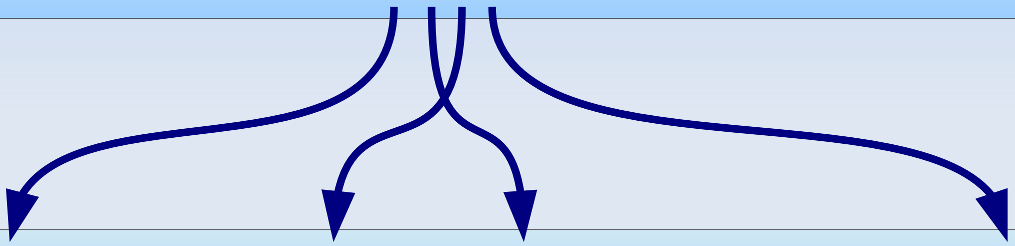
- Static regex

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

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");
```

$$\big\downarrow$$

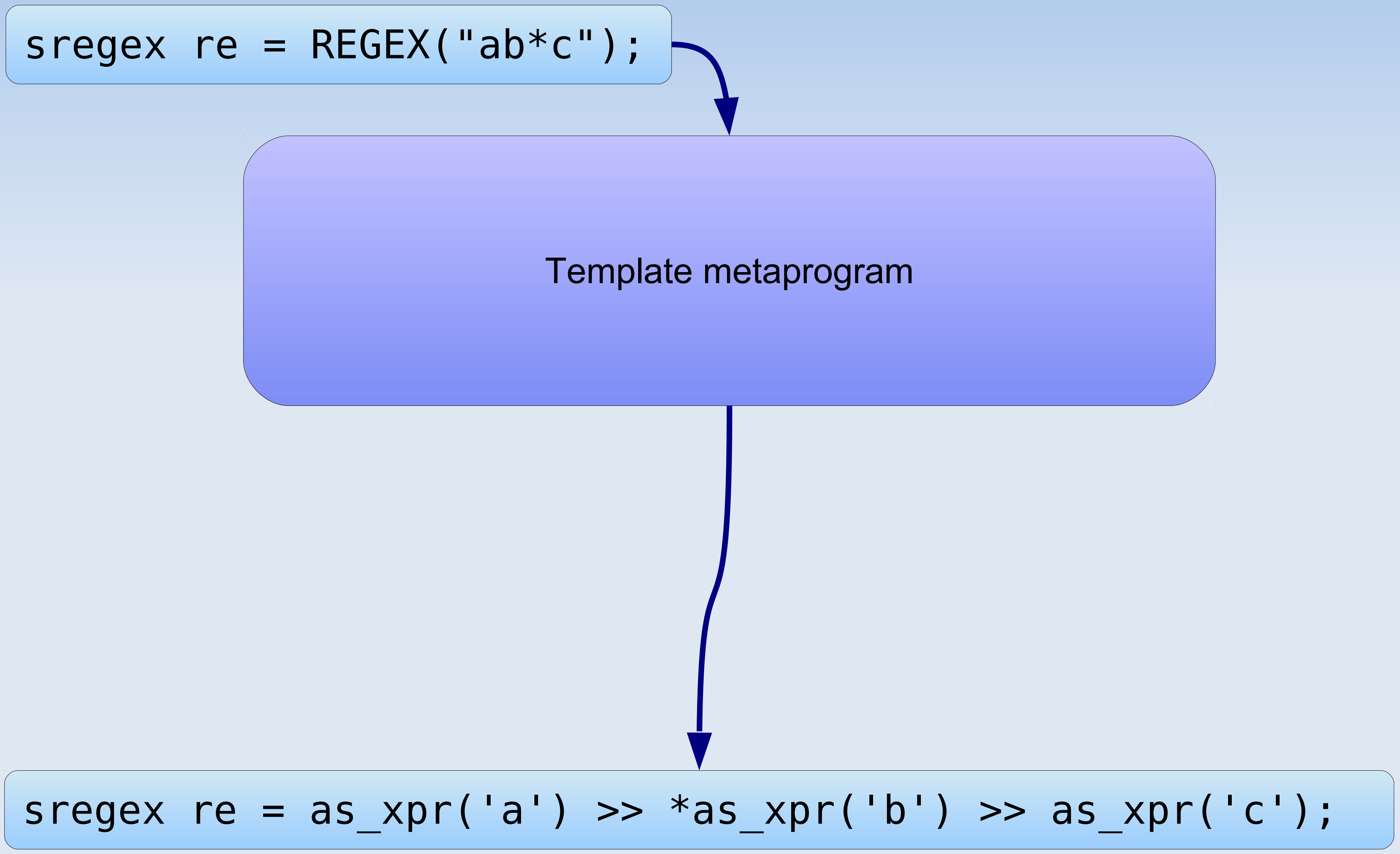
```
sregex re = as_xpr('a') >> *as_xpr('b') >> as_xpr('c');
```

Safe static regexes

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

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

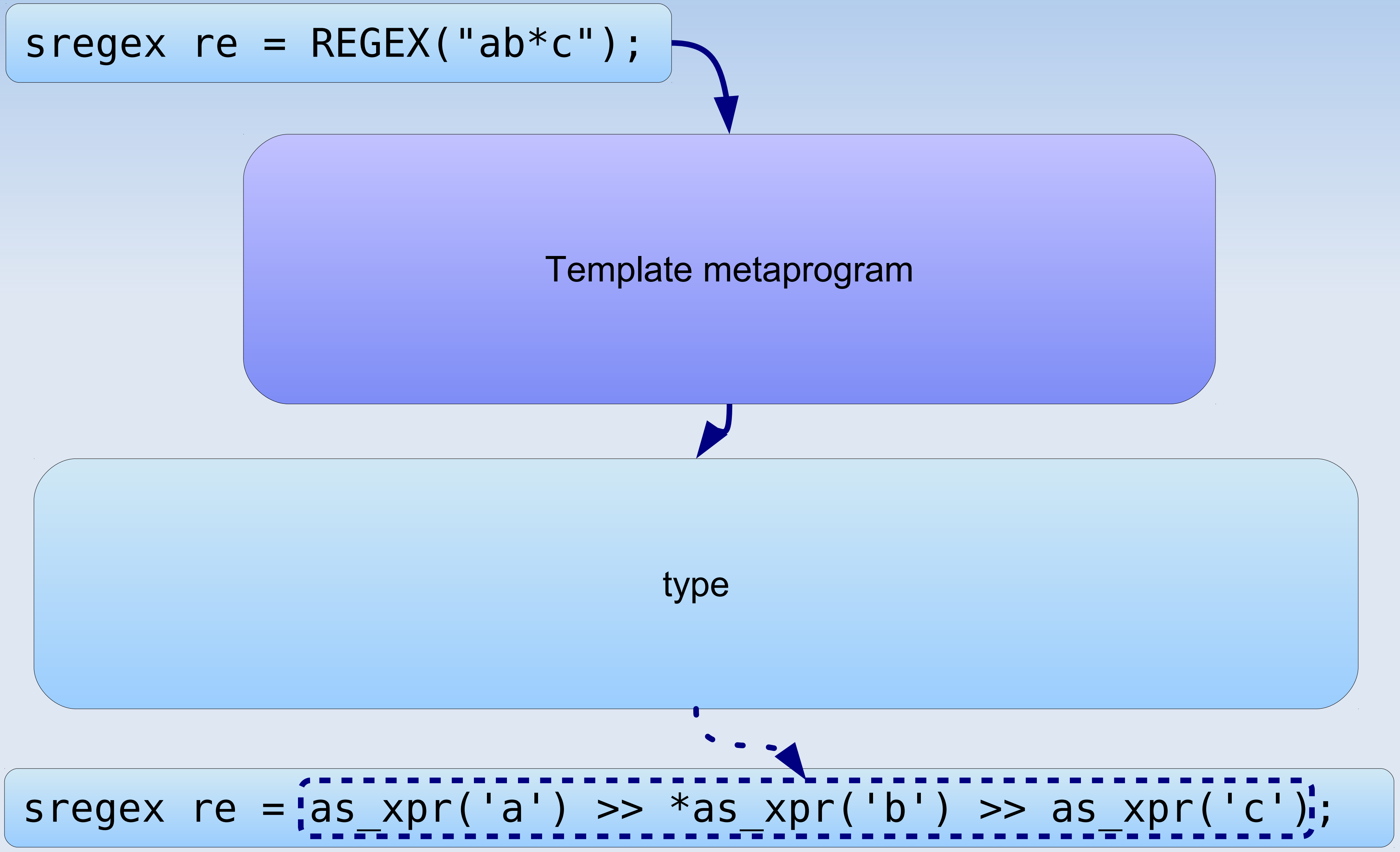
Safe static regexes



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

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

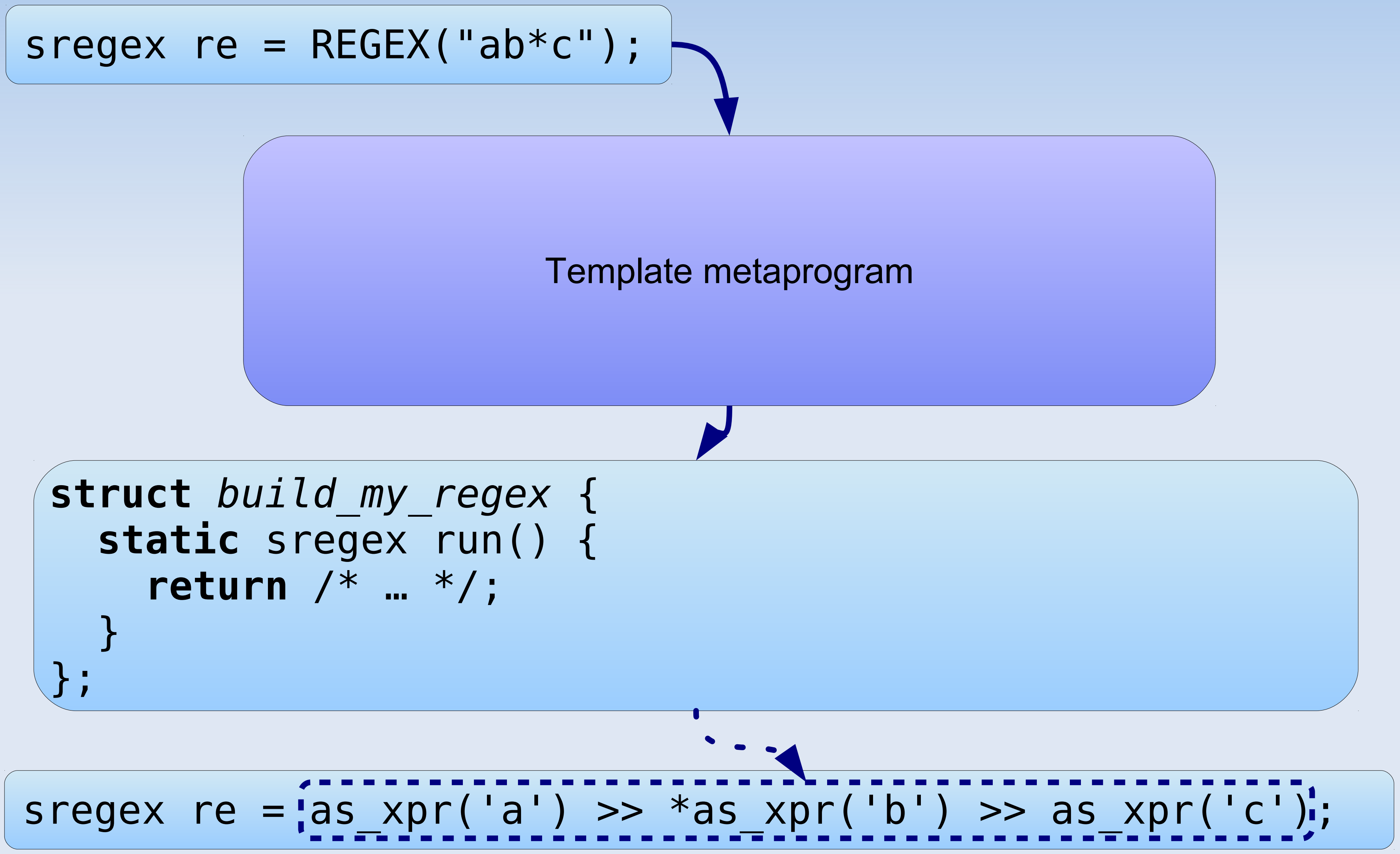
Safe static regexes



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

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

Safe static regexes

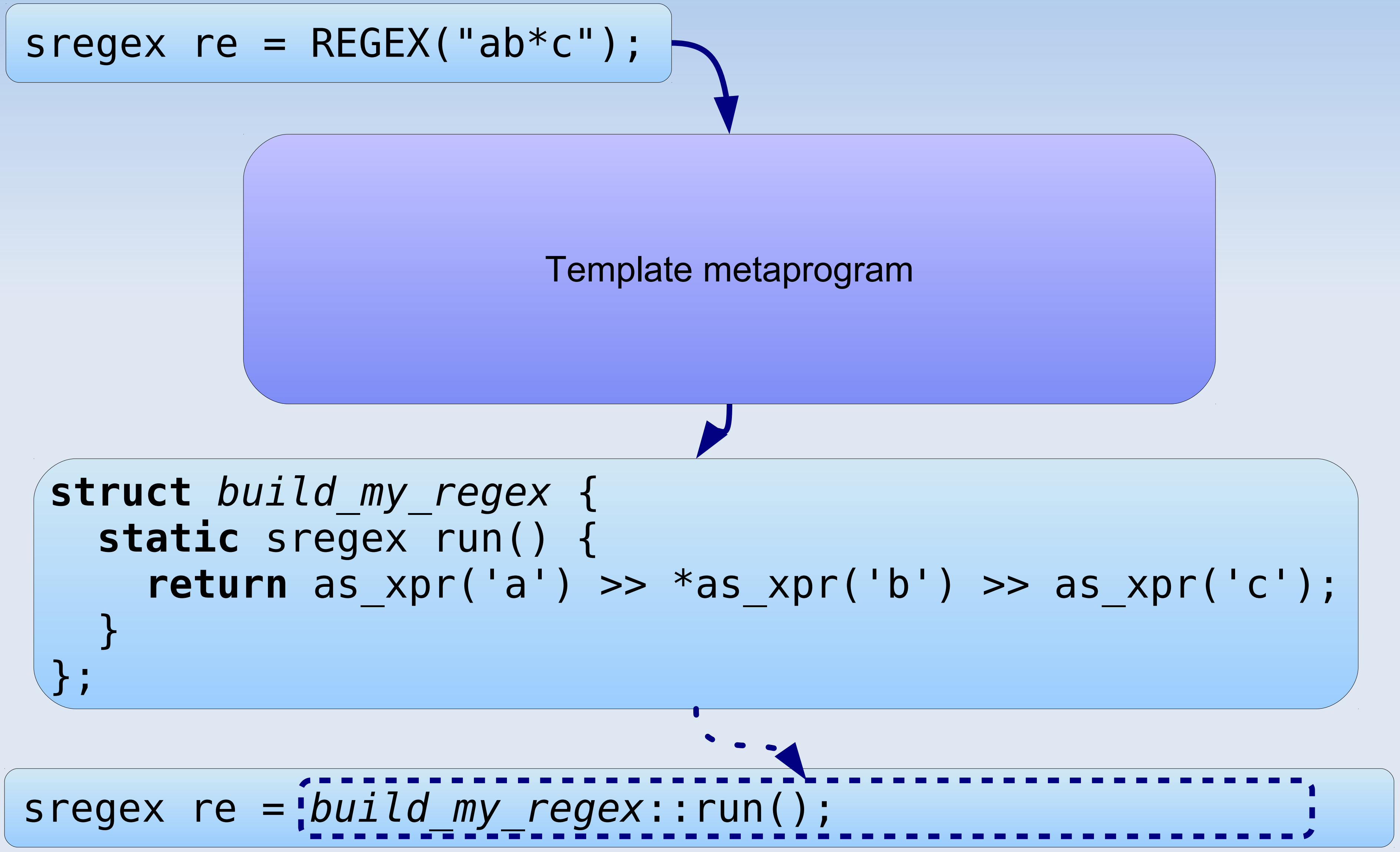


```
sregex re = REGEX("ab*c");
```

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

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

Safe static regexes



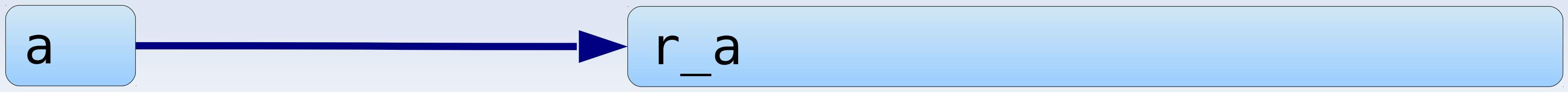
```
sregex re = REGEX("ab*c");
```

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

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

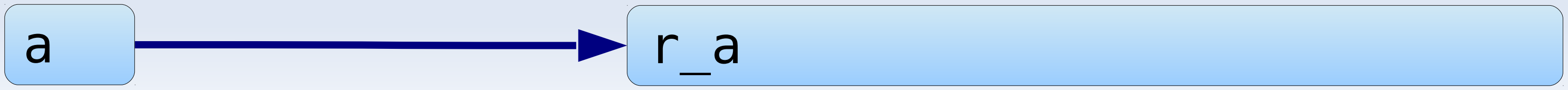
build_my_regex

```
struct r_a    {



};
```



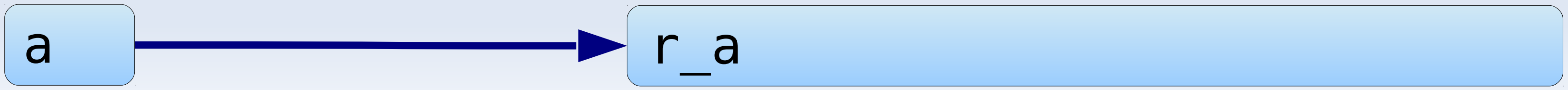
build_my_regex

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



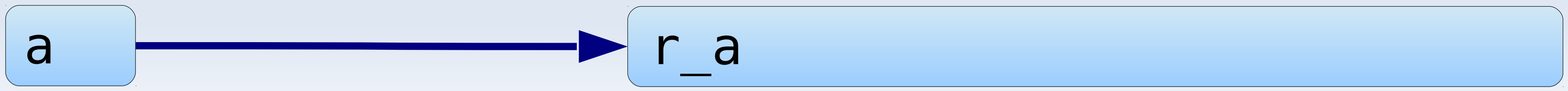
build_my_regex

```
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');
  }
};
```

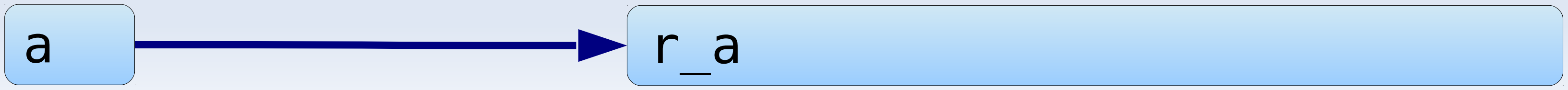
```
#define RUN(...) \
  static decltype((__VA_ARGS__)) run() { \
    return (__VA_ARGS__); \
  }
```

$$\begin{tikzcd} \text{a} \arrow[r] & \text{r\_a} \end{tikzcd}$$

build_my_regex

```
struct r_a    {

  RUN(as_xpr('a'           ))

};
```

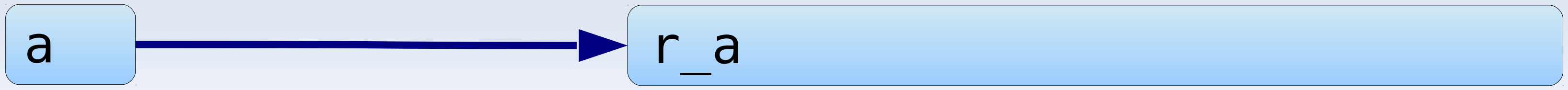
```
#define RUN(...) \
  static decltype((__VA_ARGS__)) run() { \
    return (__VA_ARGS__); \
  }
```


$$\begin{tikzcd}\boxed{a} \arrow[r] & \boxed{r\_a}\end{tikzcd}$$

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             ))

};
```

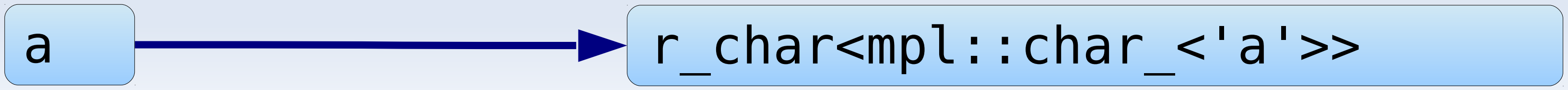
```
#define RUN(...) \
  static decltype((__VA_ARGS__)) run() { \
    return (__VA_ARGS__); \
  }
```

$$\begin{tikzcd}\texttt{a} \arrow[r] & \texttt{r\_char<   'a' >}\end{tikzcd}$$

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

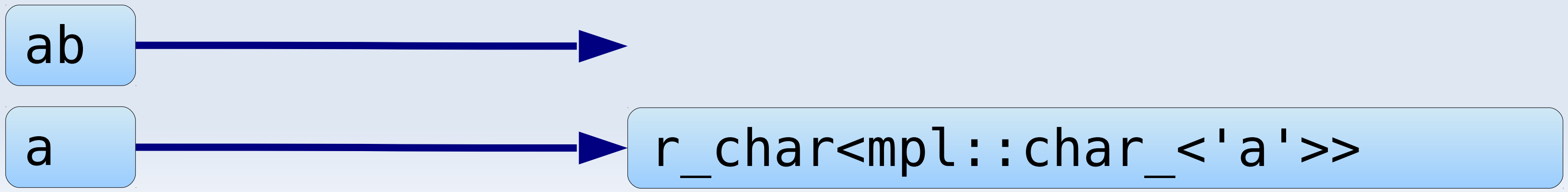
```
template <class C>
struct r_char {
  typedef r_char type;

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

};
```

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

$$\begin{tikzcd}\boxed{\texttt{a}} \arrow[r] & \boxed{\texttt{r\_char<mpl::char\_<'a'>>}}\end{tikzcd}$$

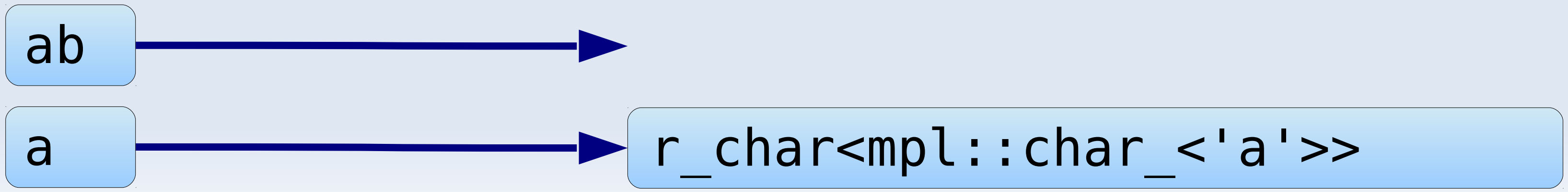
build_my_regex

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

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

};
```



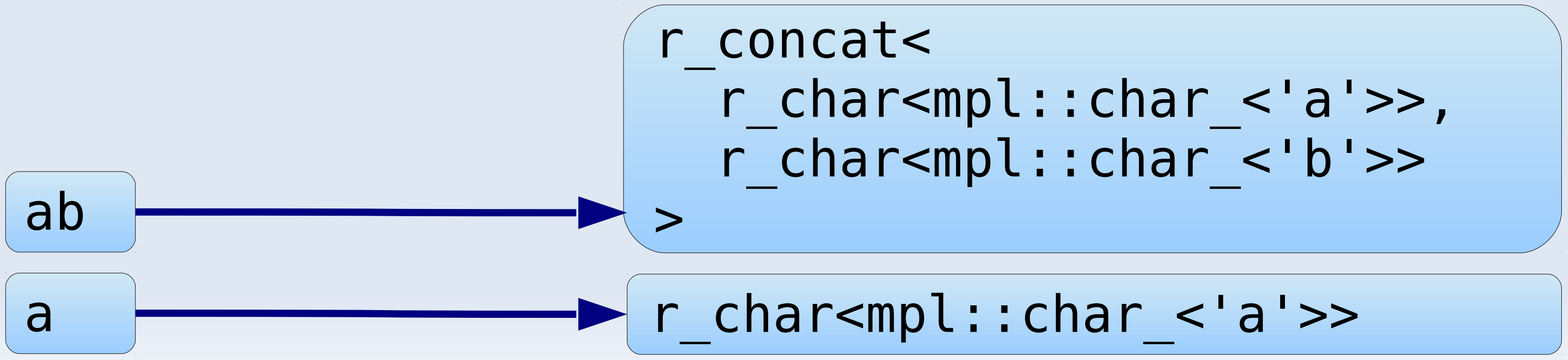
build_my_regex

```
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;

