Embedding domain-specific languages into C++

Ábel Sinkovics

There are so many ways of embedding a DSL into C++... How should I approach it?

Embedding domain-specific languages into C++

Ábel Sinkovics

There are so many ways of embedding a DSL into C++... How should I approach it?

Embedding domain-specific languages into C++

Ábel Sinkovics

Are there so ***many*** ways of embedding a DSL into C++?

Agenda

- Domain-specific languages
- Embedding an example DSL
- DSL embedding methods
- Measurements

What is C++?

Template meta-programming!

Class hierarchies

A hybrid language

Buffer overflows

A multi-paradigm programming language

It’s C!

Classes

Embedded systems programming language

Too big!

Low level!

Generic programming

An object-oriented programming language

A random collection of features

What is C++?

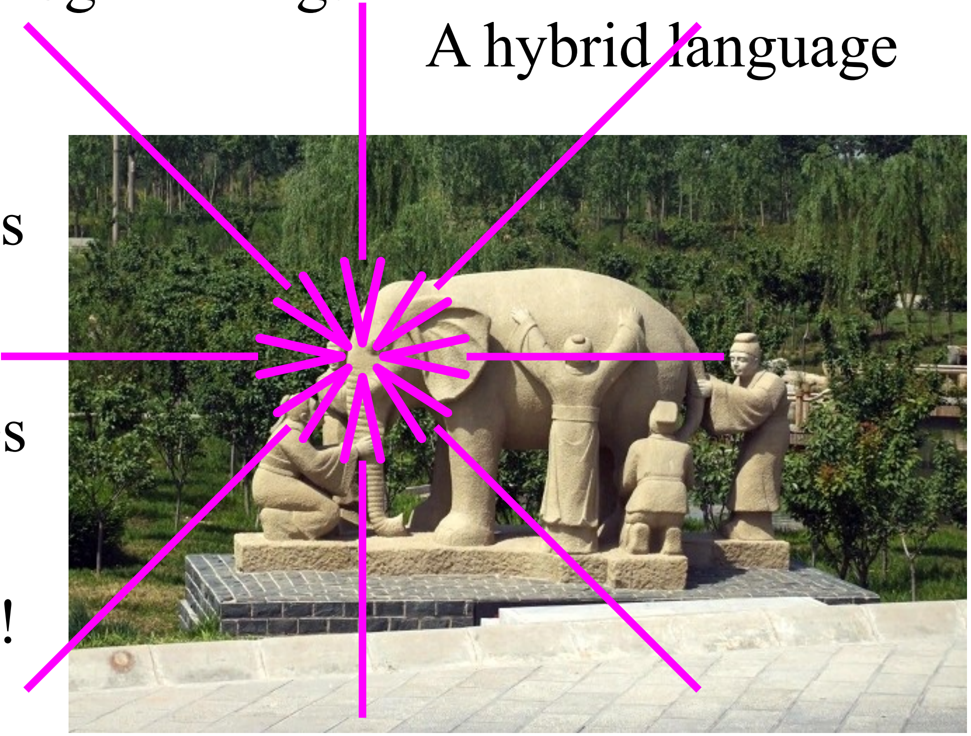
Template meta-programming!

Buffer overflows

Classes

Too big!

A hybrid language



Generic programming

An object-oriented programming language

Class hierarchies

A multi-paradigm programming language

It’s C!

Embedded systems programming language

Low level!