};
```

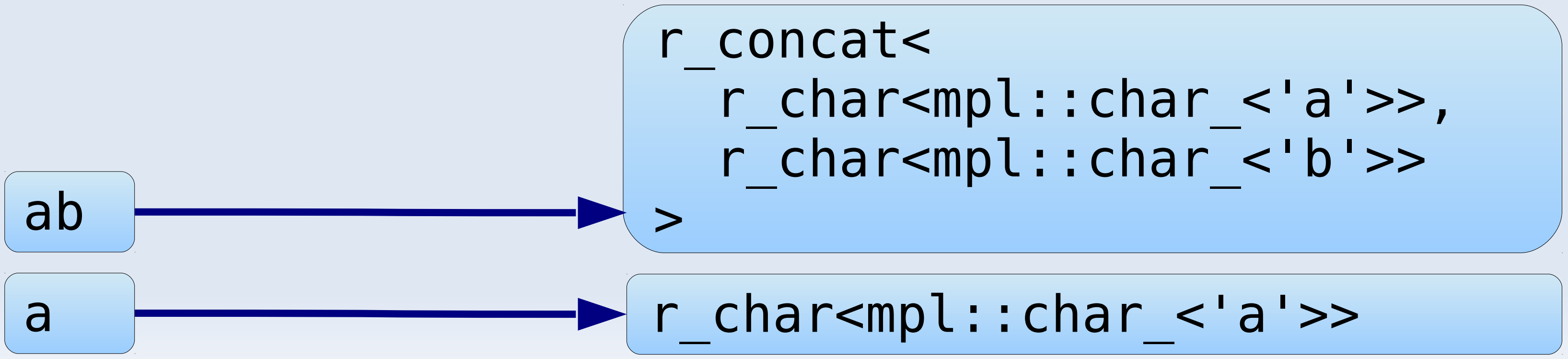


```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

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

build_my_regex

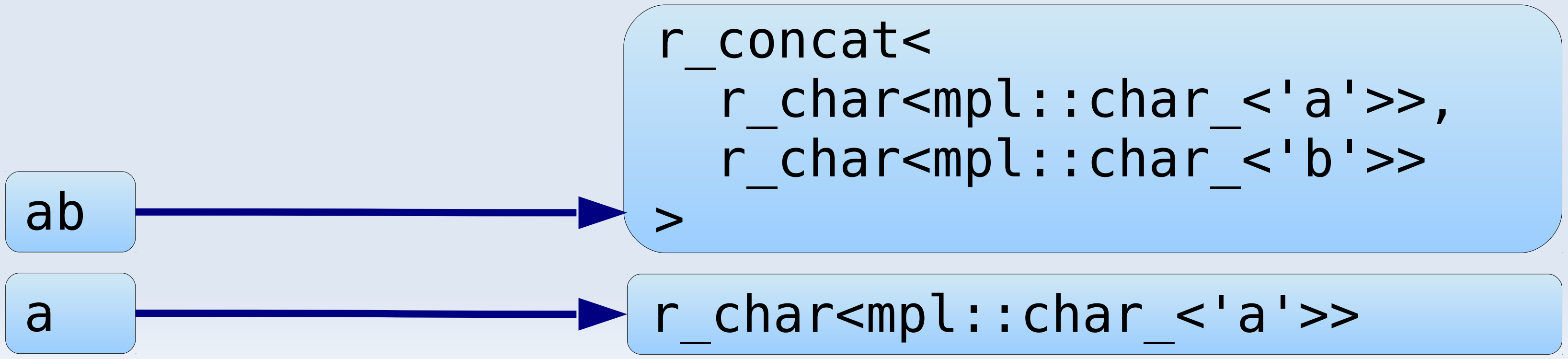
```
template <class C>
struct r_char {
  typedef r_char type;

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

};

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



```
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()                        )
};
```

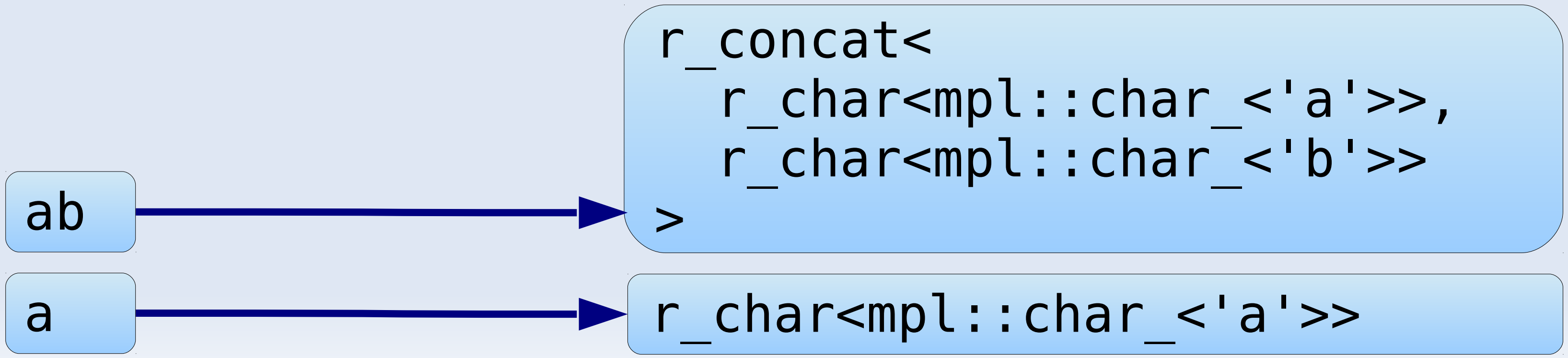


```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

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

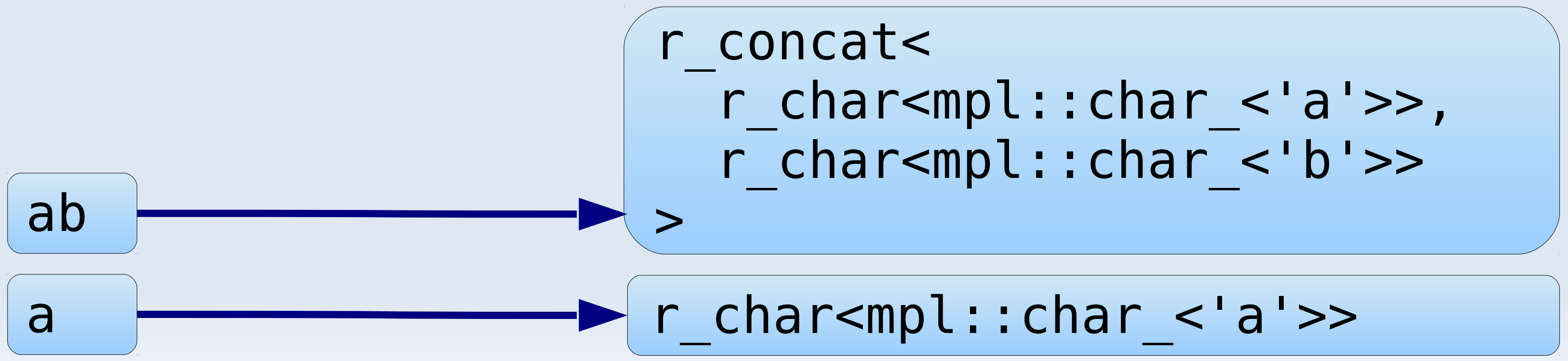
build_my_regex

```
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::run()          B::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::run()       >> B::run()      )
};
```

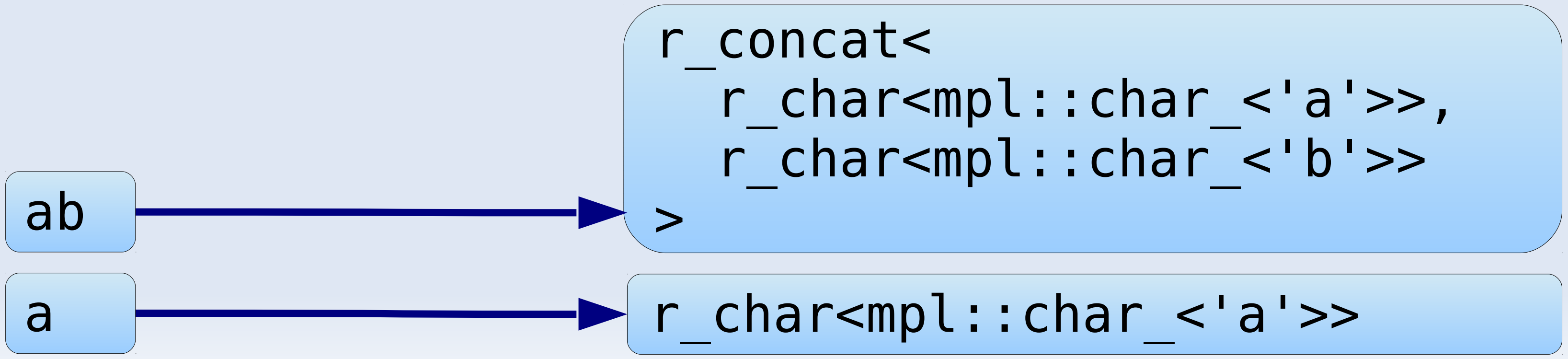


```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

```
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())
};
```

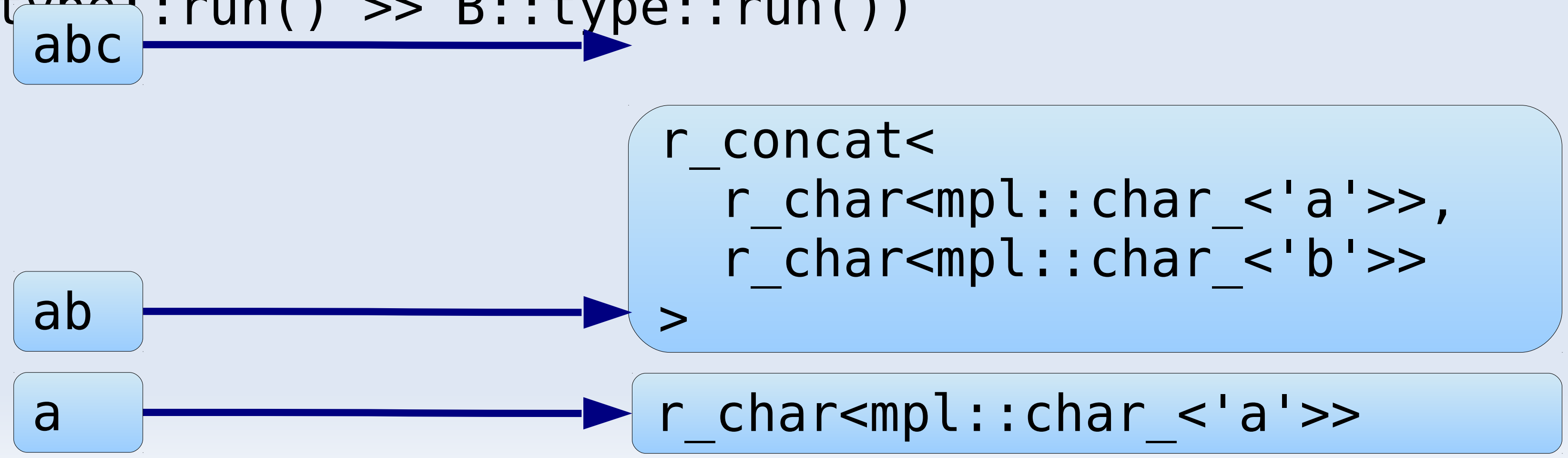


```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

```
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())
};
```

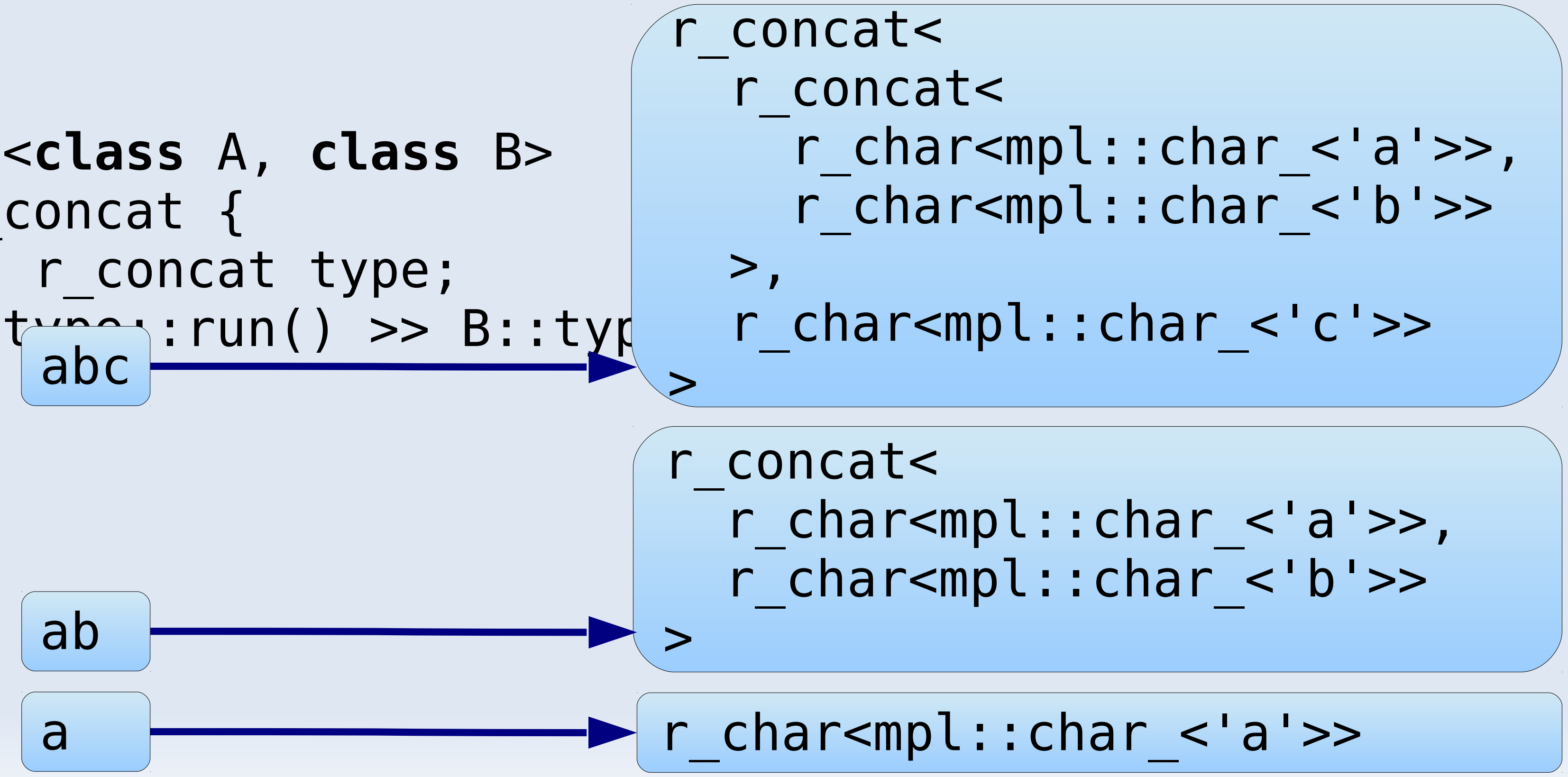


```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

```
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::typ
};
```

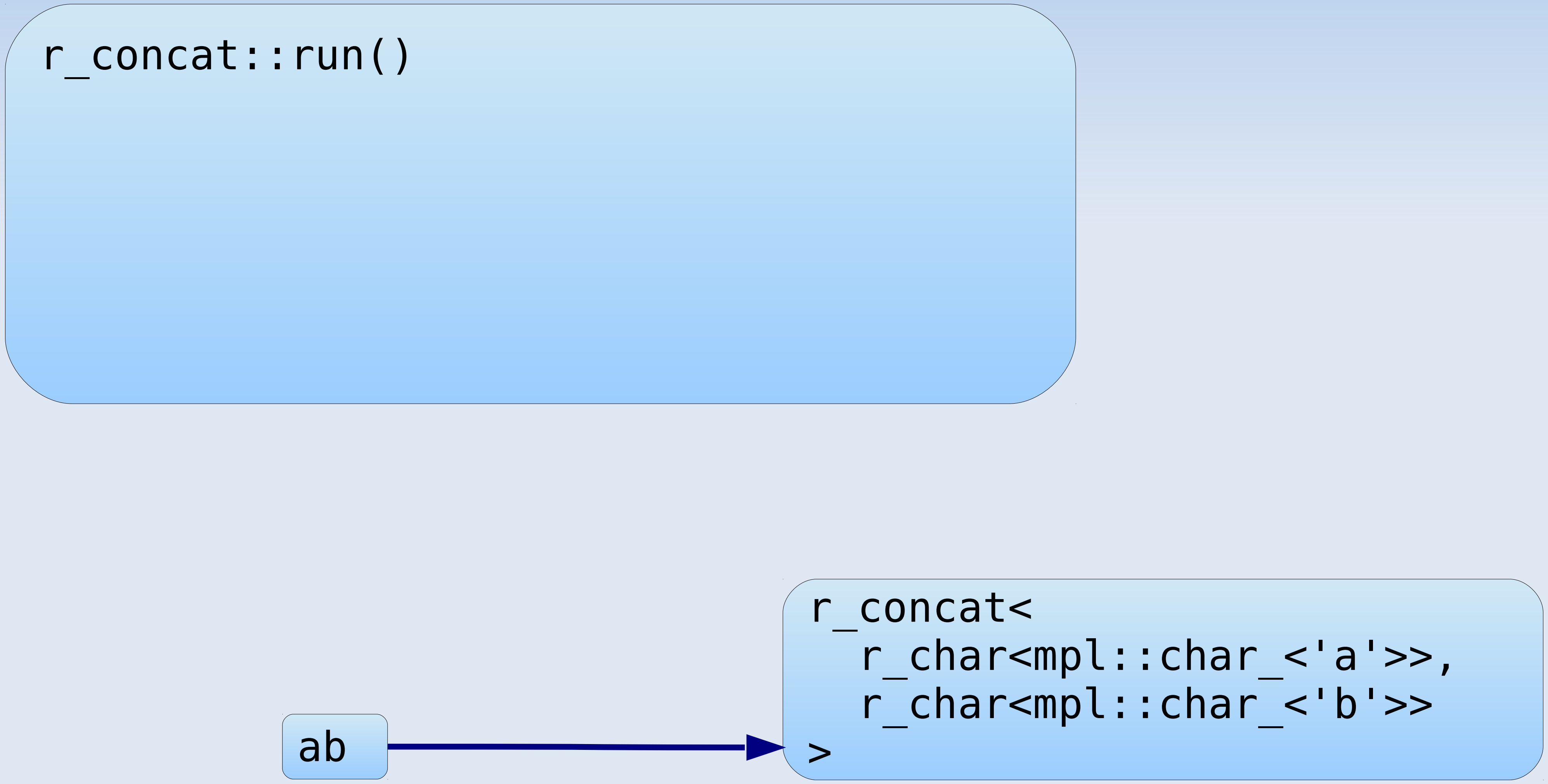


```
r_concat<
  r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
  >,
  r_char<mpl::char_<'c'>>
>
```

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

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

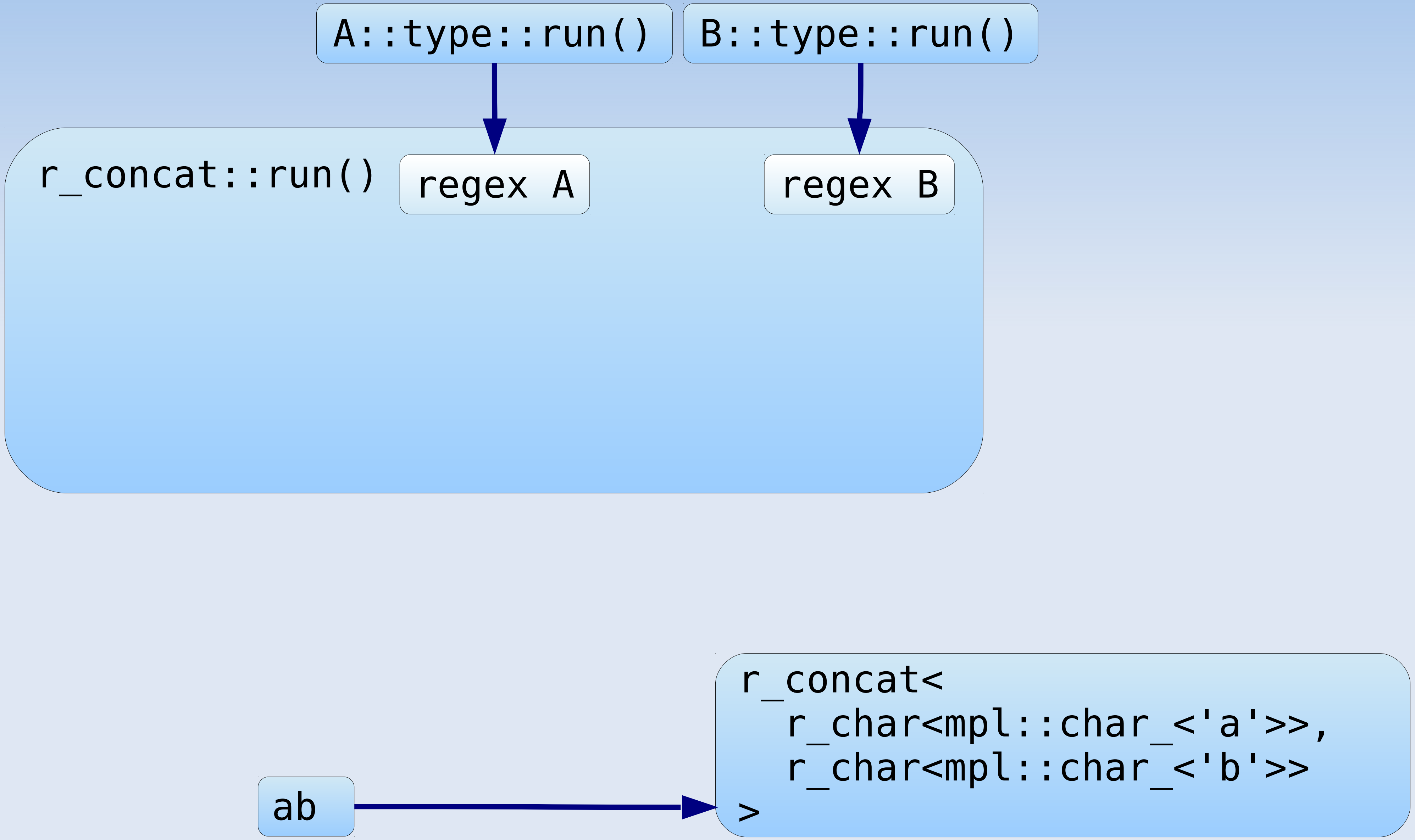
build_my_regex



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

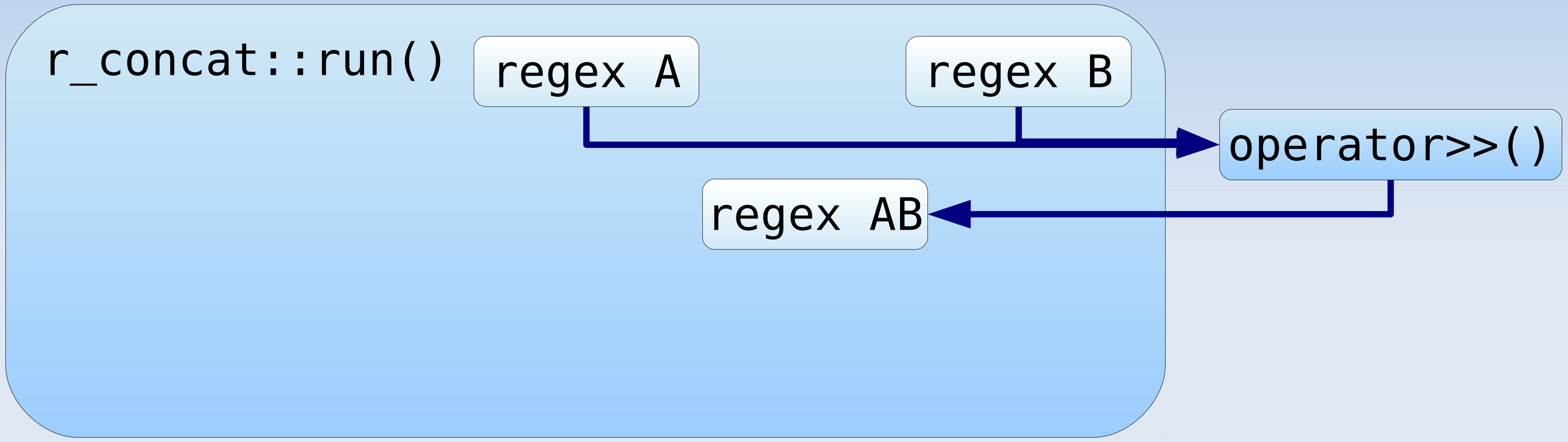
```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

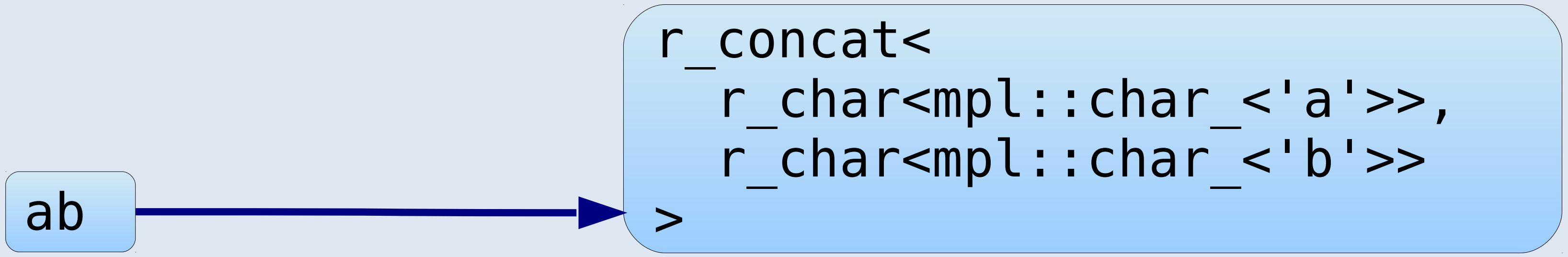
build_my_regex



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

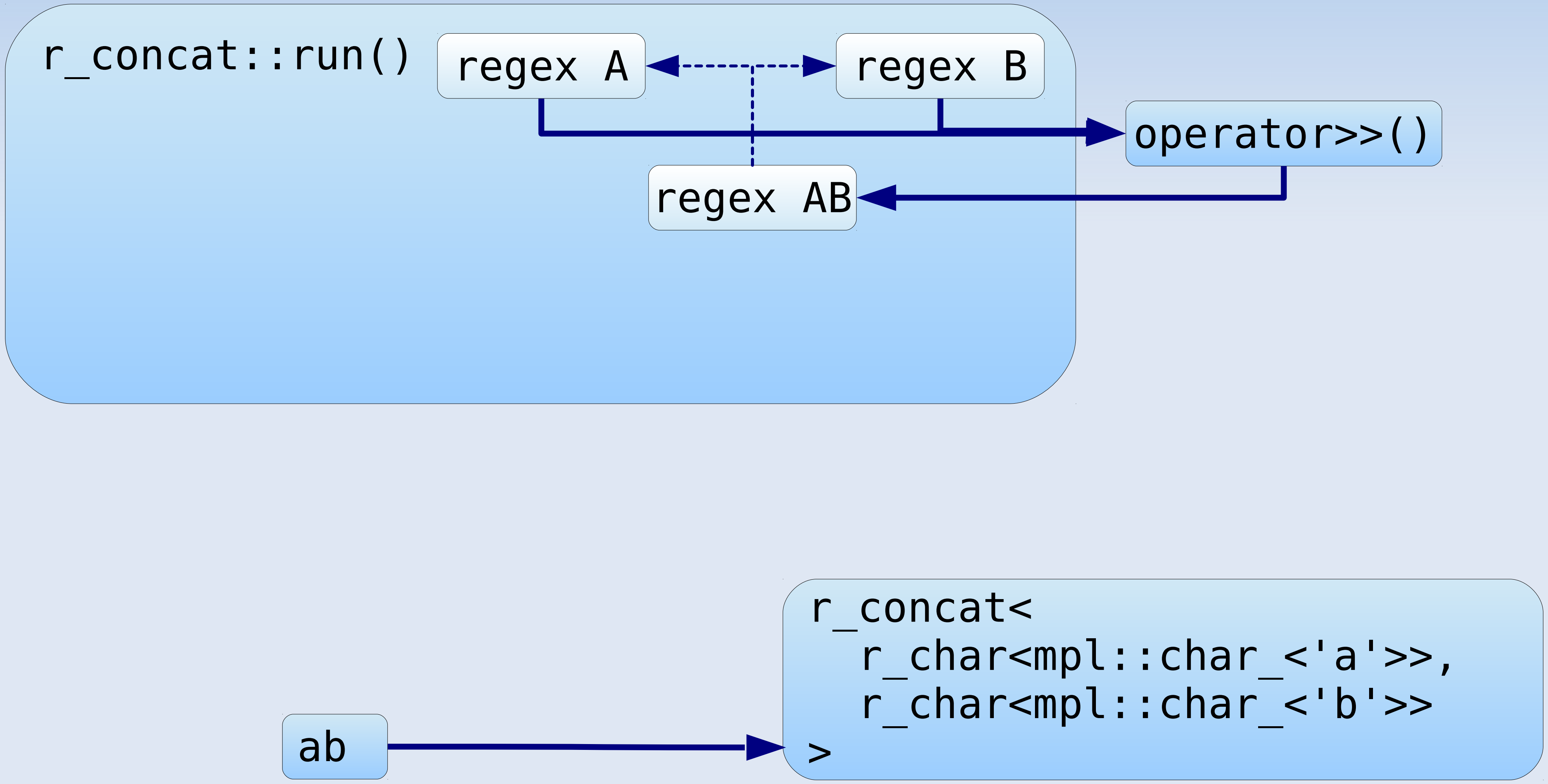
build_my_regex





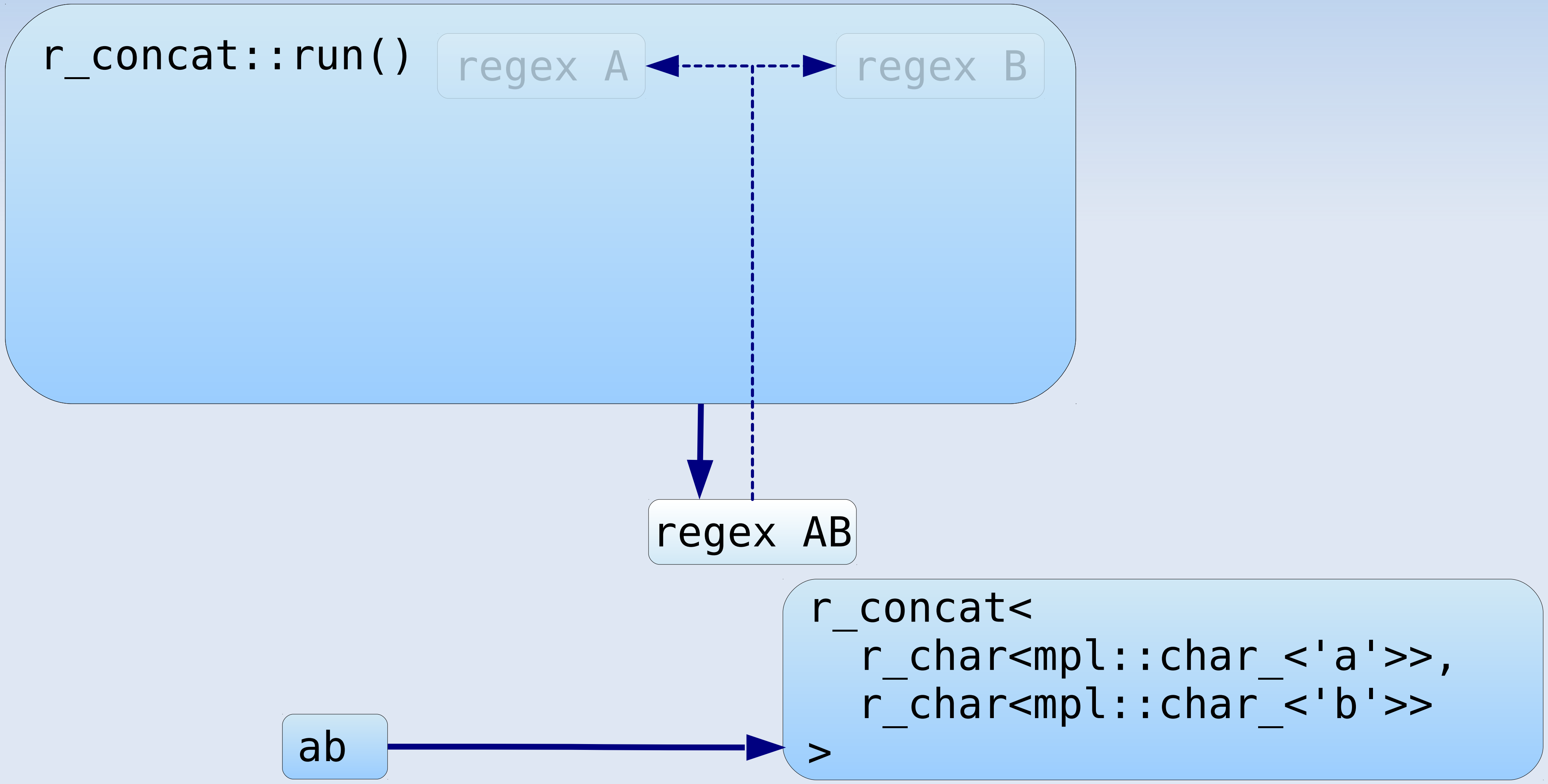
```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

build_my_regex



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

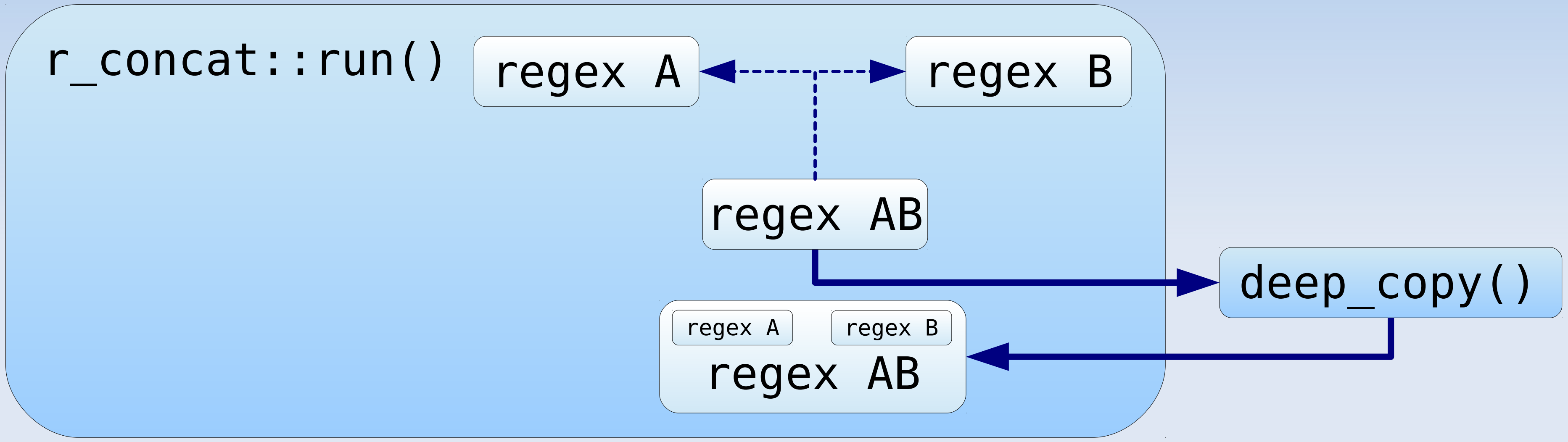
build_my_regex

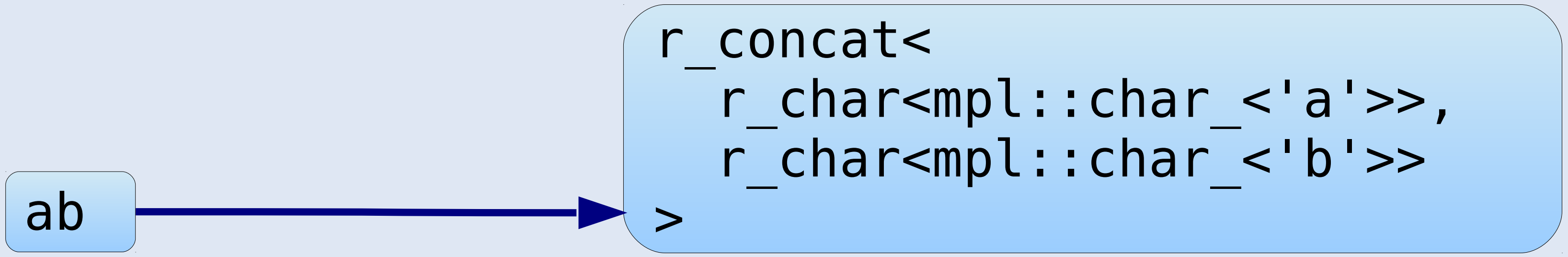


```
r_concat::run()
```

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

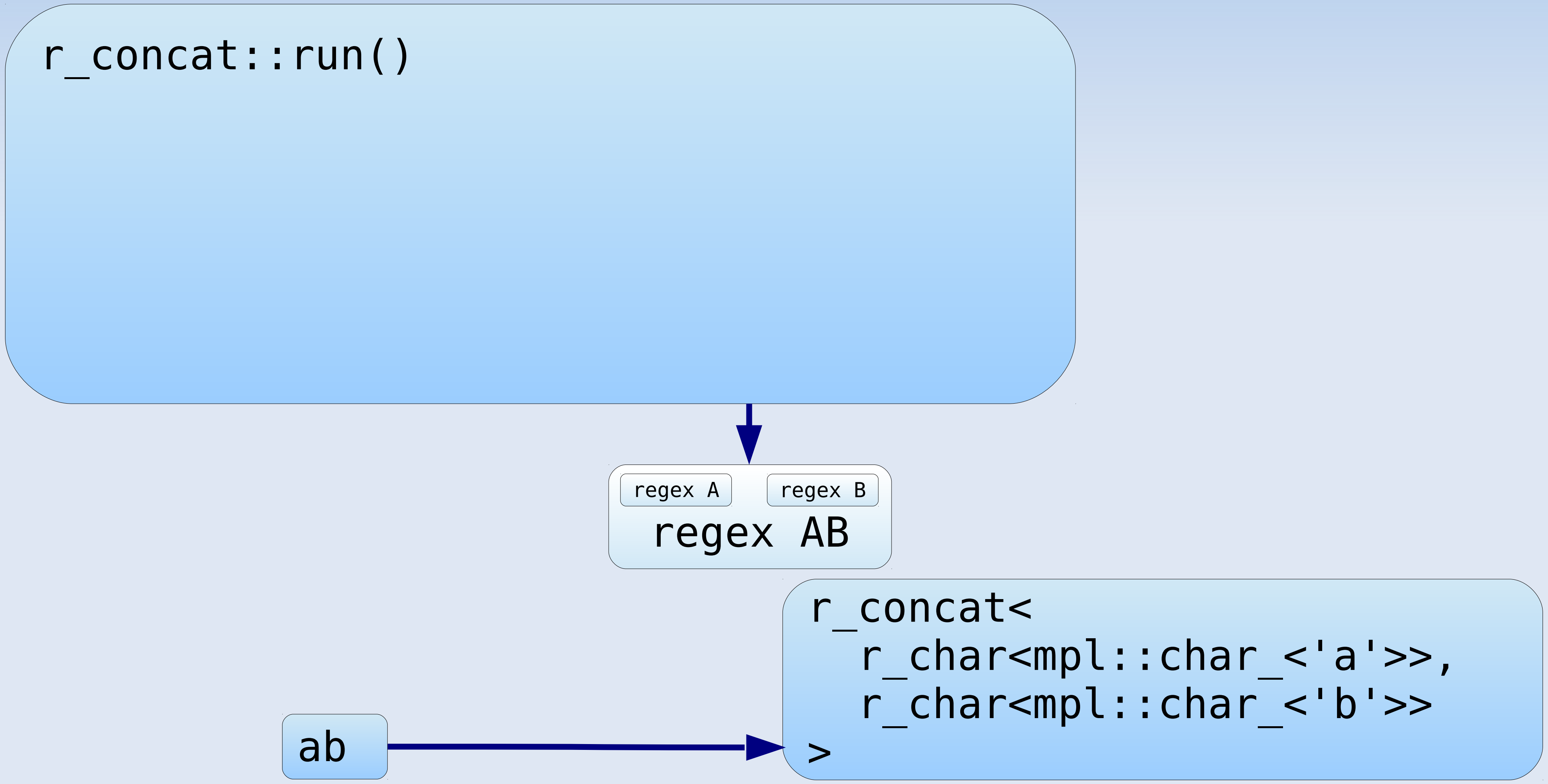
build_my_regex





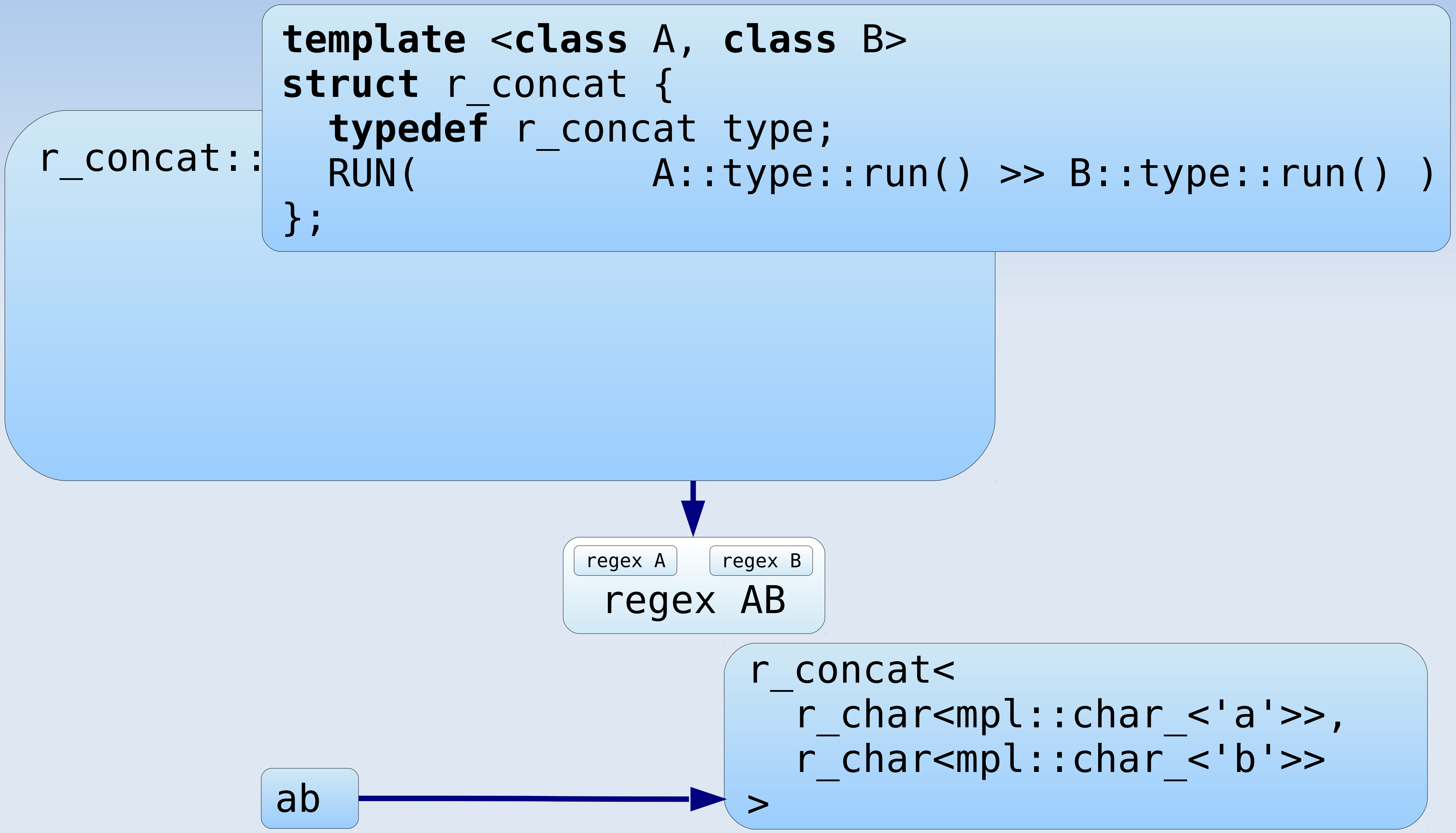
```
r_concat<
  r_char<mpl::char_<'a'>>,
  r_char<mpl::char_<'b'>>
>
```

build_my_regex



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

build_my_regex

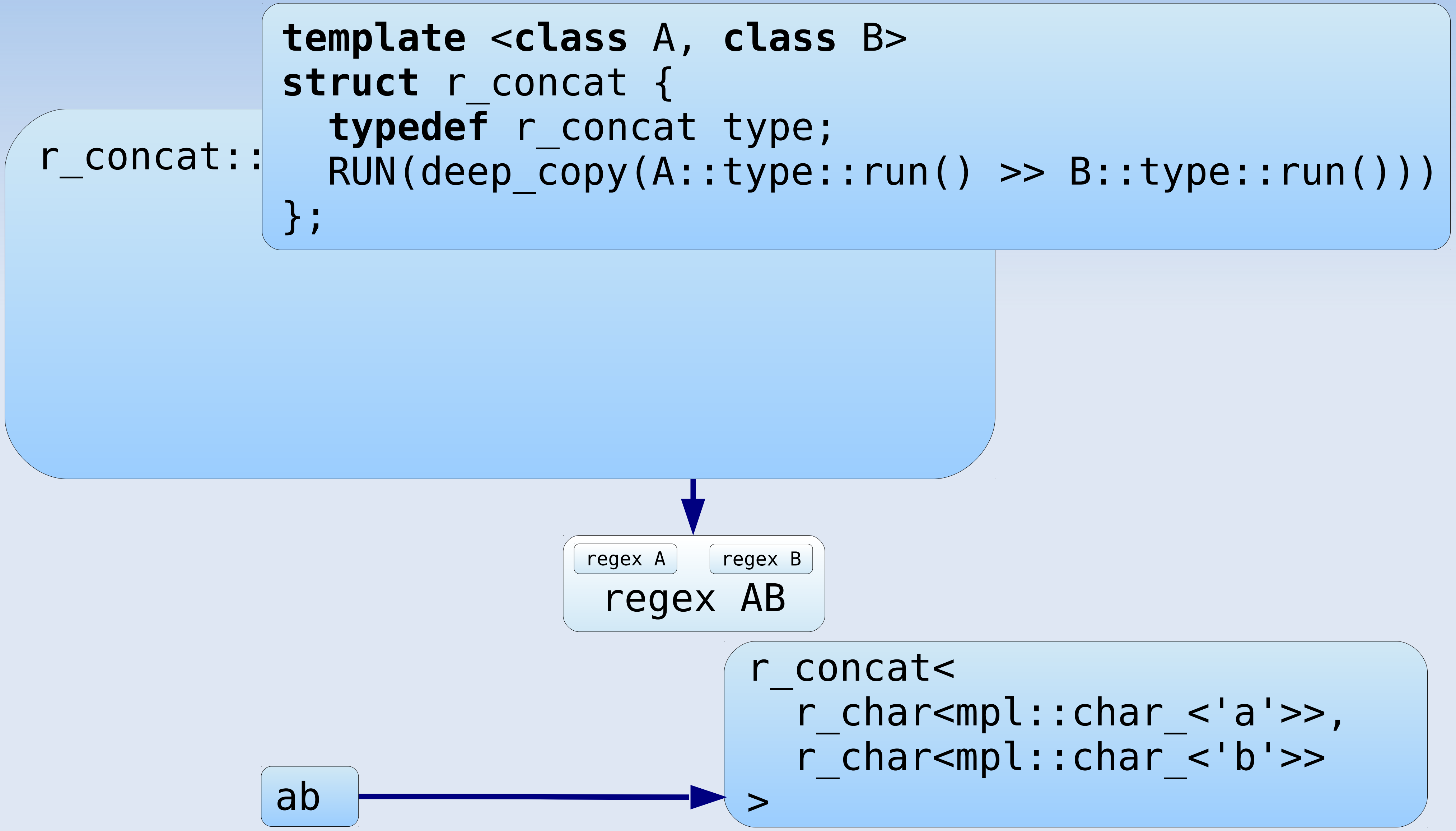


```
r_concat::
```

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

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

build_my_regex



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

```
r_concat::
```

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

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;
};
```