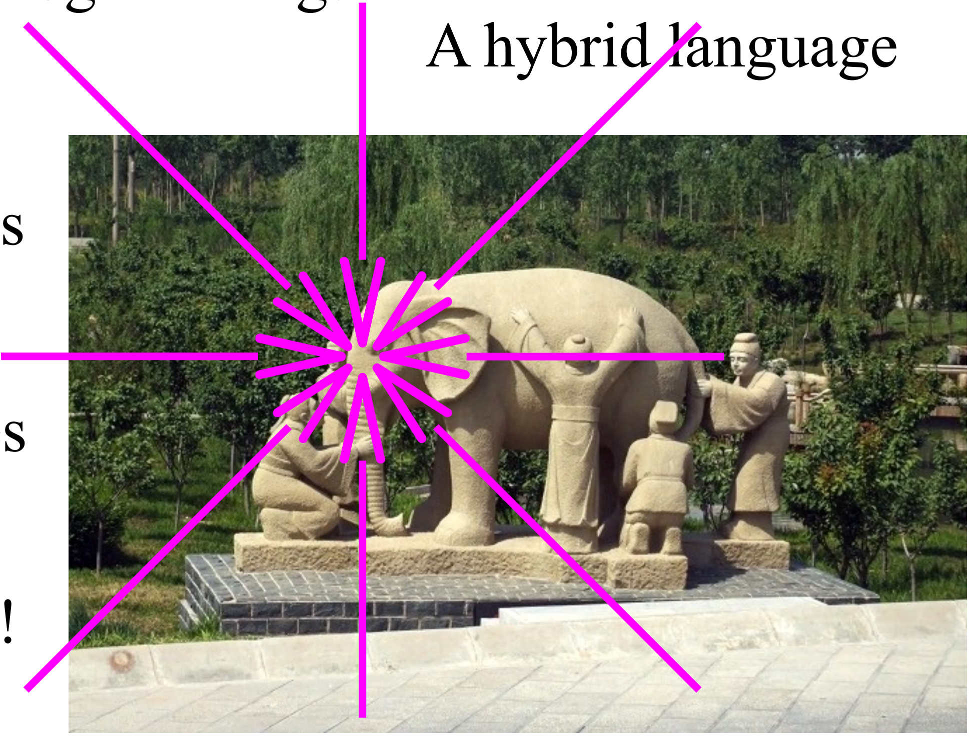
A random collection of features

What is C++?

Template meta-programming!

Class hierarchies

A hybrid language



Buffer overflows

A multi-paradigm programming language

It's C!

Classes

Embedded systems programming language

Too big!

Low level!

Generic programming

An object-oriented programming language

A random collection of features

Host language for embedded DSLs

Domain-specific language

”A *computer programming language of limited expressiveness focused on a particular domain.*”

Martin Fowler, Domain-Specific Languages

Domain-specific language

- Printf (Text formatting)
- Regular expressions (Text search)
- SQL (Database)
- Lex/Yacc (Parsing)
- *Make (Build system)
- Graphviz (Graphs)
- CSS (Website formatting)
- Cron (Scheduling)
- …

Benefits of using DSLs

- Makes the code (more) readable for domain experts

Benefits of using DSLs

- Makes the code (more) readable for domain experts
- Shorten the development cycle

Benefits of using DSLs

- Makes the code (more) readable for domain experts
- Shorten the development cycle
- Easier to maintain

Benefits of using DSLs

- Makes the code (more) readable for domain experts
- Shorten the development cycle
- Easier to maintain
- Enables domain-specific optimisations

Benefits of using DSLs

- Makes the code (more) readable for domain experts
- Shorten the development cycle
- Easier to maintain
- Enables domain-specific optimisations
- Can introduce other programming paradigm

Challenges of using DSLs

- Yet another...
  - language to learn
  - tool to integrate

Challenges of using DSLs

- Yet another...
  - language to learn
  - tool to integrate
- Needs to be processed
  - Error reporting
  - Debugging
  - Maintenance
  - …

Categories

- Standalone

- Embedded

Categories

- Standalone
  - The entire program is written in the DSL
  - Example: Make
- Embedded

Categories

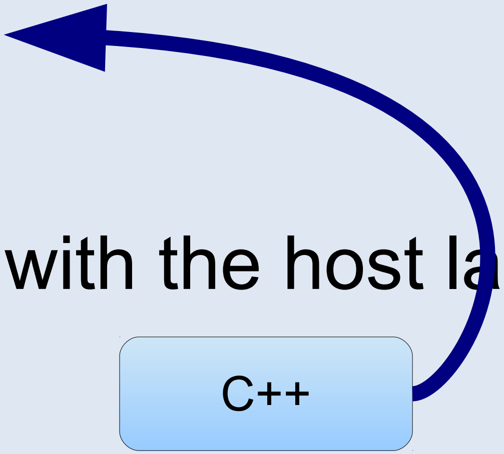
- Standalone
  - The entire program is written in the DSL
  - Example: Make
- Embedded
  - Parts of a larger program are written in the DSL
  - There is a host language
  - Example: SQL