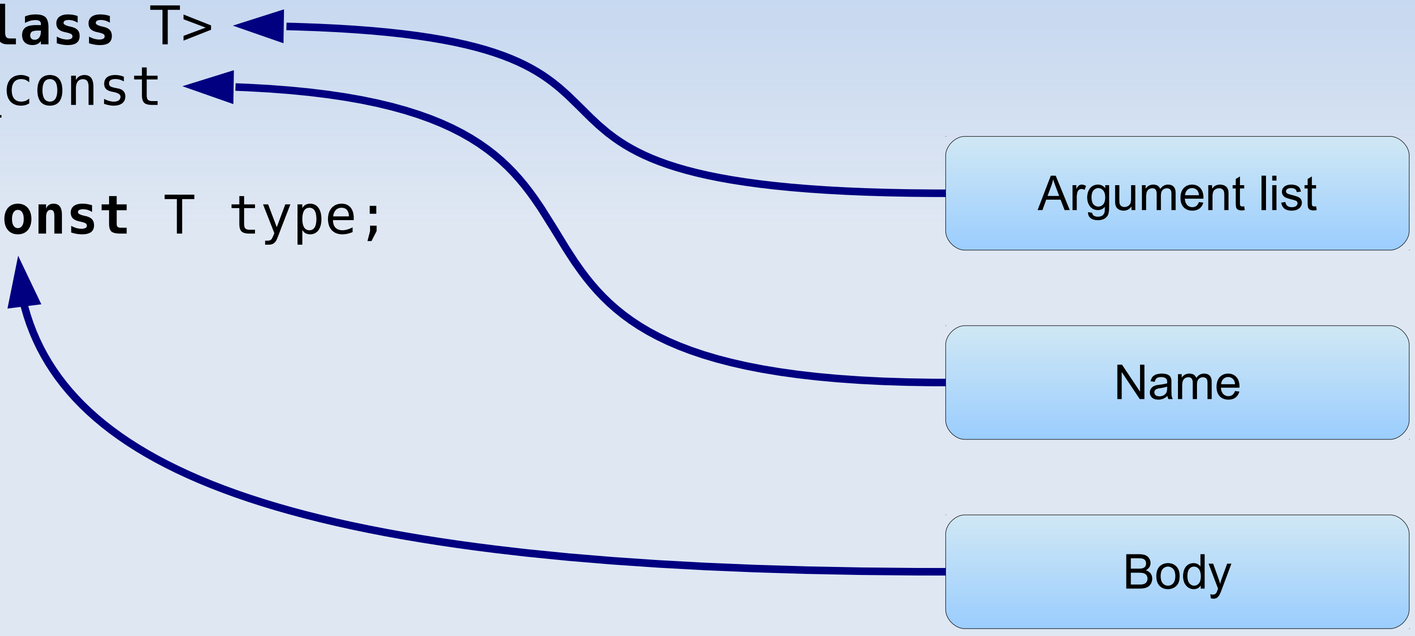
Argument list

Name

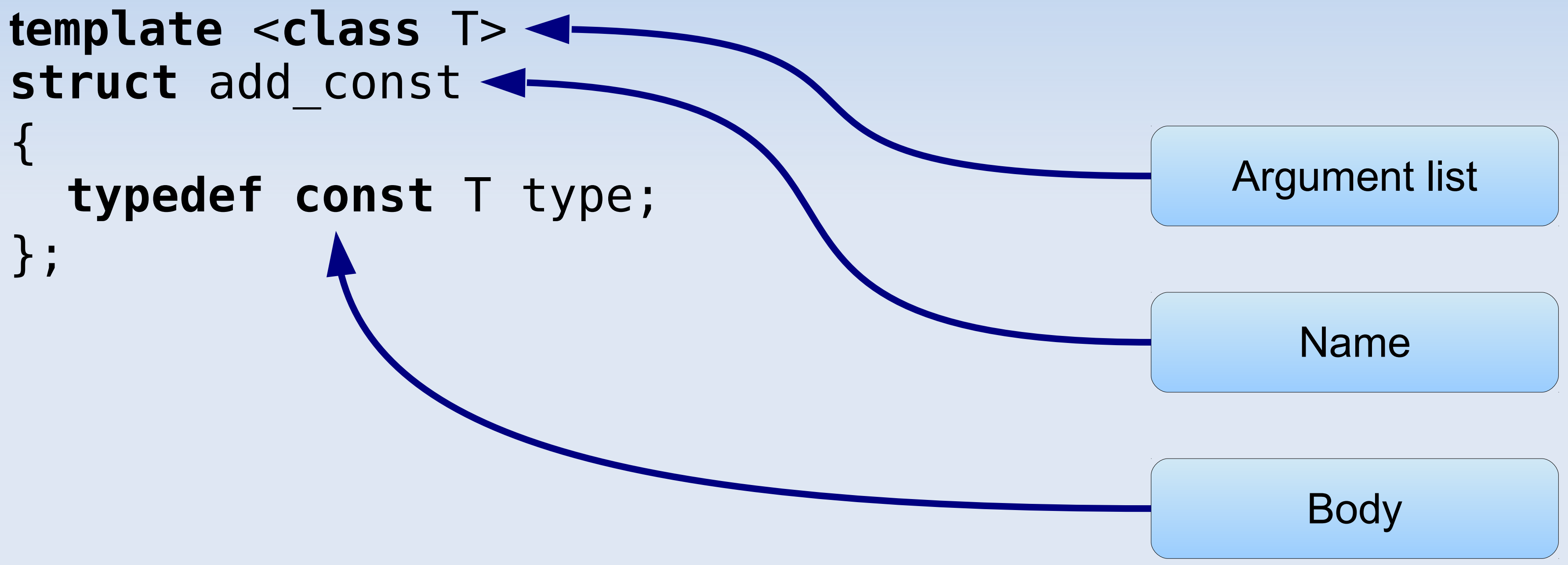
Body

C++ template metafunction

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



C++ template metafunction

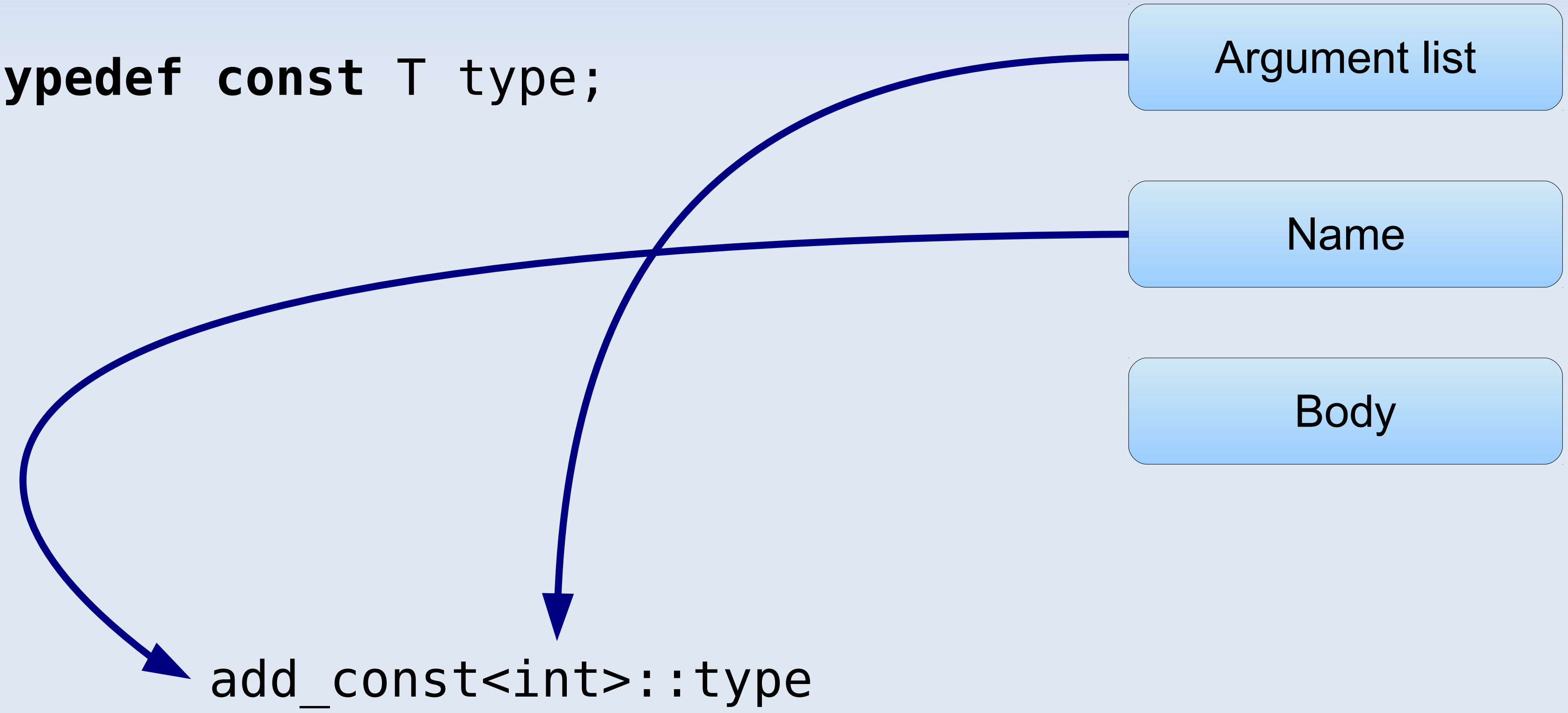


```
template <class T>
struct add_const
{
  typedef const T type;
};
```

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

C++ template metafunction

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



```
add_const<int>::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
{






};
```

C++ template metafunction

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

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

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

      typename add_const<T>::type

    type;
};
```

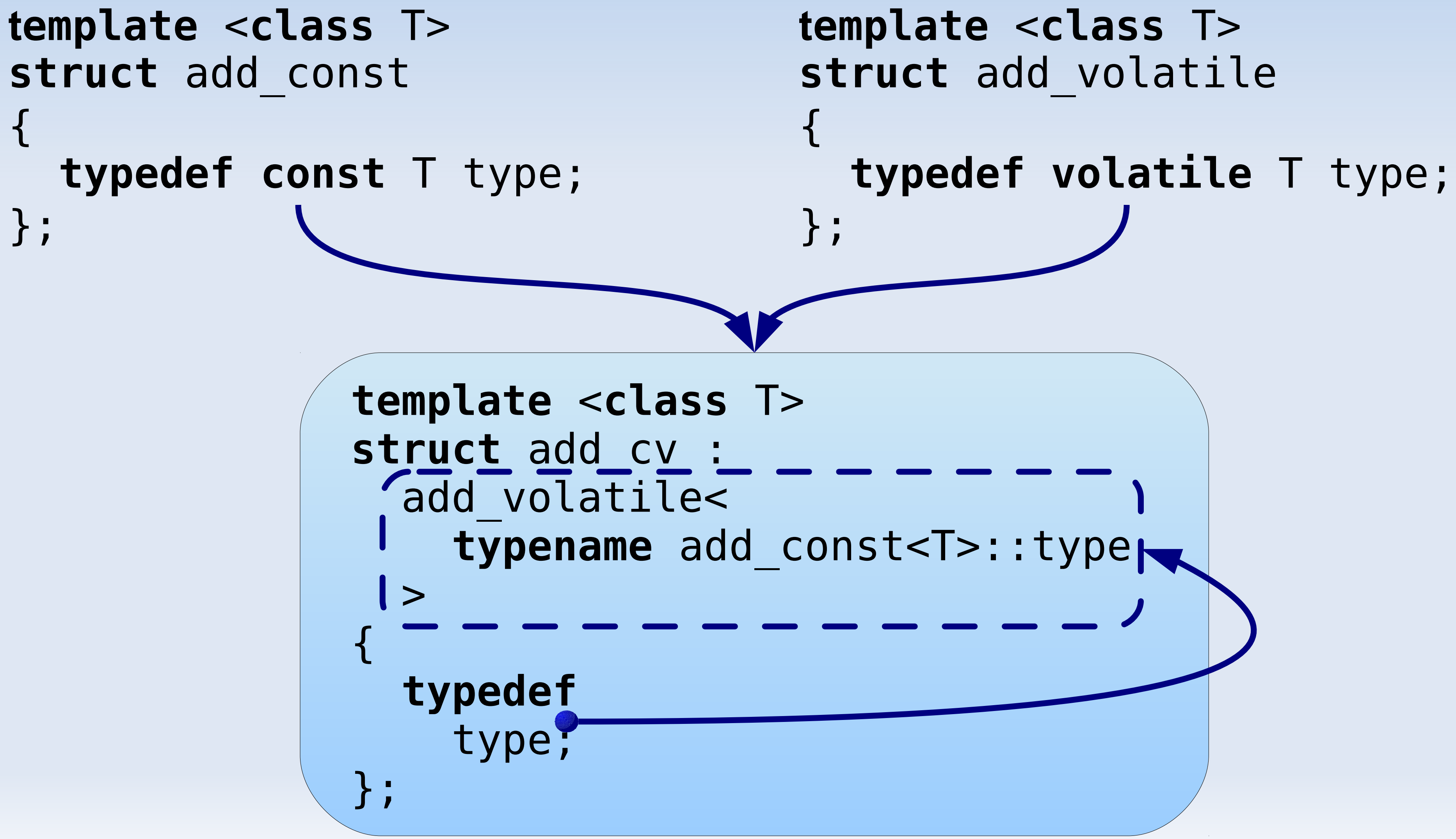
C++ template metafunction

```
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

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

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

```cpp
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"` → `('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
```

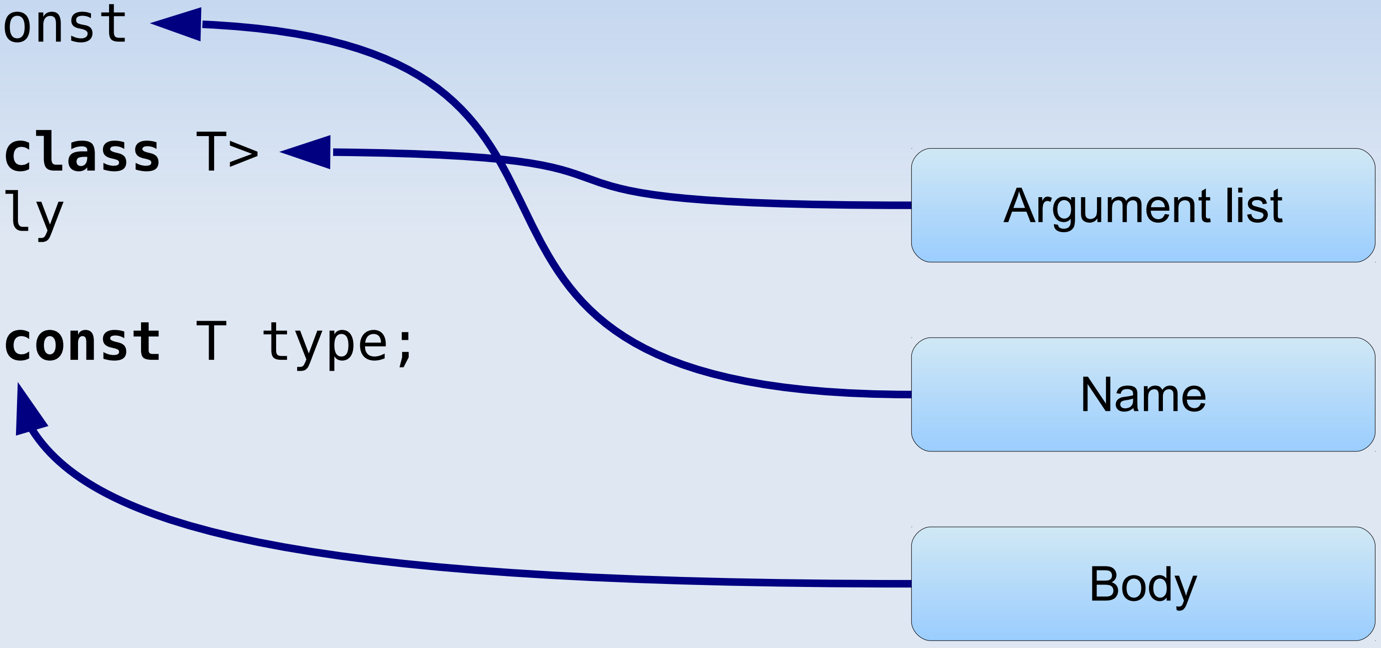
Higher order functions

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

};
```

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

Template metafunction class

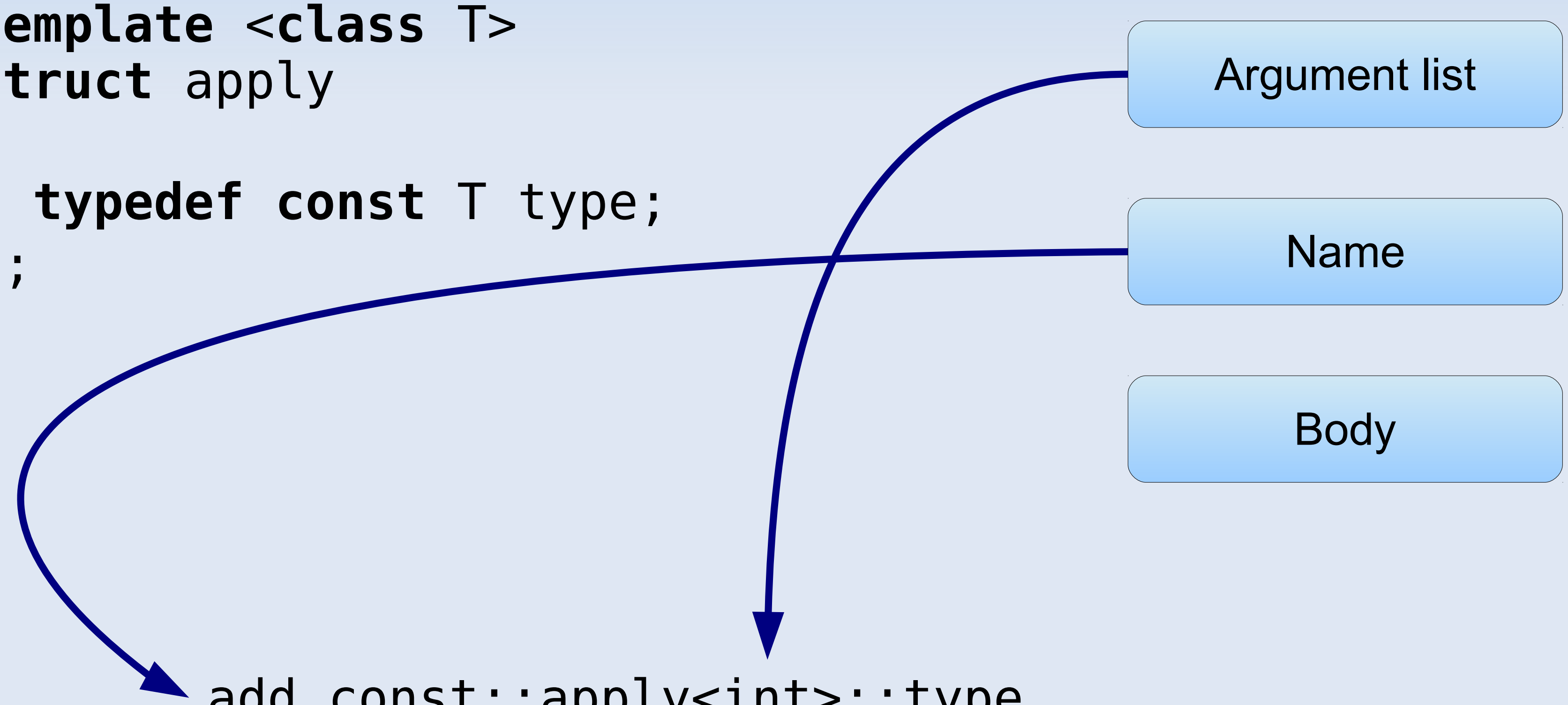
```
struct add_const
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };

};
```



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

Template metafunction class

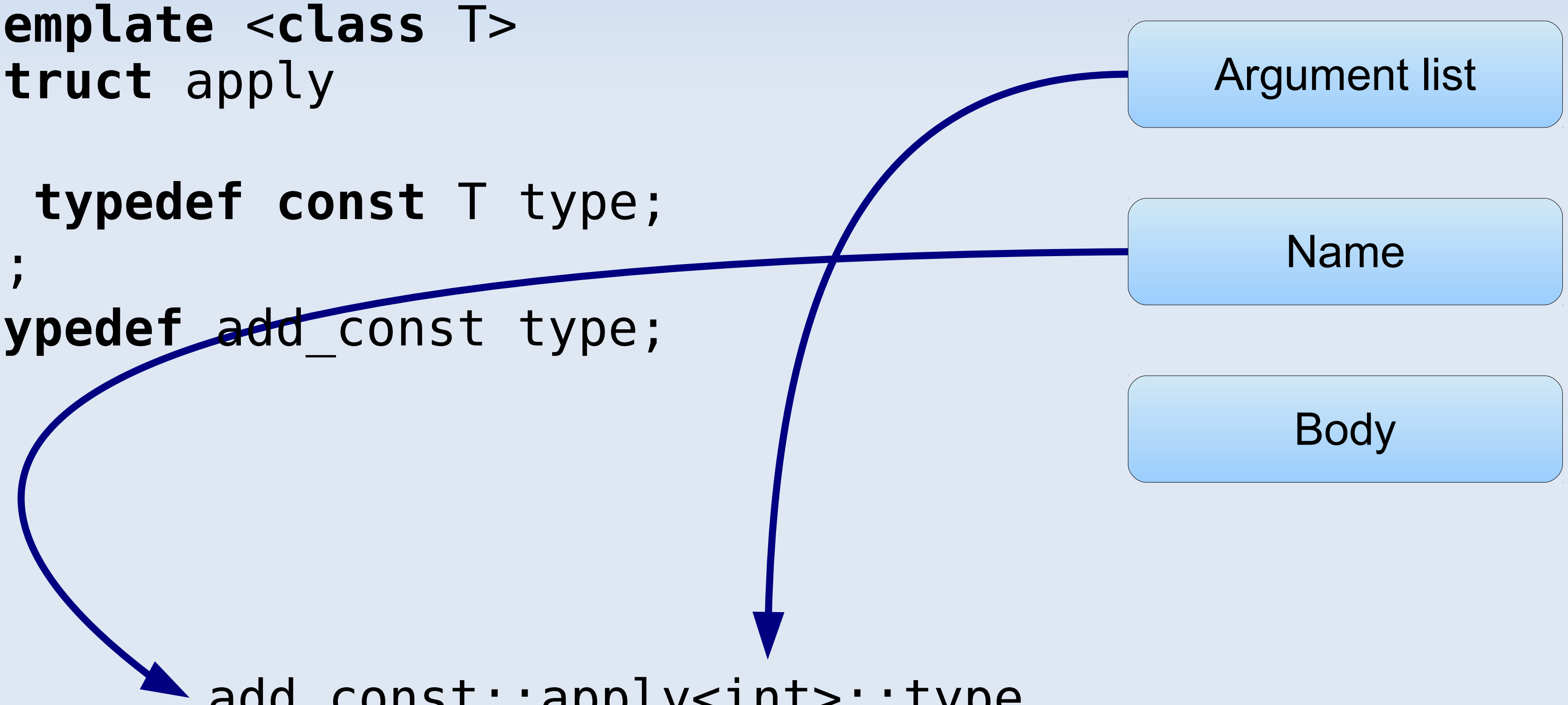
```
struct add_const
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };

};
```



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

Template metafunction class

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



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

Lab 5

- Turn `beginning_and_end` into a template metafunction class

The grammar

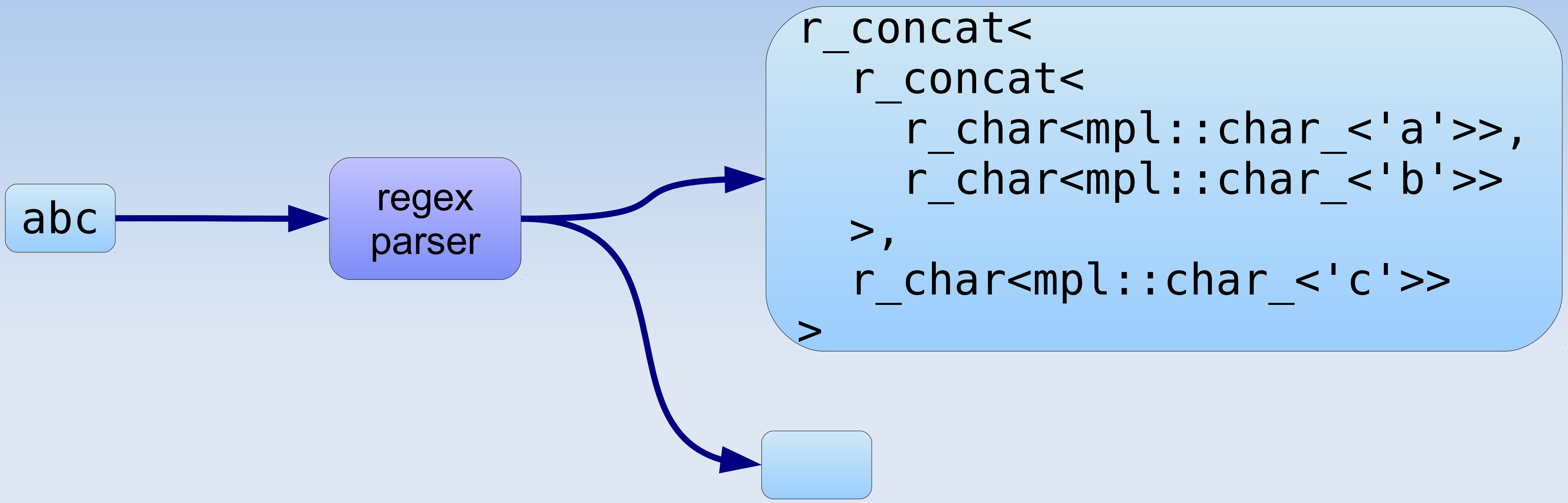
- We will support
  - letters and numbers (eg. **`abc123`**)
  - **`.`**
  - **`*`**
  - brackets (eg. **`(abc)*`**)

The grammar

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

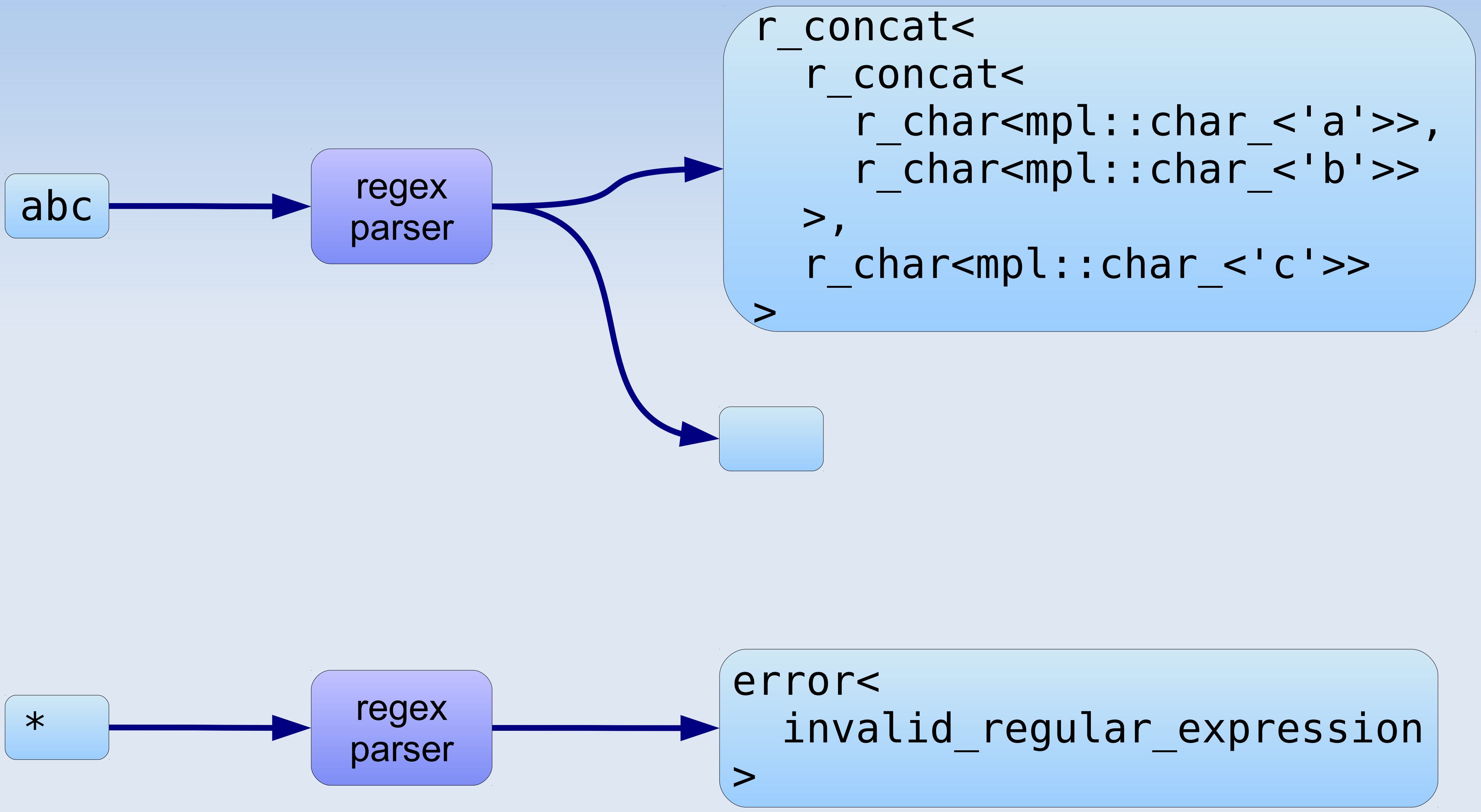
```
reg_exp      ::= unary_item*
unary_item   ::= item '*'?
item         ::= any | bracket_exp | char_
any          ::= '.'
bracket_exp ::= '(' reg_exp ')'
char_        ::= number | letter
number       ::= '0'..'9'
letter       ::= 'a'..'z' | 'A'..'Z'
```

Parsing regular expressions



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

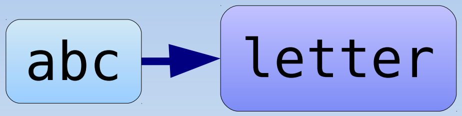
Parsing regular expressions



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

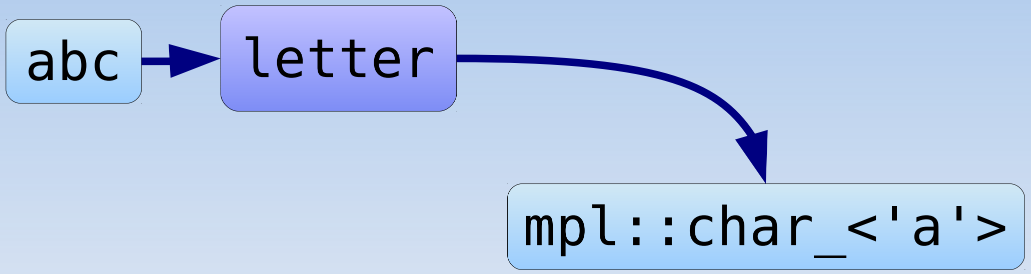
```
error<
  invalid_regular_expression
>
```

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



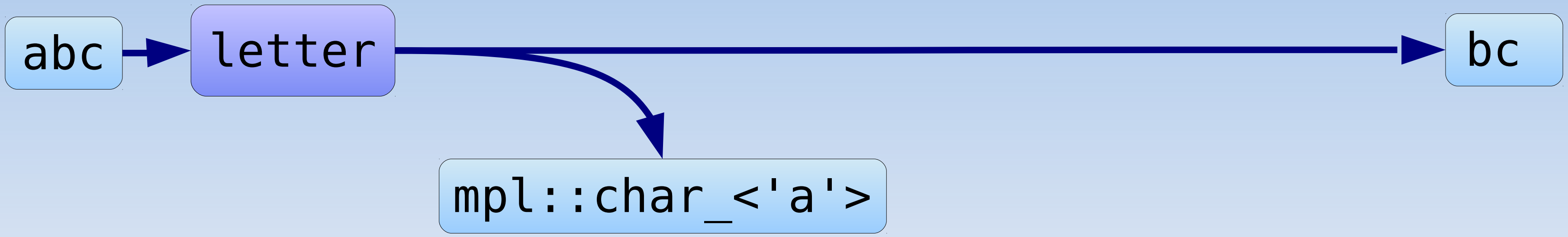
letter

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



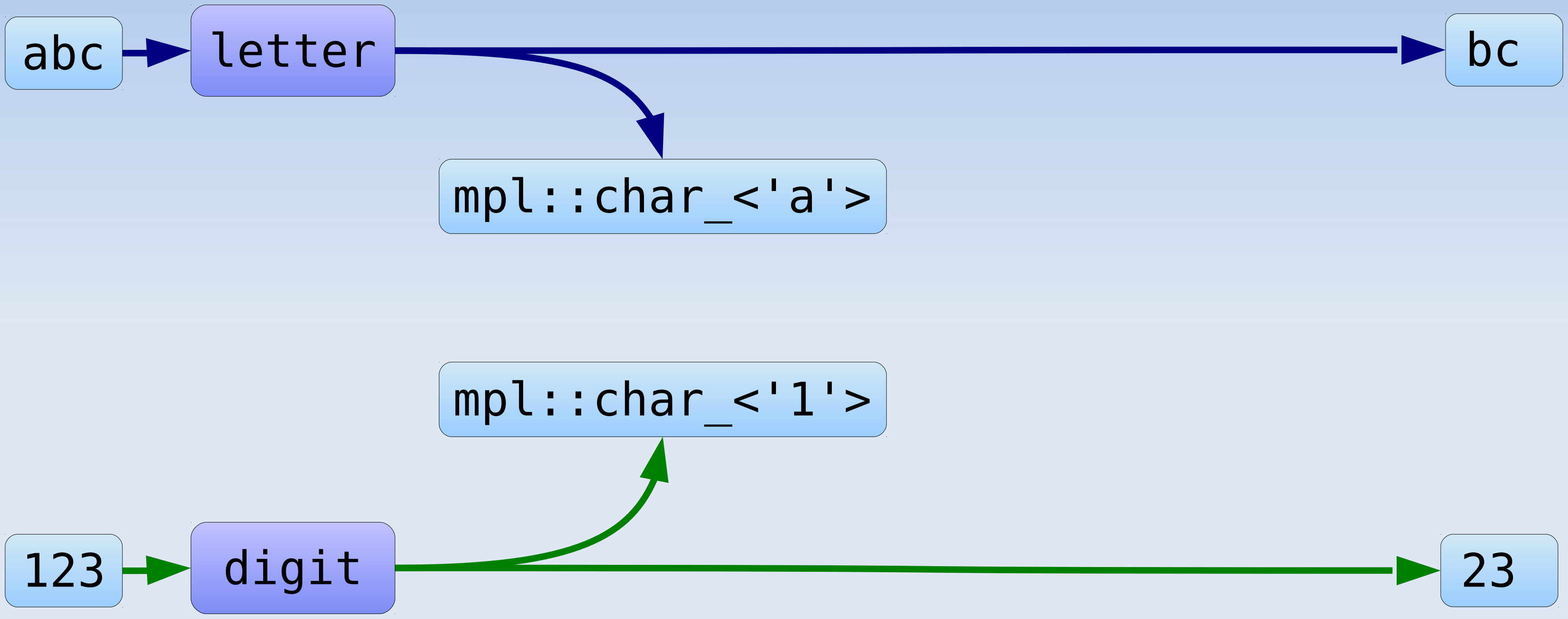
letter

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



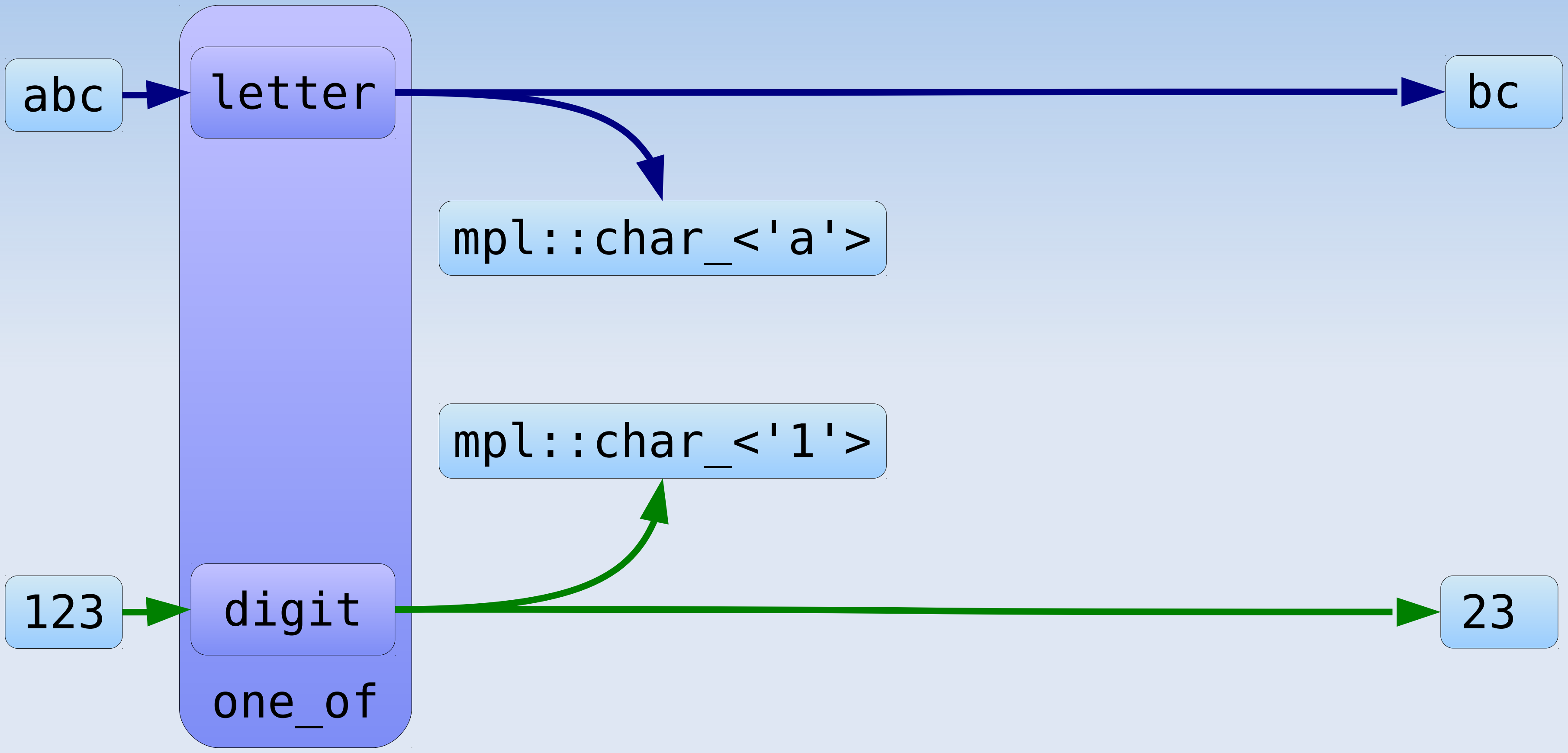
letter

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



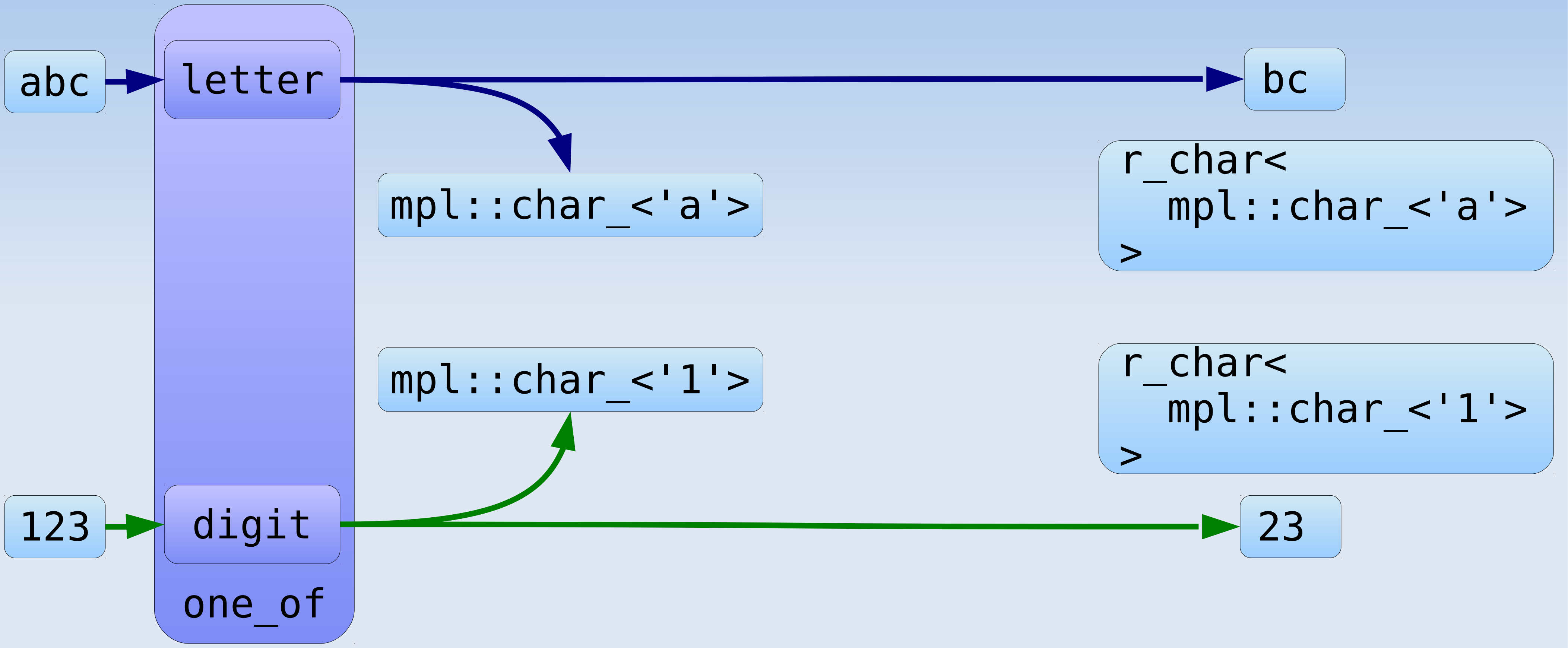
letter digit

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



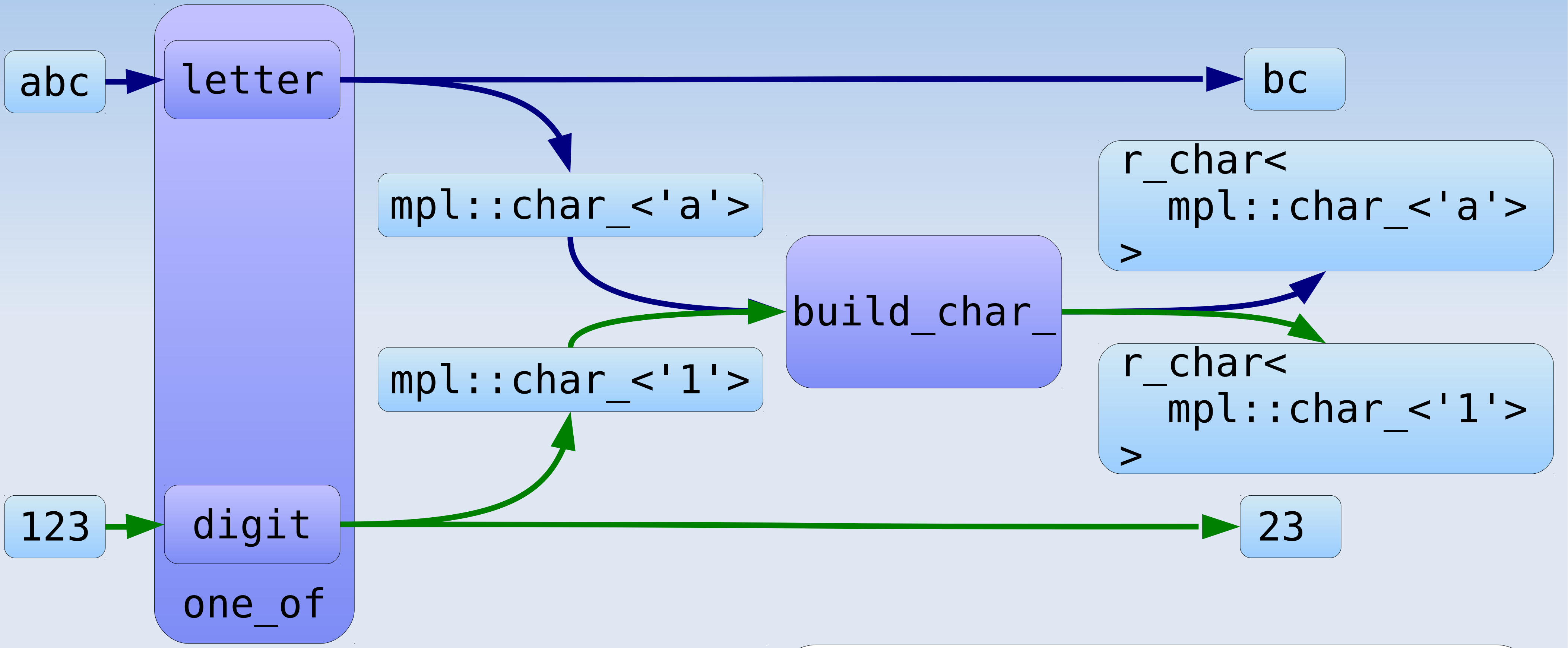
one_of<letter, digit>

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



one_of<letter, digit>

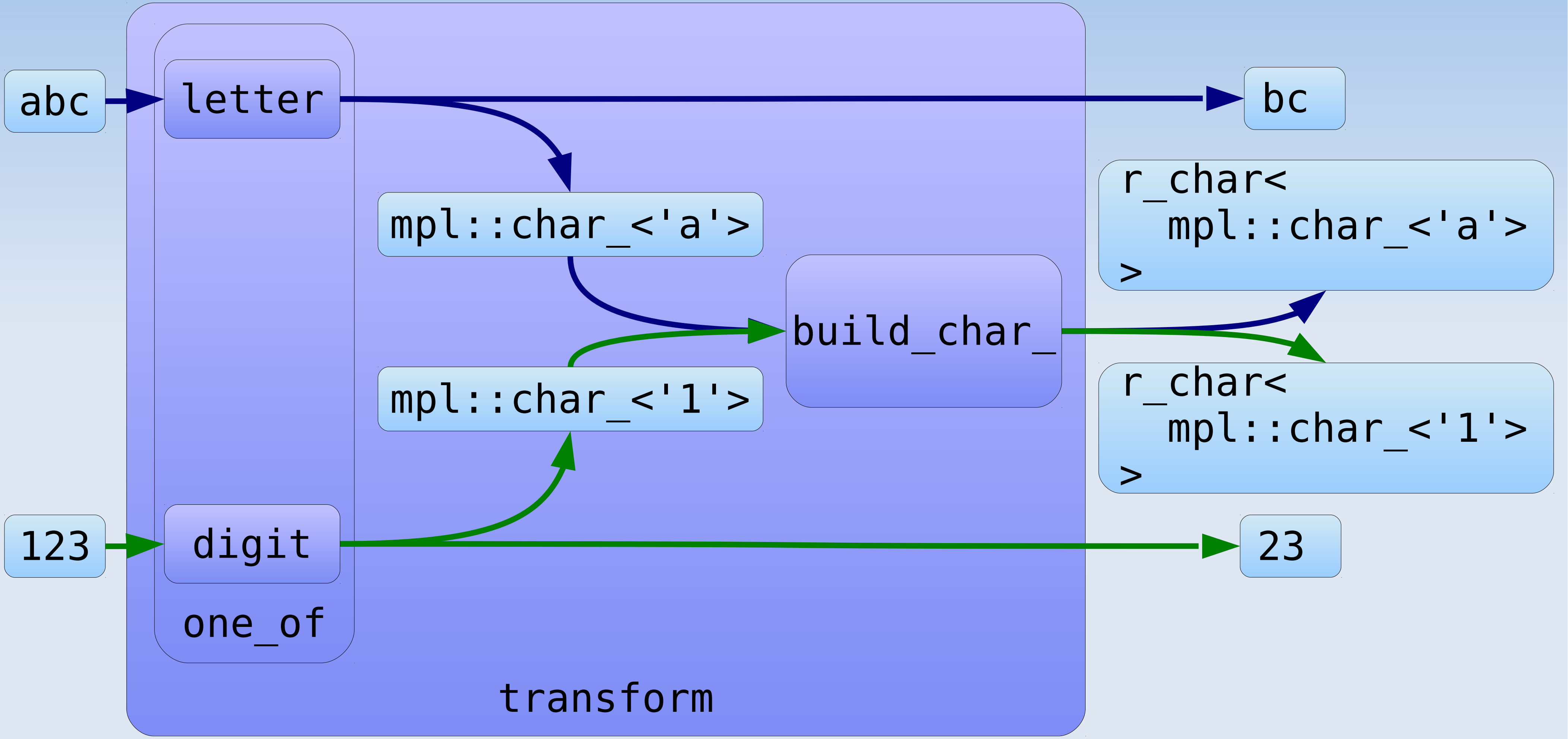
```
char_   ::= number | letter
number ::= '0'..'9'
letter ::= 'a'..'z' | 'A'..'Z'
```



one_of<lette

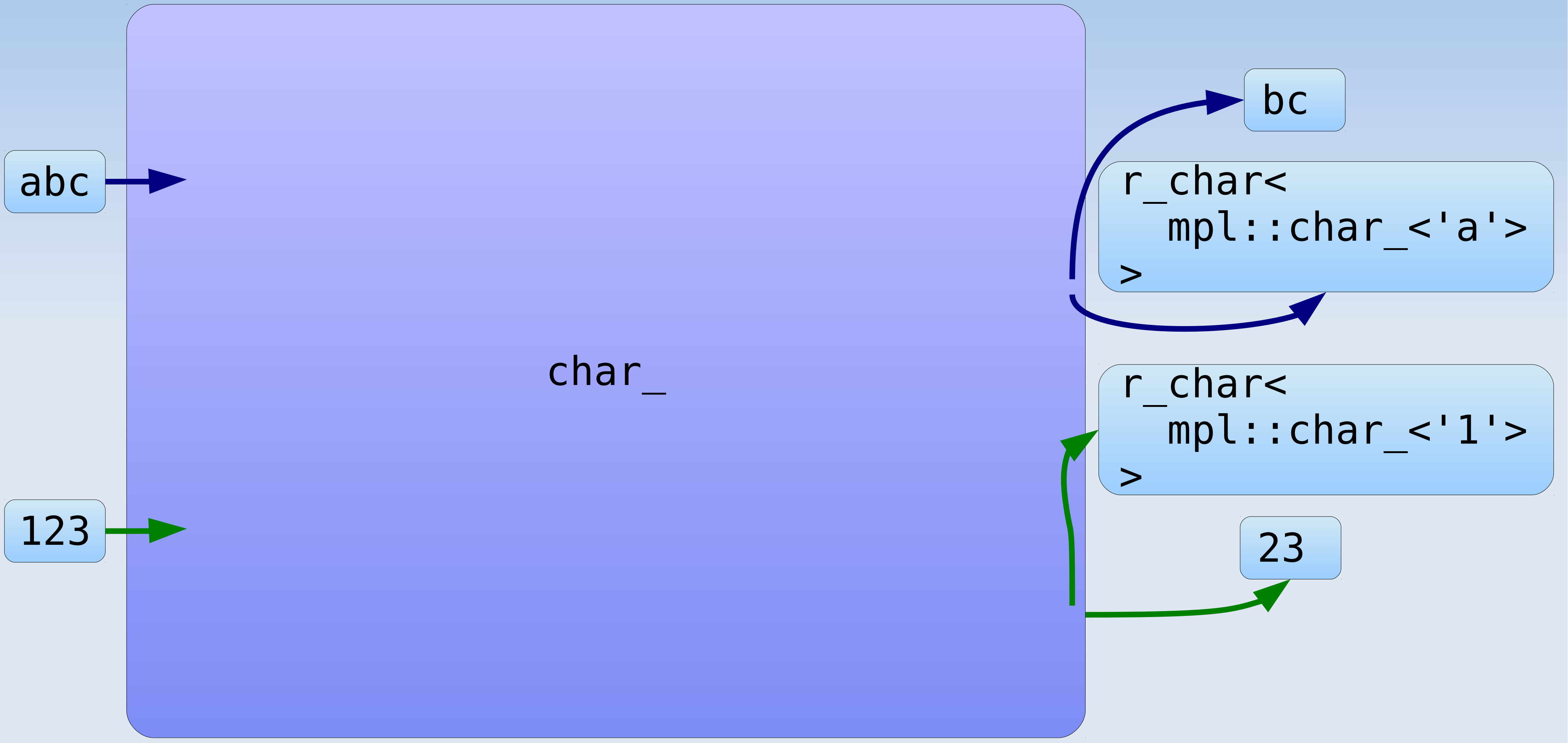
```
struct build_char_ {
  typedef build_char_ type;

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

```
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' ✓
```

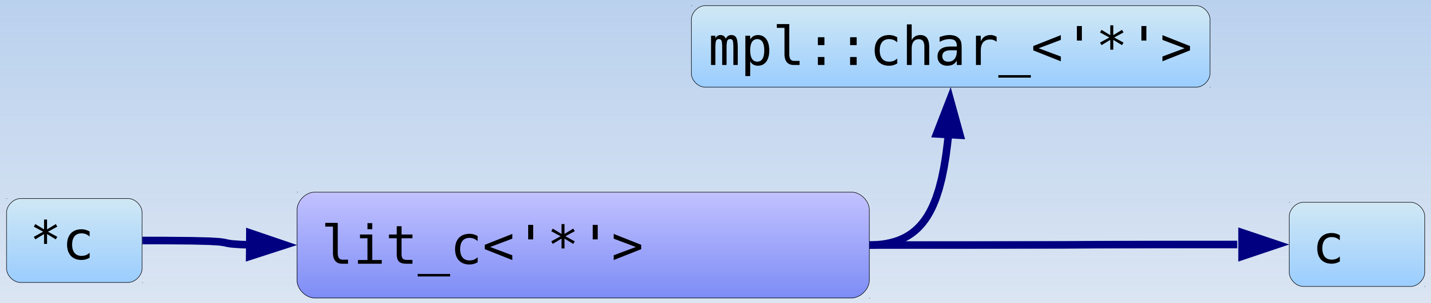
The grammar

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

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

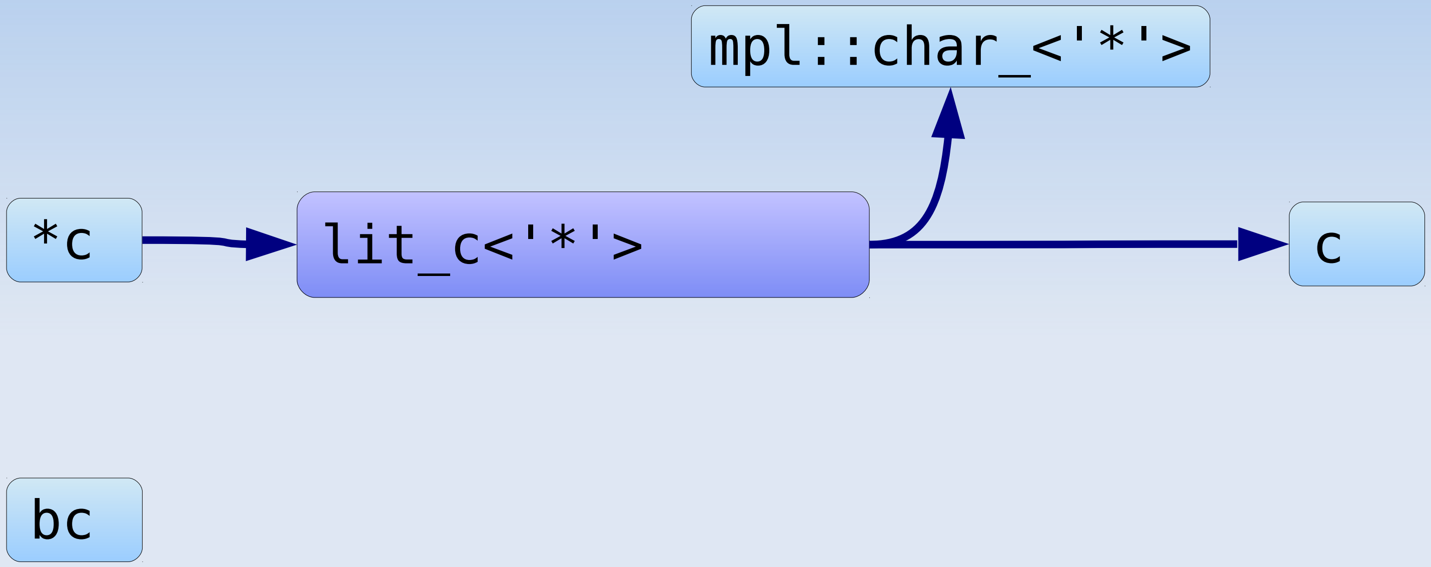
```
reg_exp      ::= unary_item*
unary_item   ::= item '*'?
item         ::= any | bracket_exp | char_
any          ::= '.' ✓
bracket_exp  ::= '(' reg_exp ')'
char_        ::= number | letter ✓
number       ::= '0'..'9' ✓
letter       ::= 'a'..'z' | 'A'..'Z' ✓
```

unary_item ::= item '*'?