Categories

- Standalone
  - The entire program is written in the DSL
  - Example: Make
- Embedded
  - Parts of a larger program are written in the DSL
  - There is a host language
  - Example: SQL
  - +Challenge: cooperation with the host language

Categories

- Standalone
  - The entire program is written in the DSL
  - Example: Make
- Embedded
  - Parts of a larger program are written in the DSL
  - There is a host language
  - Example: SQL
  - +Challenge: cooperation with the host language



C++

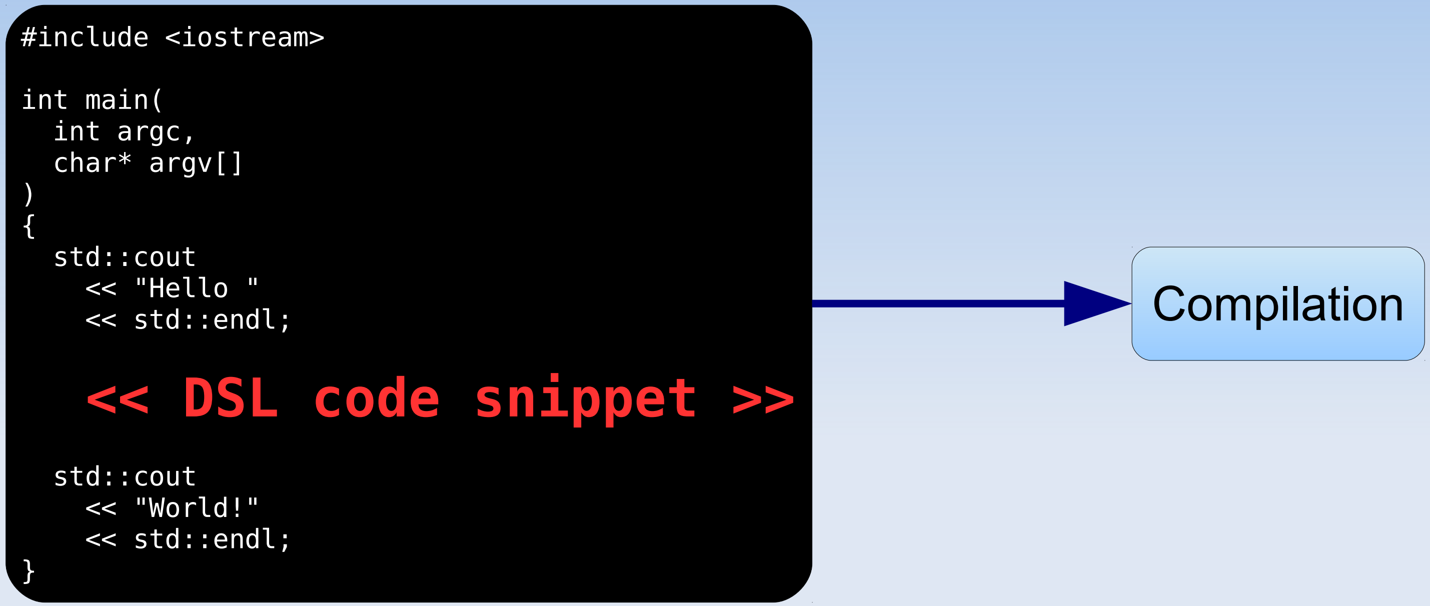
Embedding a DSL

```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;




  std::cout
    << "World!"
    << std::endl;
}
```

Embedding a DSL

```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

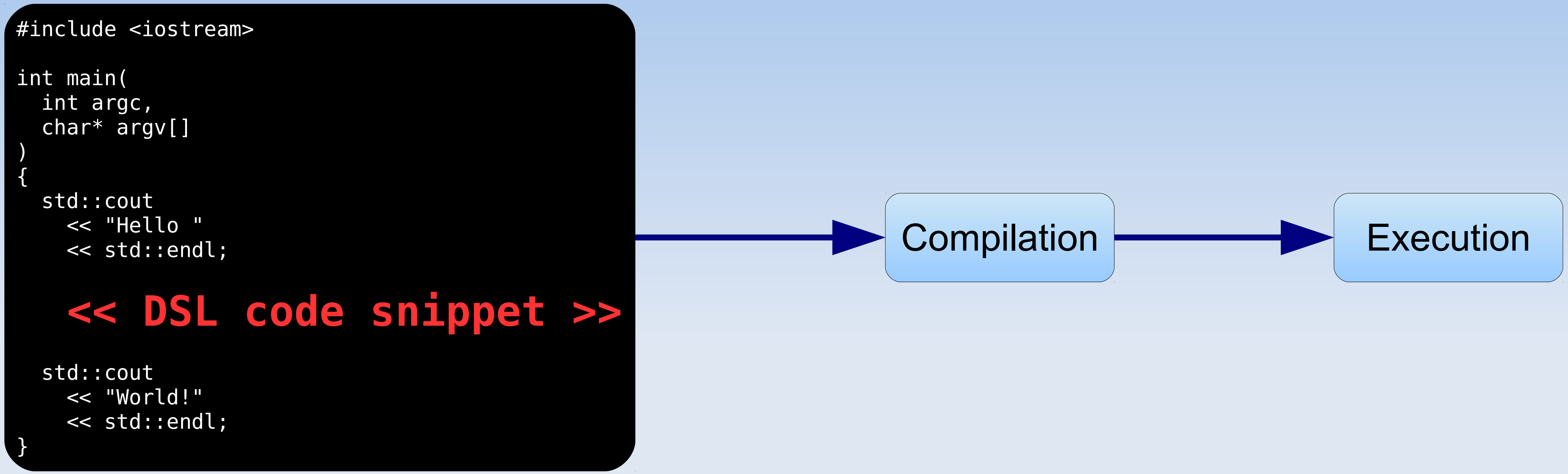
Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Compilation

Embedding a DSL



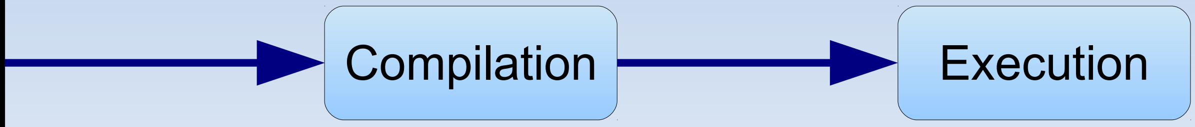
```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Compilation