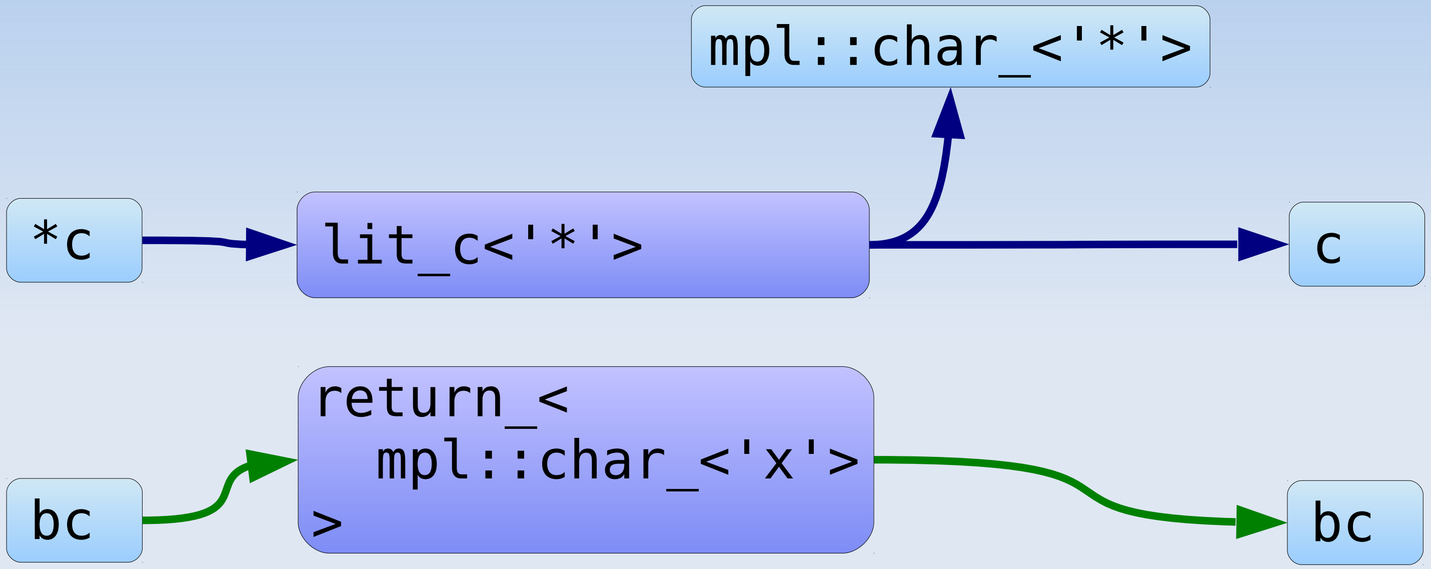
lit_c<'*'>

unary_item ::= item '*'?



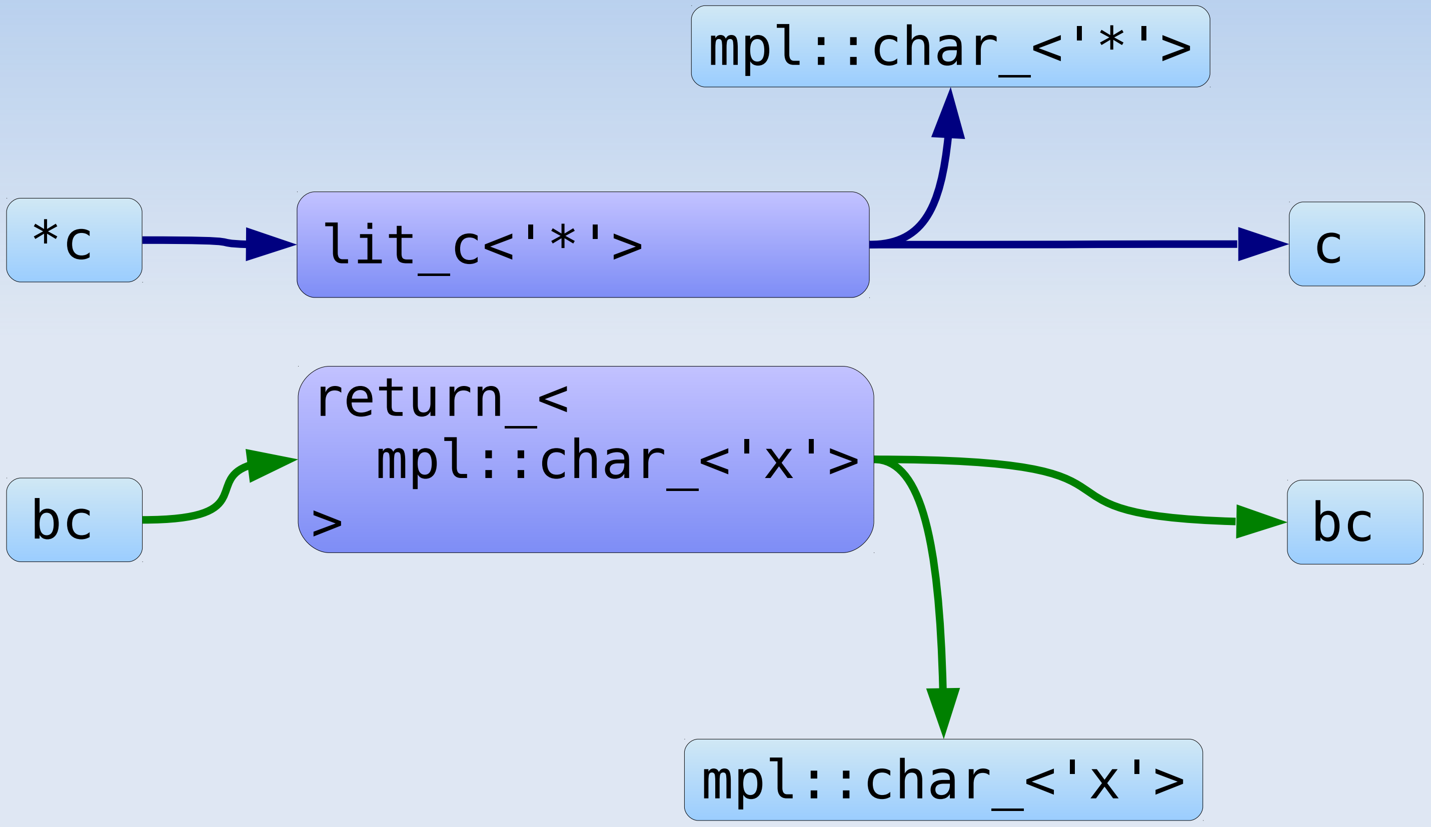
lit_c<'*'>

unary_item ::= item '*'?



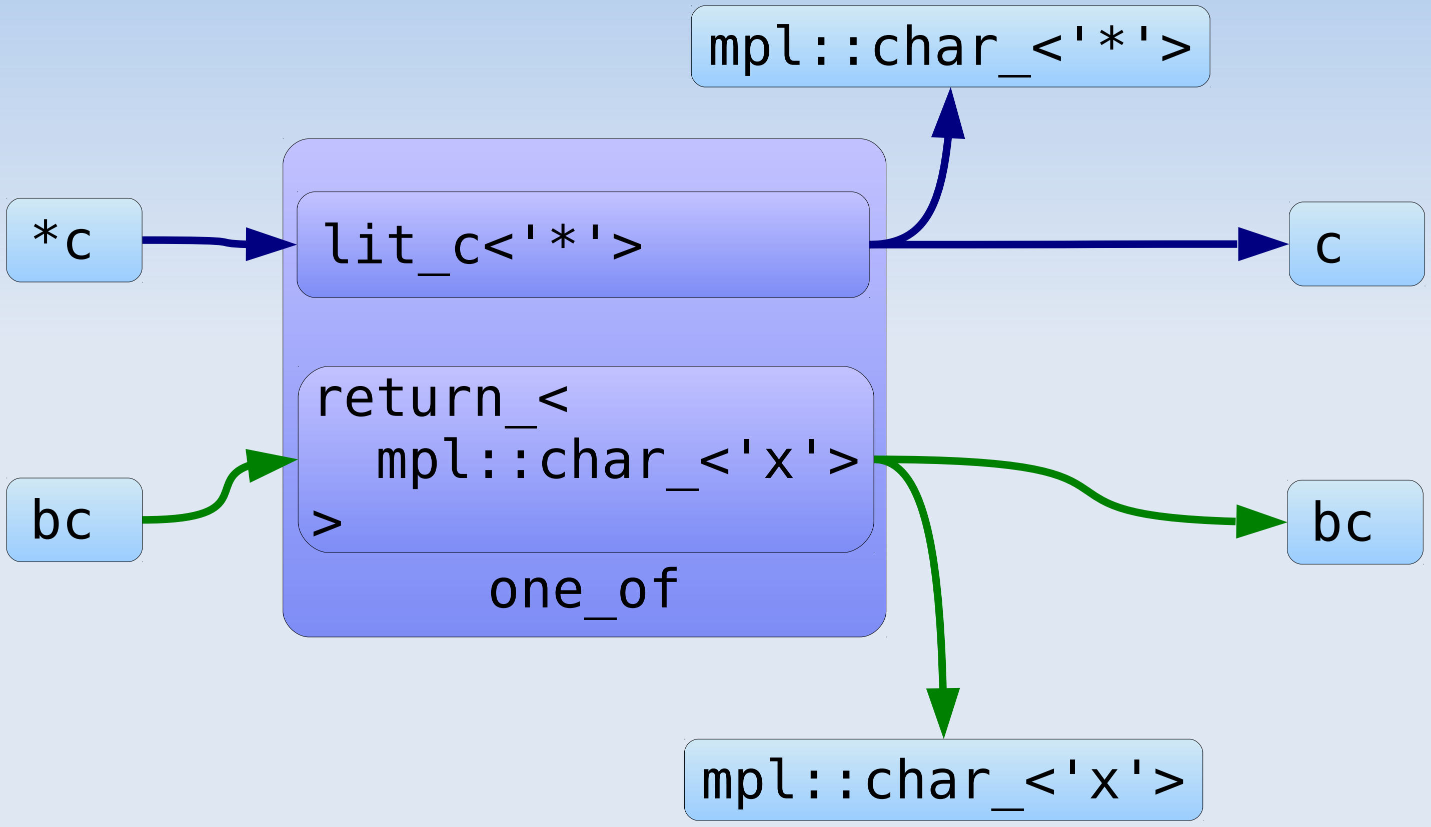
```
lit_c<'*'>  return_<mpl::char_<'x'>>
```

unary_item ::= item '*'?



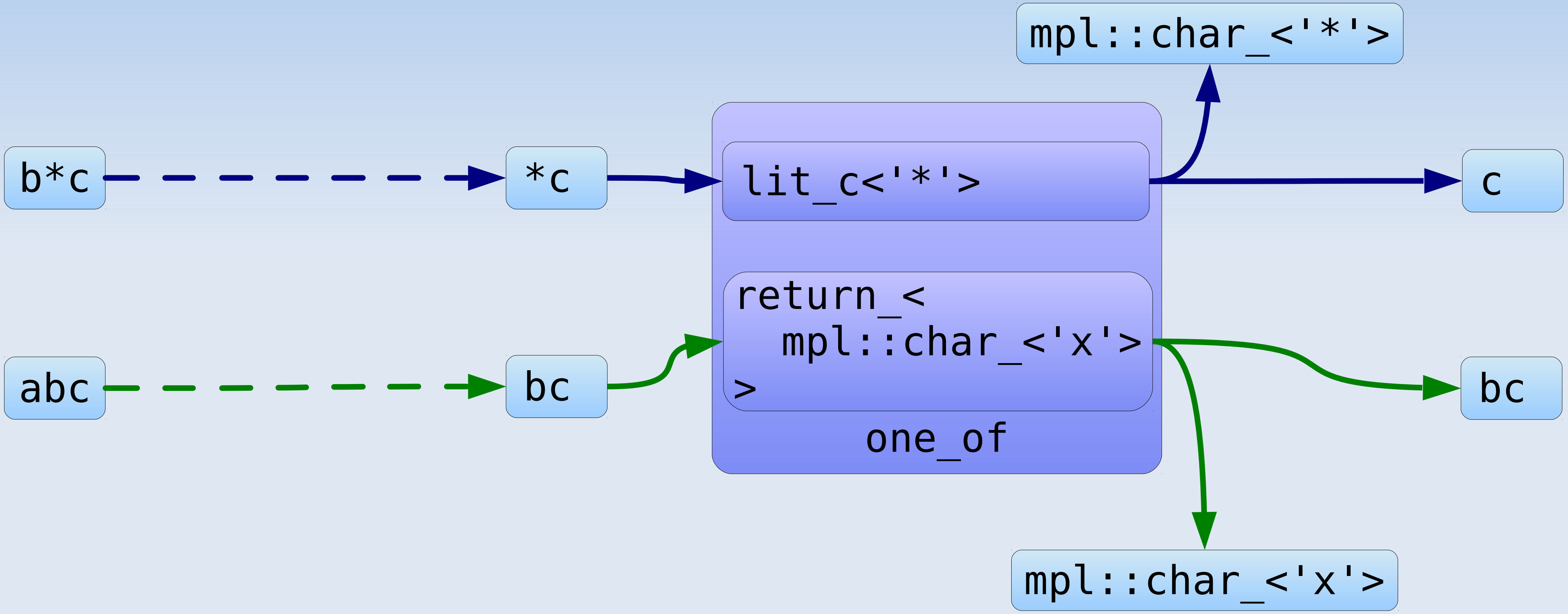
```
lit_c<'*'>   return_<mpl::char_<'x'>>
```

unary_item ::= item '*'?



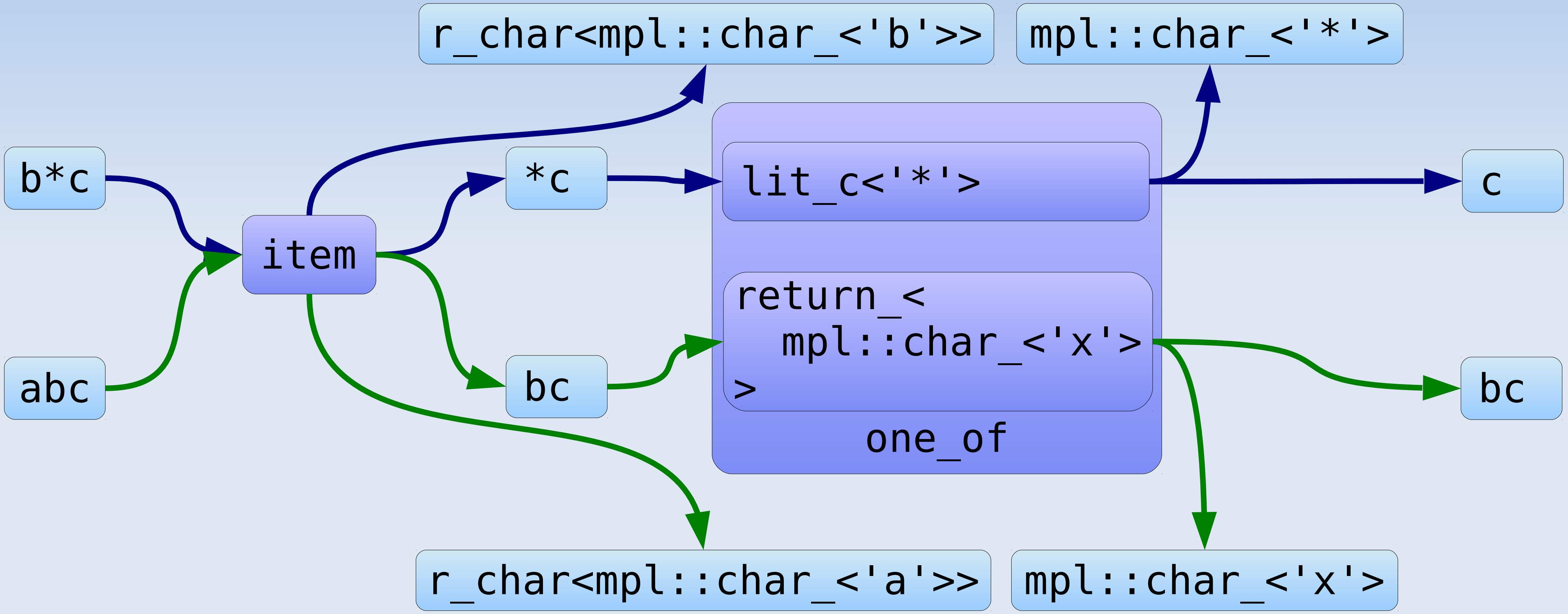
```
one_of<lit_c<'*'>, return_<mpl::char_<'x'>>>
```

unary_item ::= item '*'?



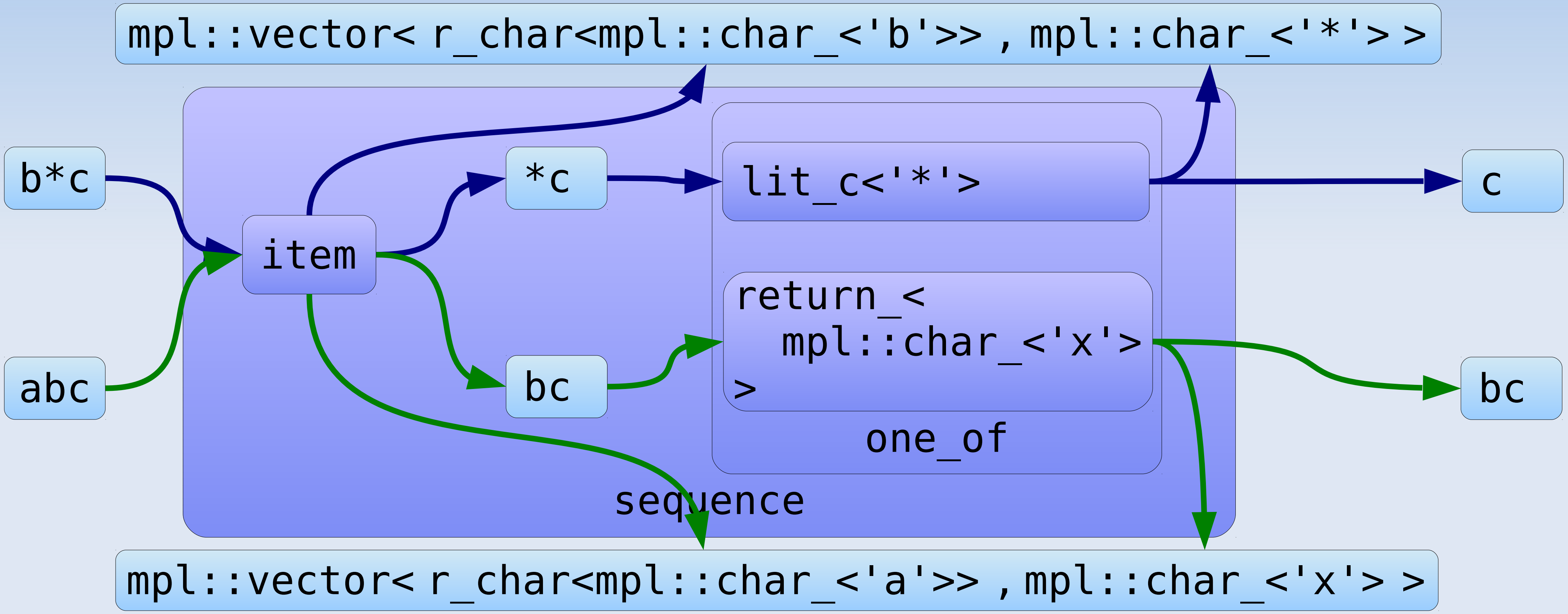
```
one_of<lit_c<'*'>, return_<mpl::char_<'x'>>>
```

unary_item ::= item '*'?



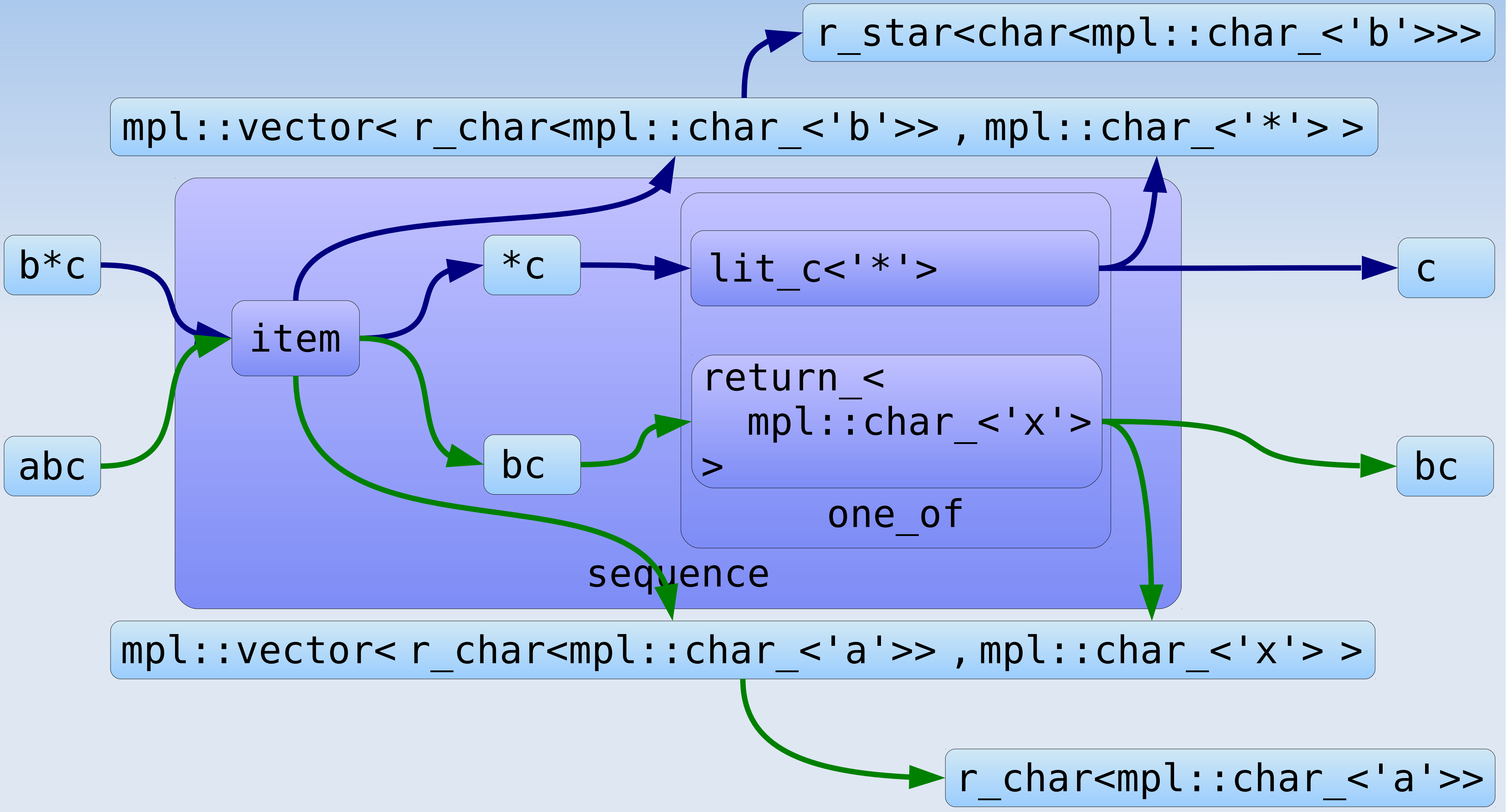
```
item  one_of<lit_c<'*'>, return_<mpl::char_<'x'>>>
```

unary_item ::= item '*'?


mpl::vector< r_char<mpl::char_<'b'>> , mpl::char_<'*'> >
b*c
abc
item
*c
bc
lit_c<'*'>
return_< mpl::char_<'x'> >
one_of
sequence
c
bc
mpl::vector< r_char<mpl::char_<'a'>> , mpl::char_<'x'> >

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

unary_item ::= item '*'?


r_star<char<mpl::char_<'b'>>>
mpl::vector< r_char<mpl::char_<'b'>> , mpl::char_<'*'> >
b*c
item
*c
lit_c<'*'>
c
abc
bc
return_<
mpl::char_<'x'>
>
bc
one_of
sequence
mpl::vector< r_char<mpl::char_<'a'>> , mpl::char_<'x'> >
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

c

abc

bc

```
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> {};
```

b*c

abc

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

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

unary_item ::= item '*'?

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

b*c c

abc bc

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

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

template <class RegExp>
struct impl<RegExp, 'x'> : 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'>>>

b*

c

ab

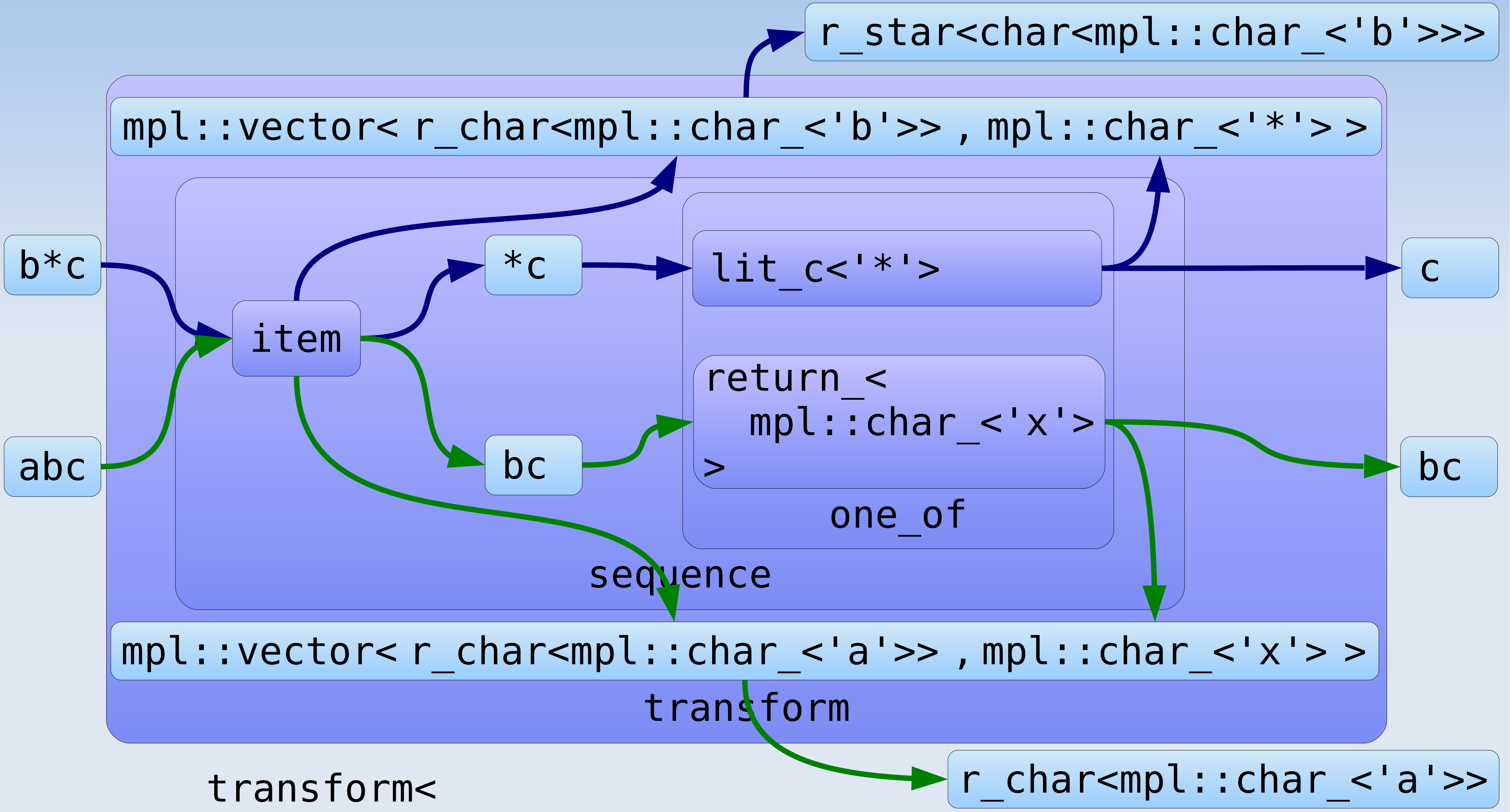
bc

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

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

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'>>

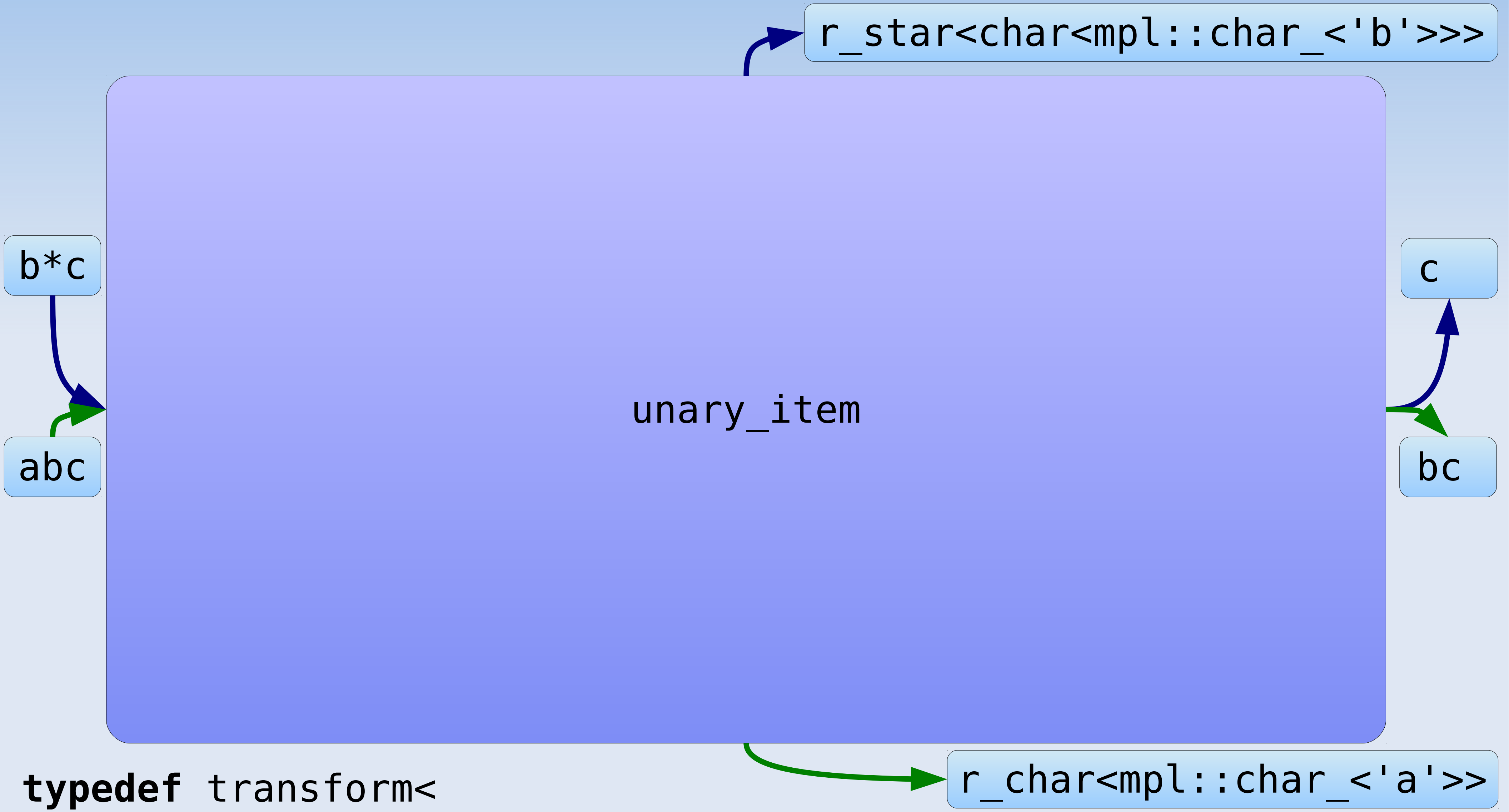
```
sequence<item, one_of<lit_c<'*'>, return_<mpl::char_<'x'>>>>
```

unary_item ::= item '*'?


r_star<char<mpl::char_<'b'>>>
mpl::vector< r_char<mpl::char_<'b'>> , mpl::char_<'*'> >
b*c
item
*c
lit_c<'*'>
c
abc
bc
return_<
mpl::char_<'x'>
>
bc
one_of
sequence
mpl::vector< r_char<mpl::char_<'a'>> , mpl::char_<'x'> >
transform
r_char<mpl::char_<'a'>>

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

unary_item ::= item '*'?



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

The grammar

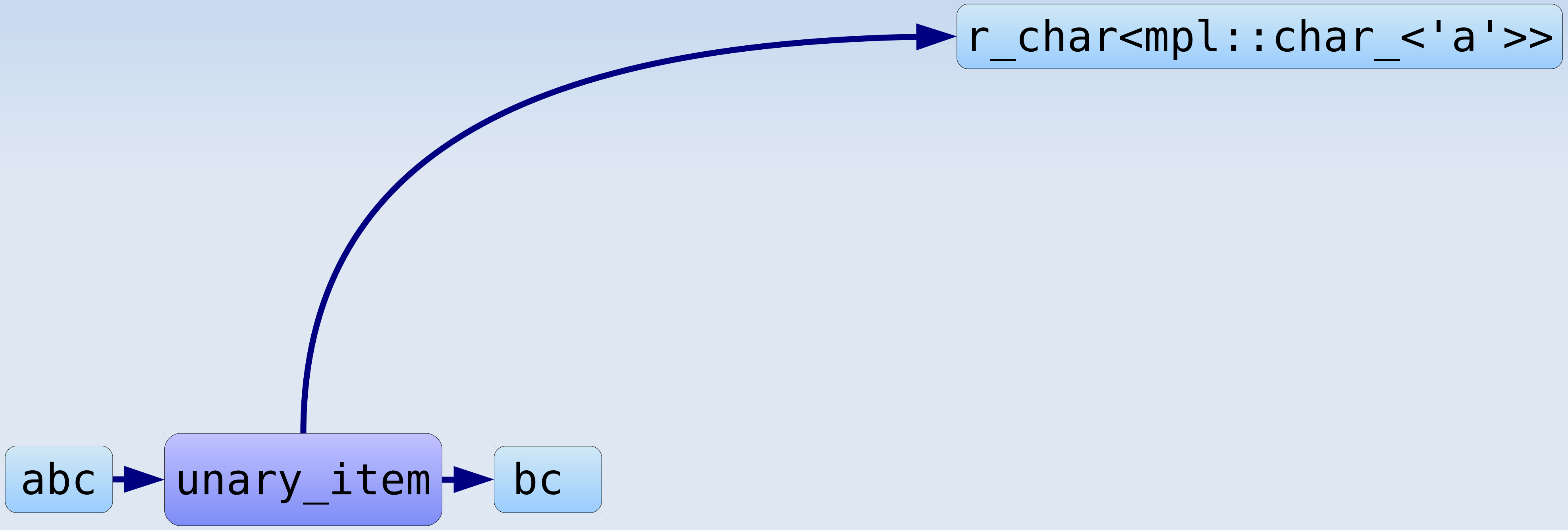
- We will support
  - letters and numbers (eg. **`abc123`**)
  - **`.`**
  - **`*`**
  - brackets (eg. **`(abc)*`**)

```
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*

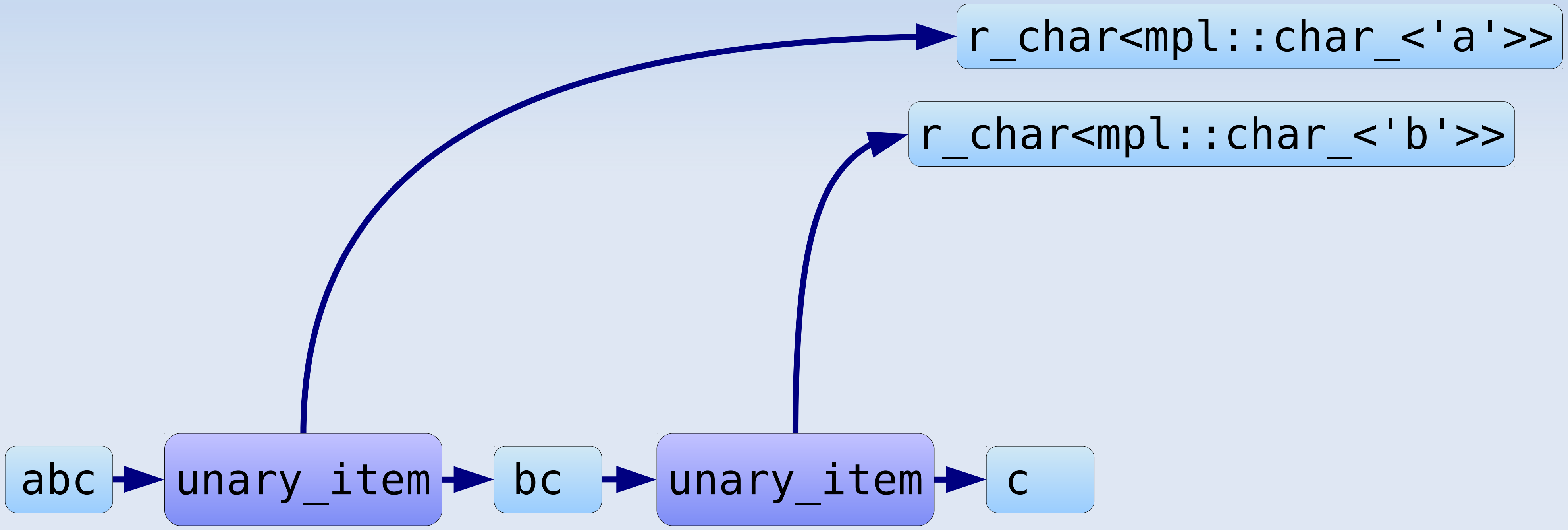
abc

reg_exp ::= unary_item*



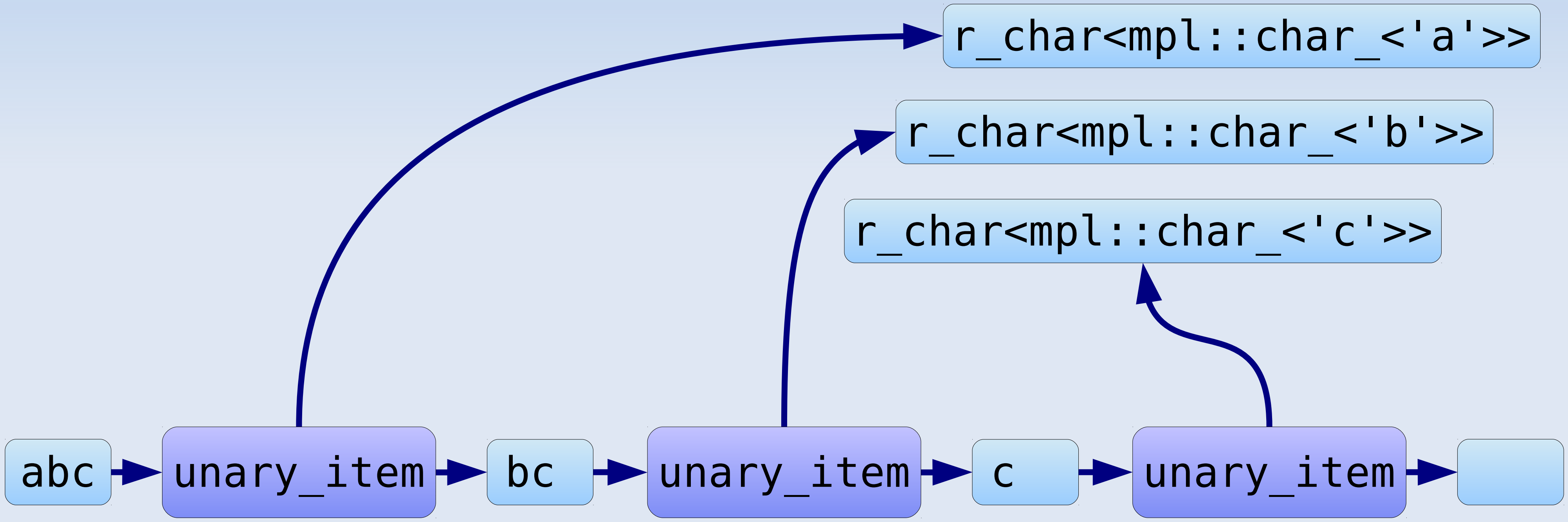
unary_item

reg_exp ::= unary_item*



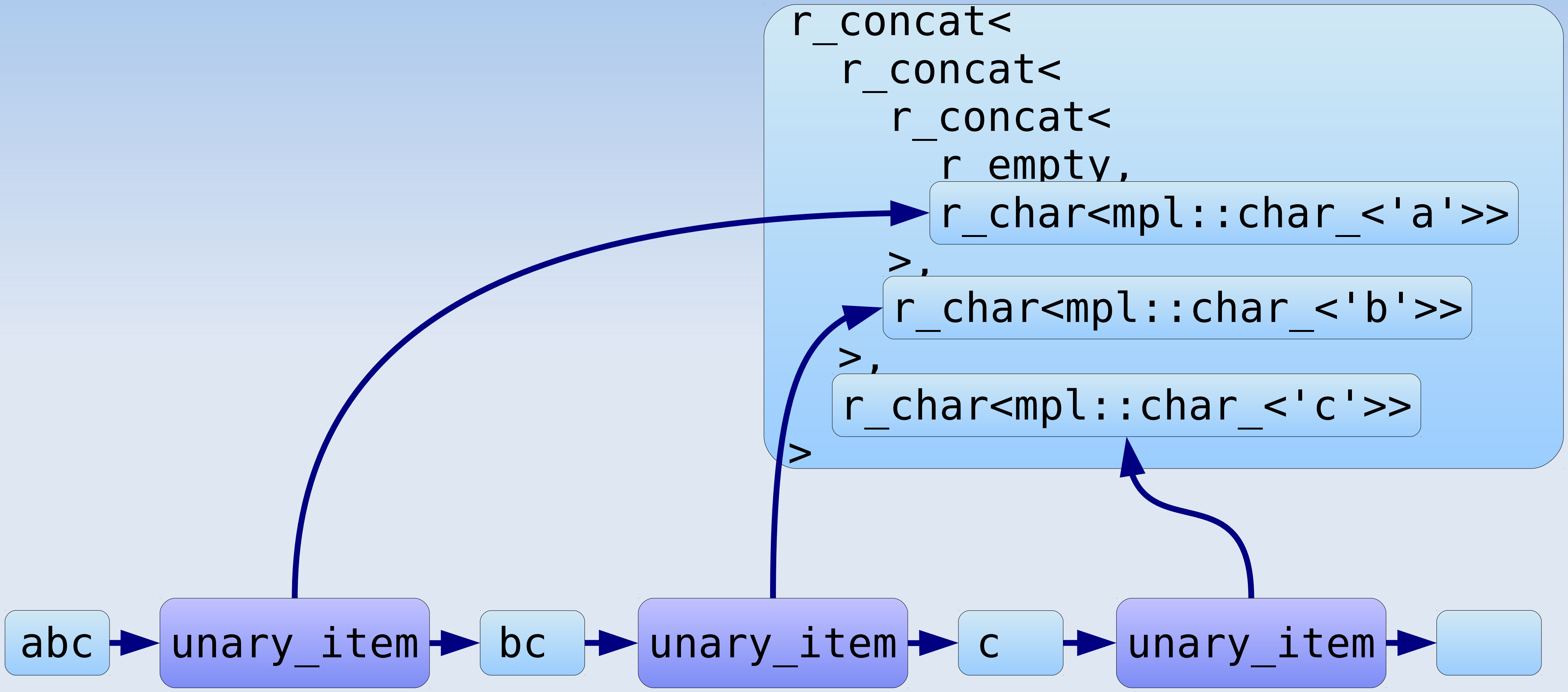
unary_item

reg_exp ::= unary_item*



unary_item

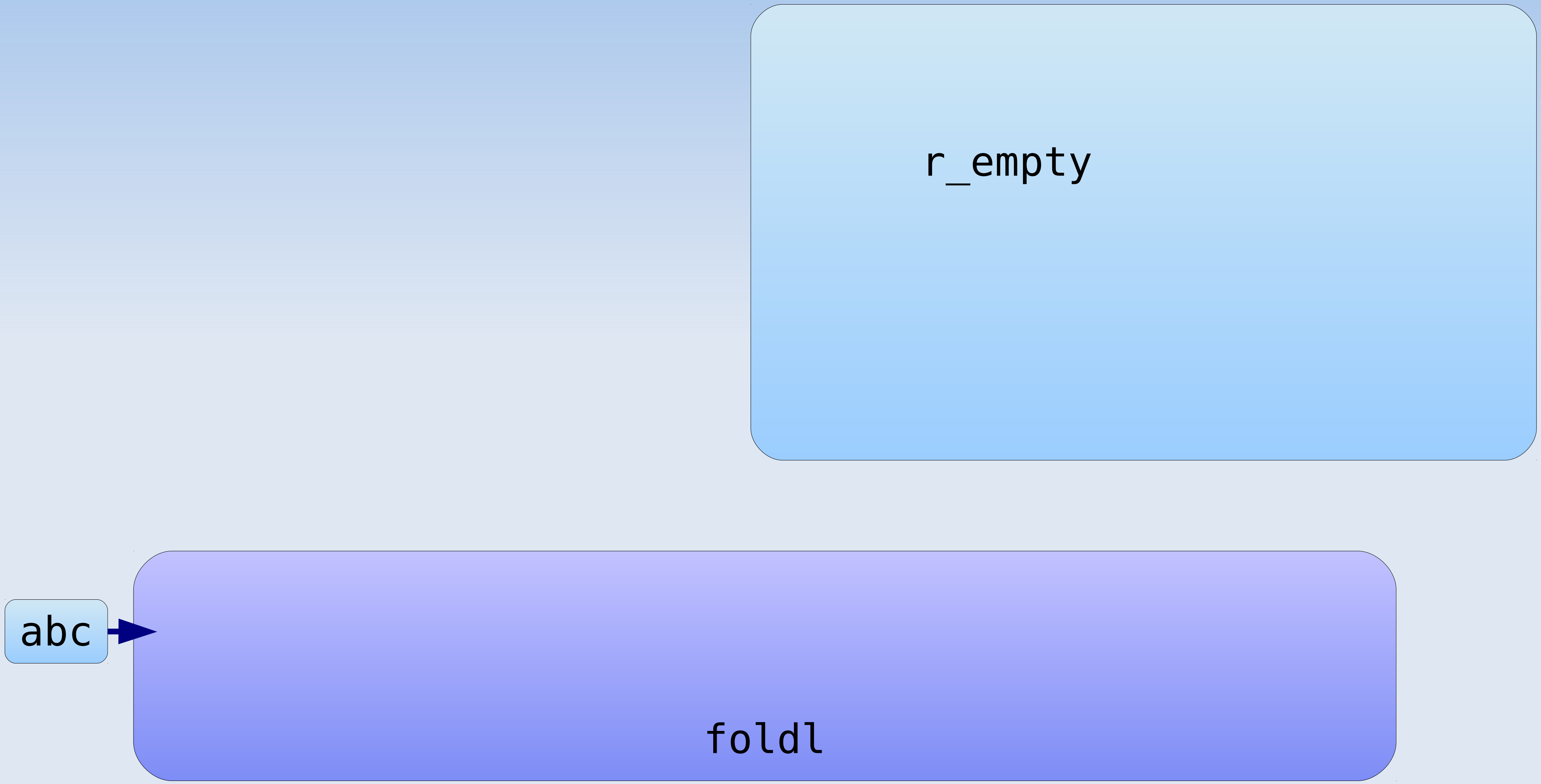
reg_exp ::= unary_item*



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

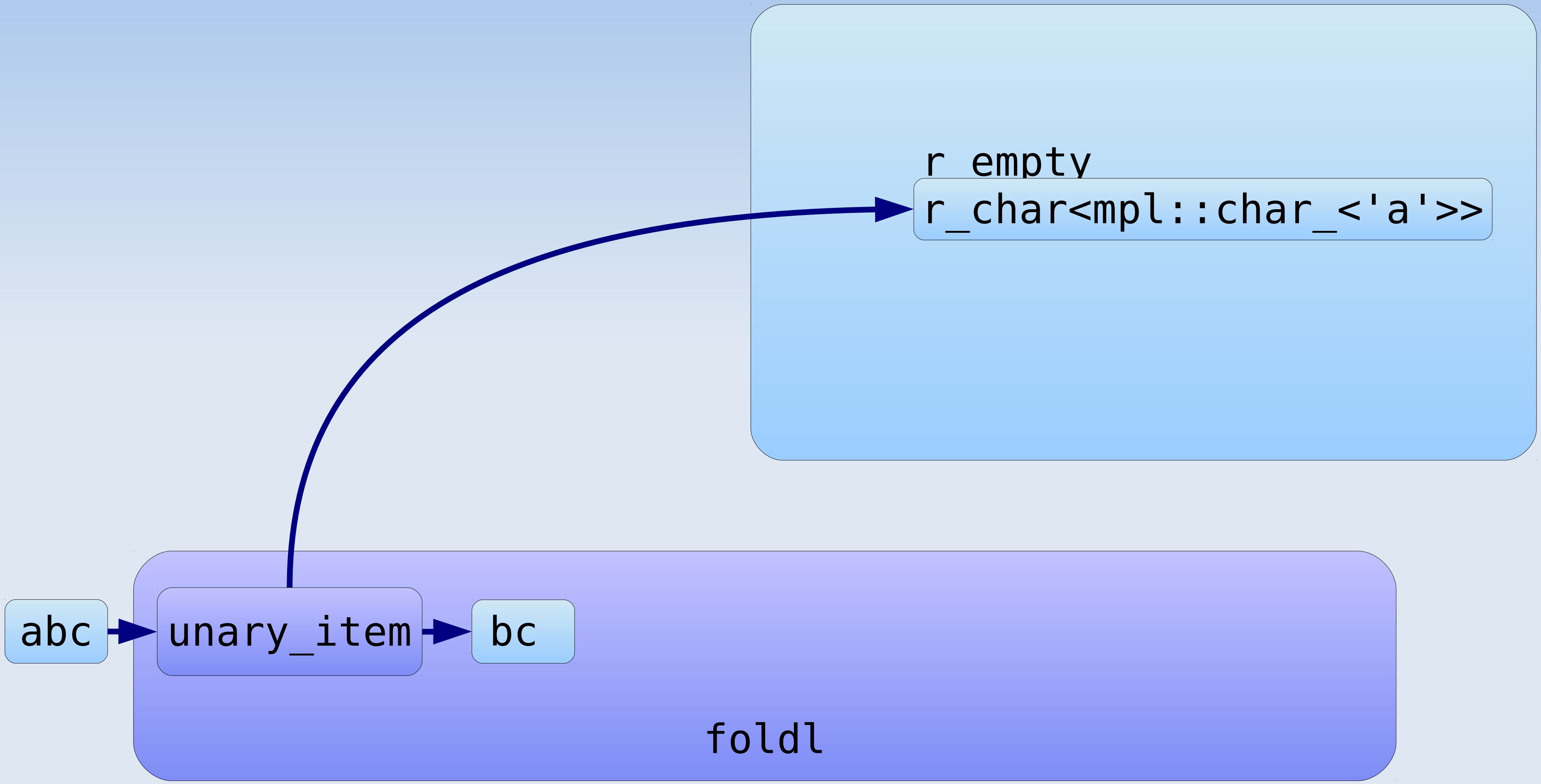
unary_item

reg_exp ::= unary_item*



foldl<unary_item, r_empty, build_reg_exp>

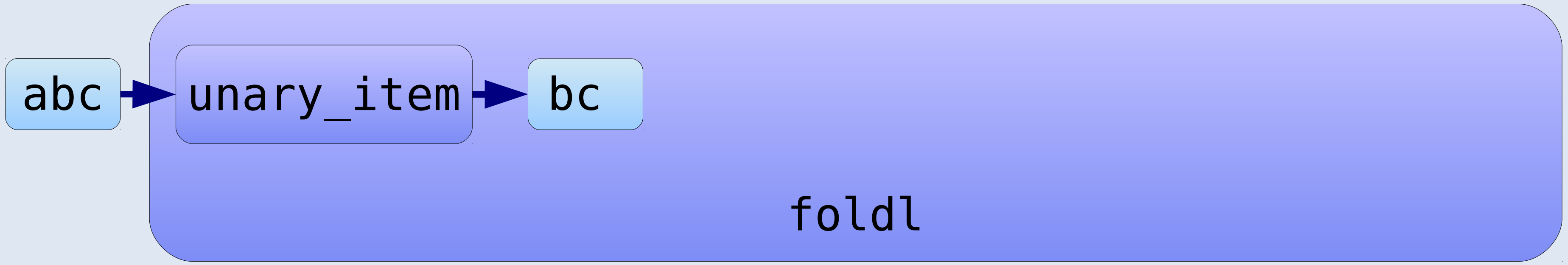
reg_exp ::= unary_item*



foldl<unary_item, r_empty, build_reg_exp>

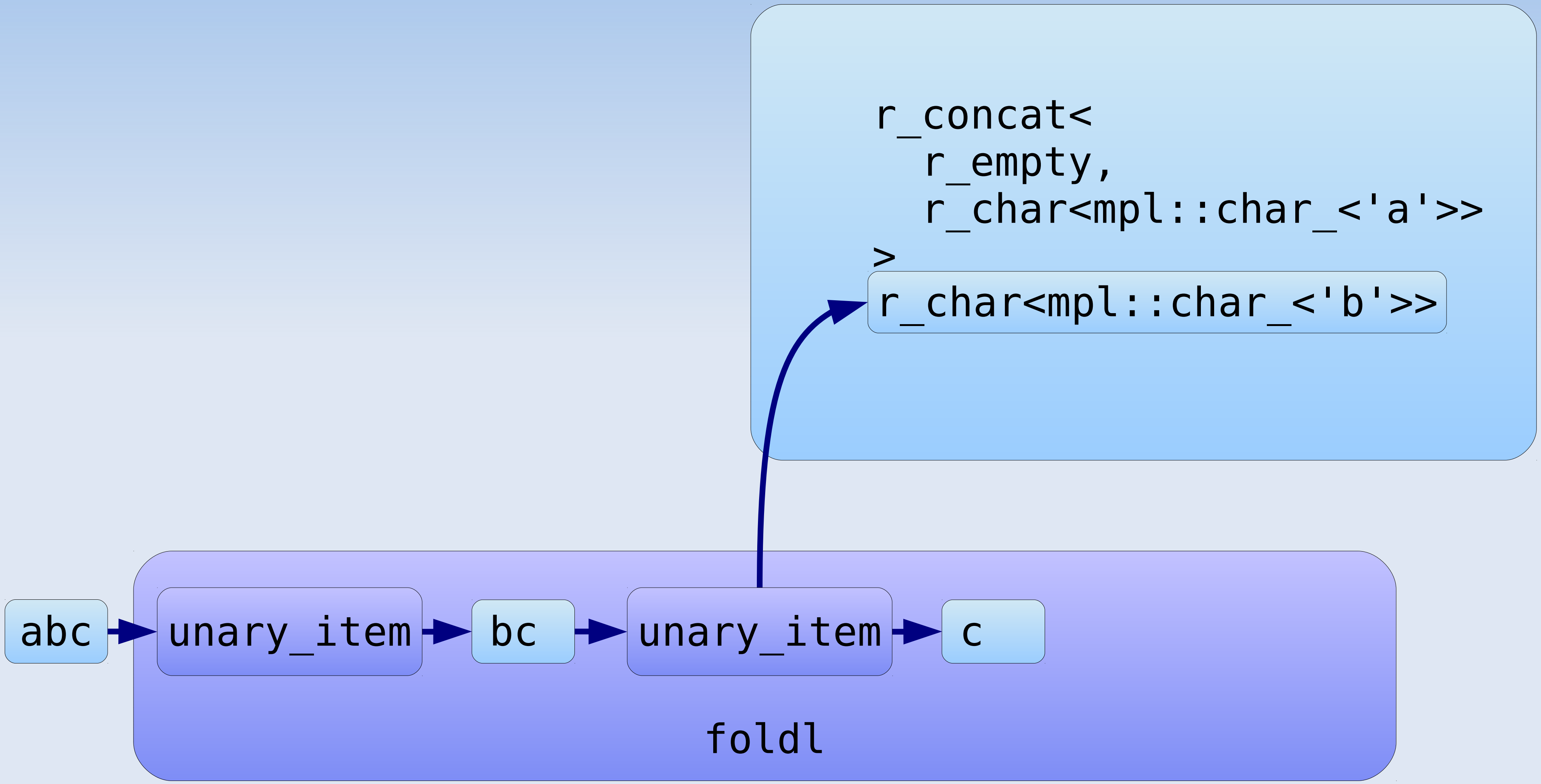
reg_exp ::= unary_item*

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



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

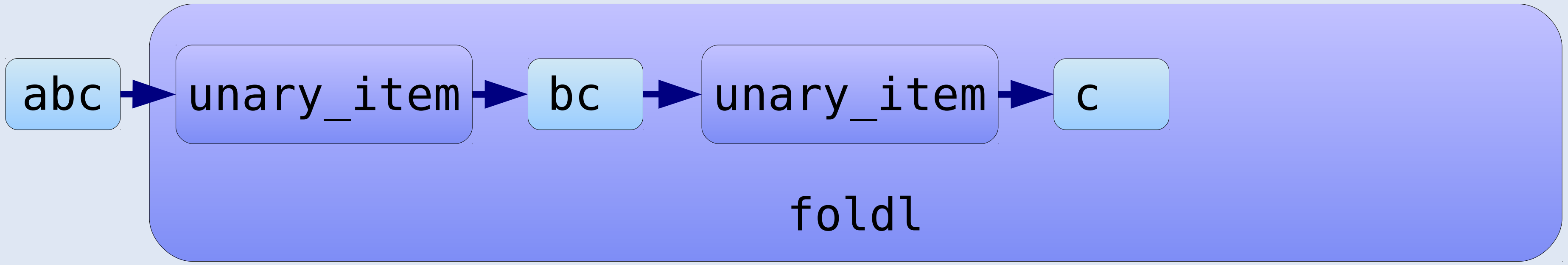
reg_exp ::= unary_item*



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

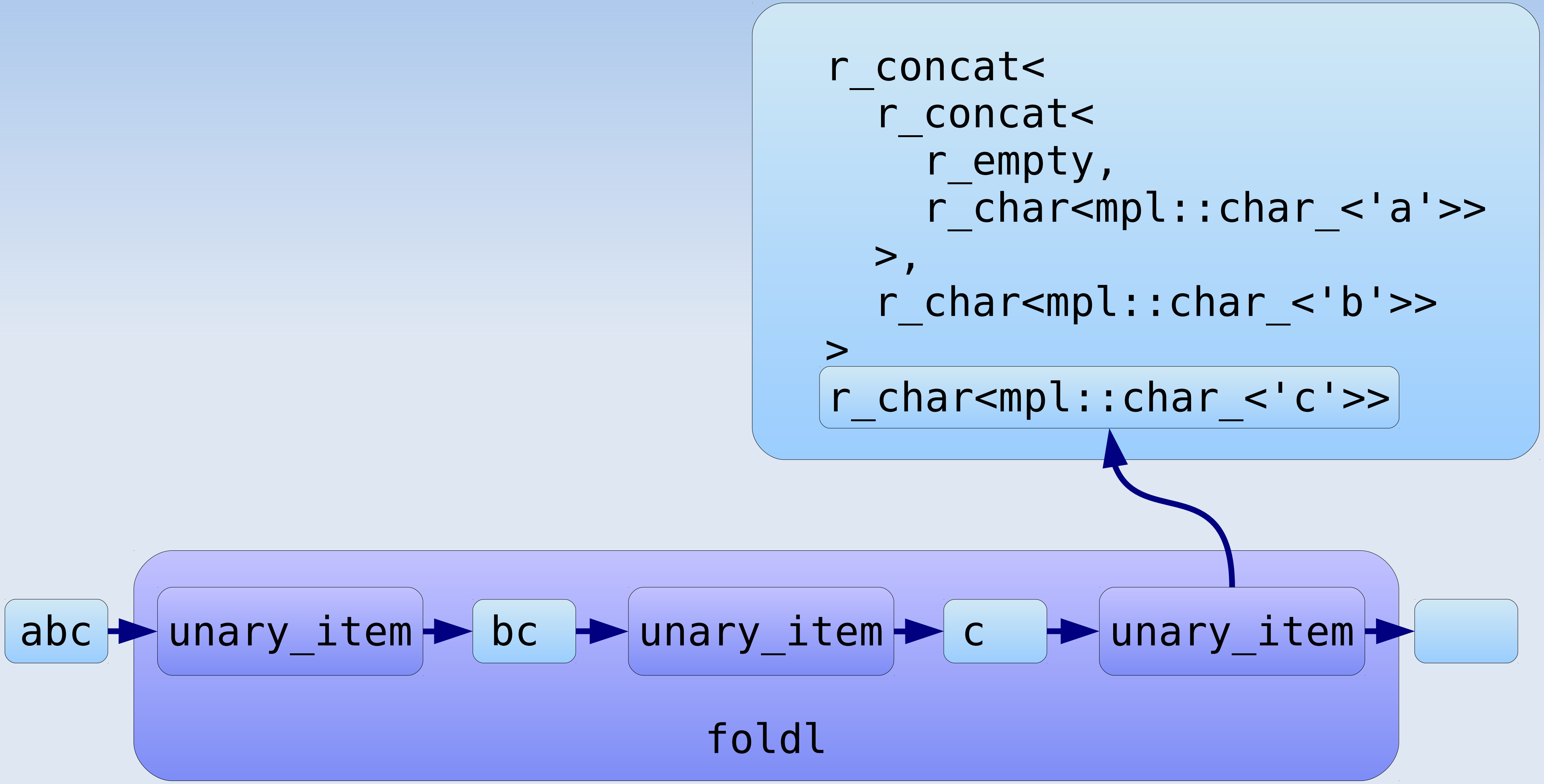
reg_exp ::= unary_item*

```
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*

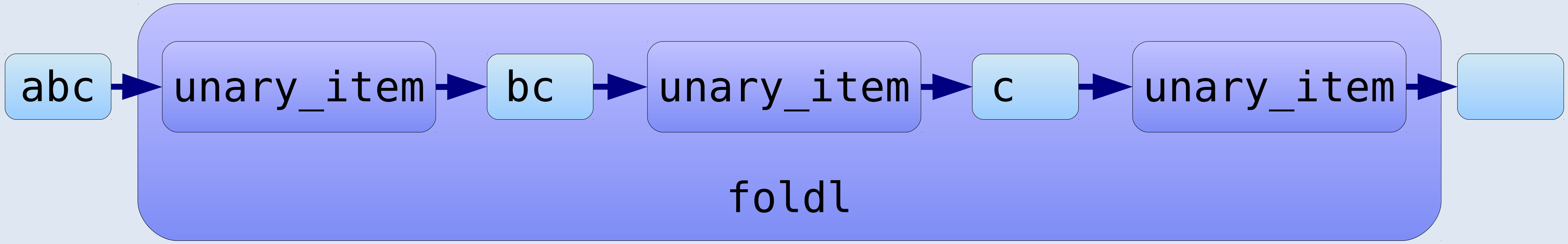


```
r_concat<
  r_concat<
    r_empty,
    r_char<mpl::char_<'a'>>
  >,
  r_char<mpl::char_<'b'>>
>
r_char<mpl::char_<'c'>>
```

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

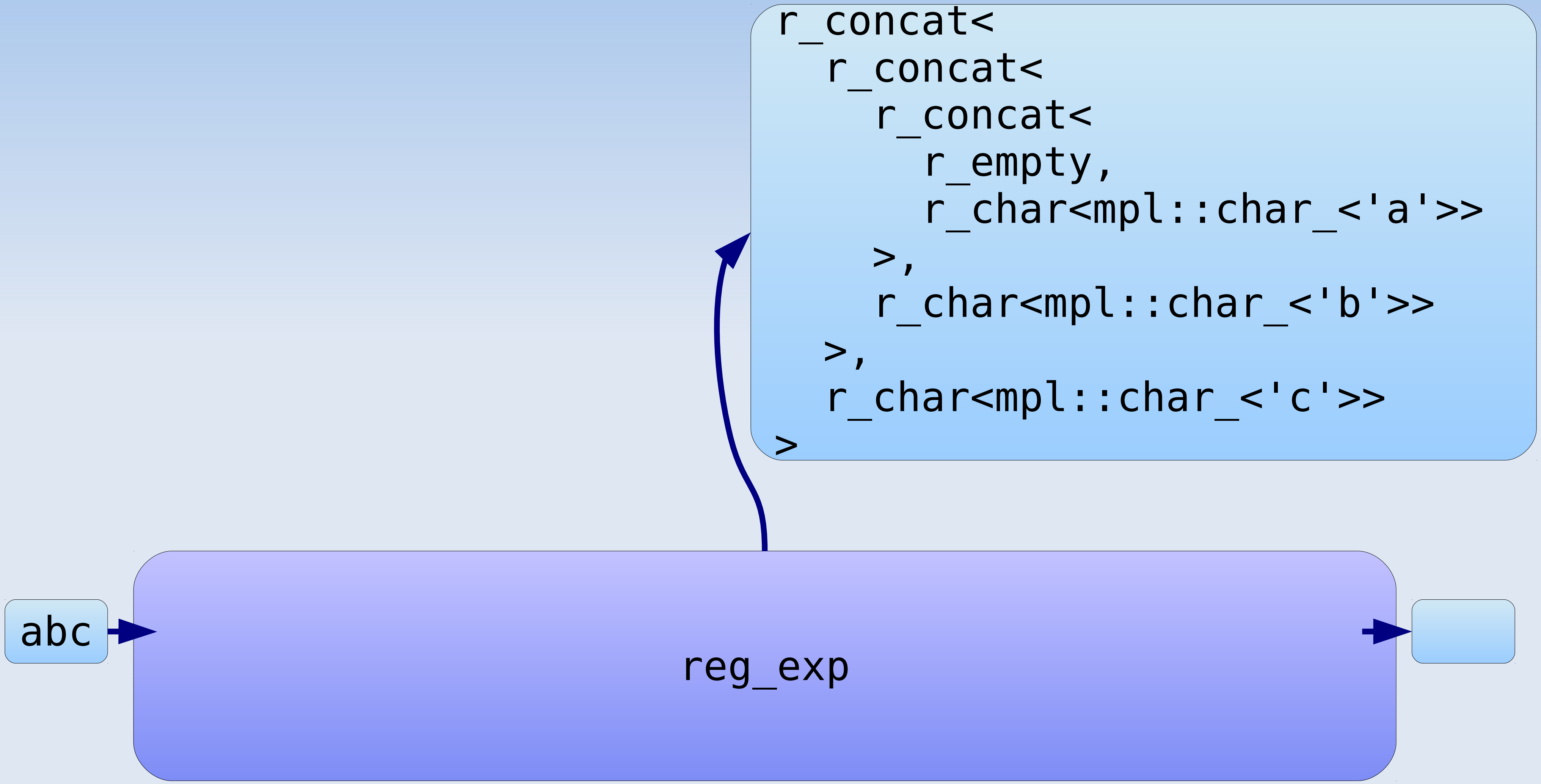
reg_exp ::= unary_item*

```
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*



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

```
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)***)

```
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

- 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'✓
```

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 ')'

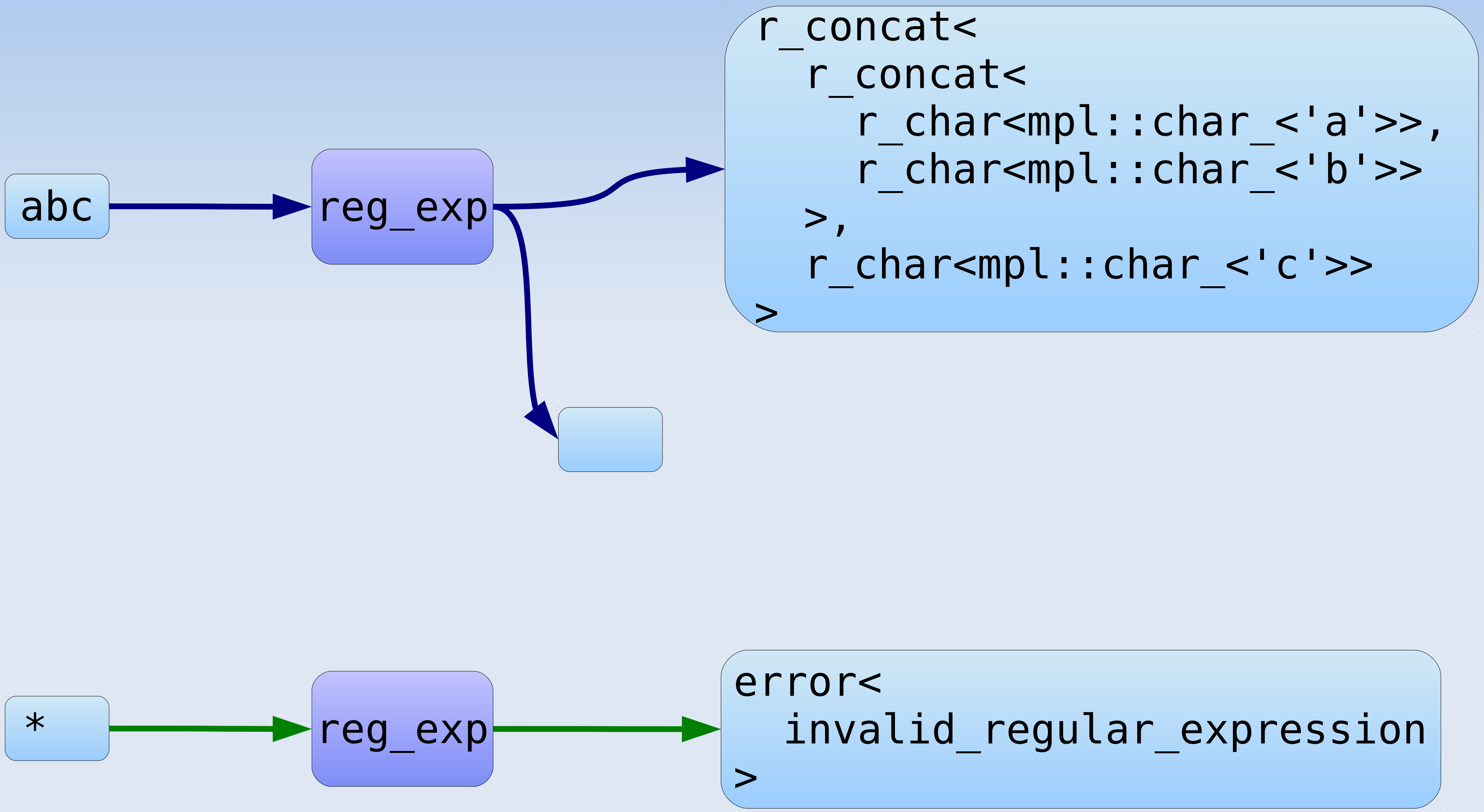
```
struct reg_exp;

// …

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

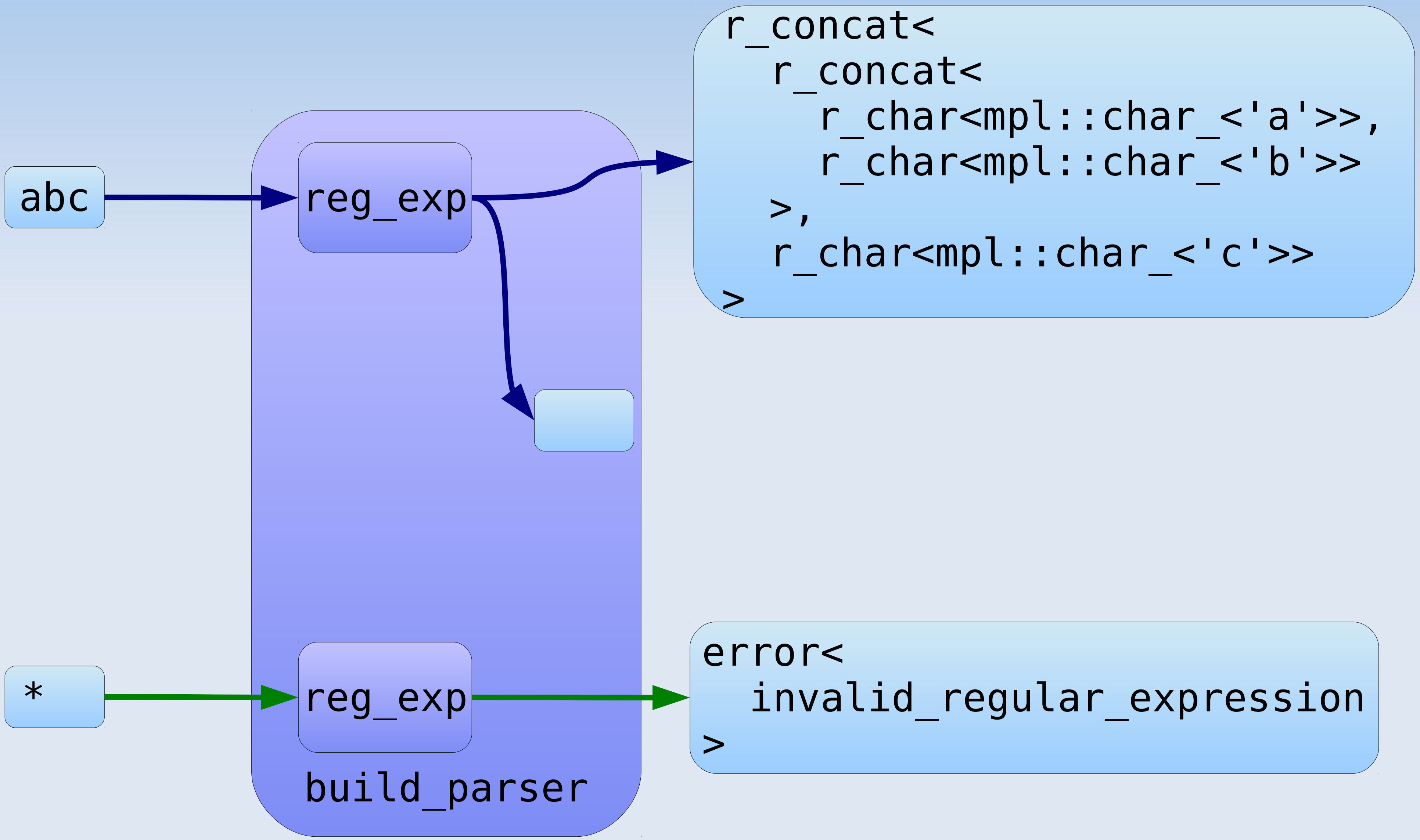


```
r_concat<
  r_concat<
    r_char<mpl::char_<'a'>>,
    r_char<mpl::char_<'b'>>
  >,
  r_char<mpl::char_<'c'>>
>
```

```
error<
  invalid_regular_expression
>
```

reg_exp

Parsing regular expressions



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

```
error<
  invalid_regular_expression
>
```

```
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