Execution

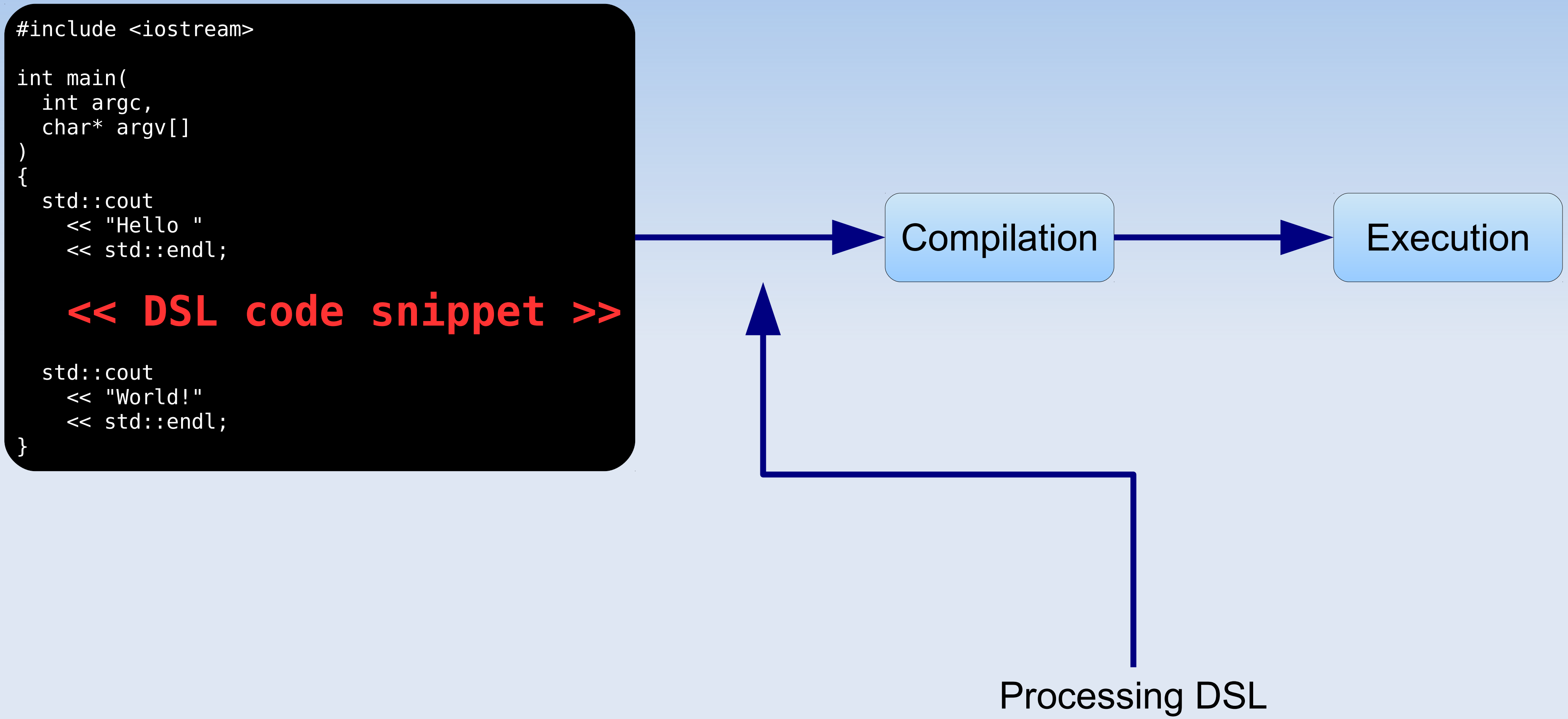
Embedding a DSL

```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```



Processing DSL

Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

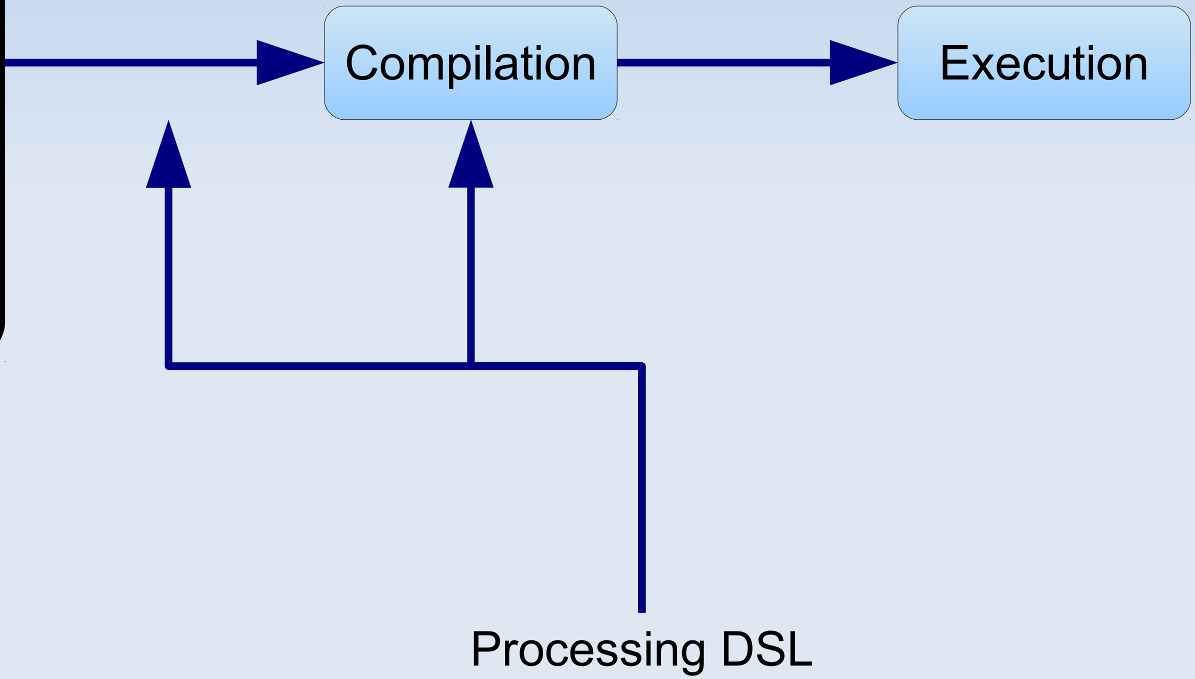
Compilation

Execution

Processing DSL

Embedding a DSL

```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```



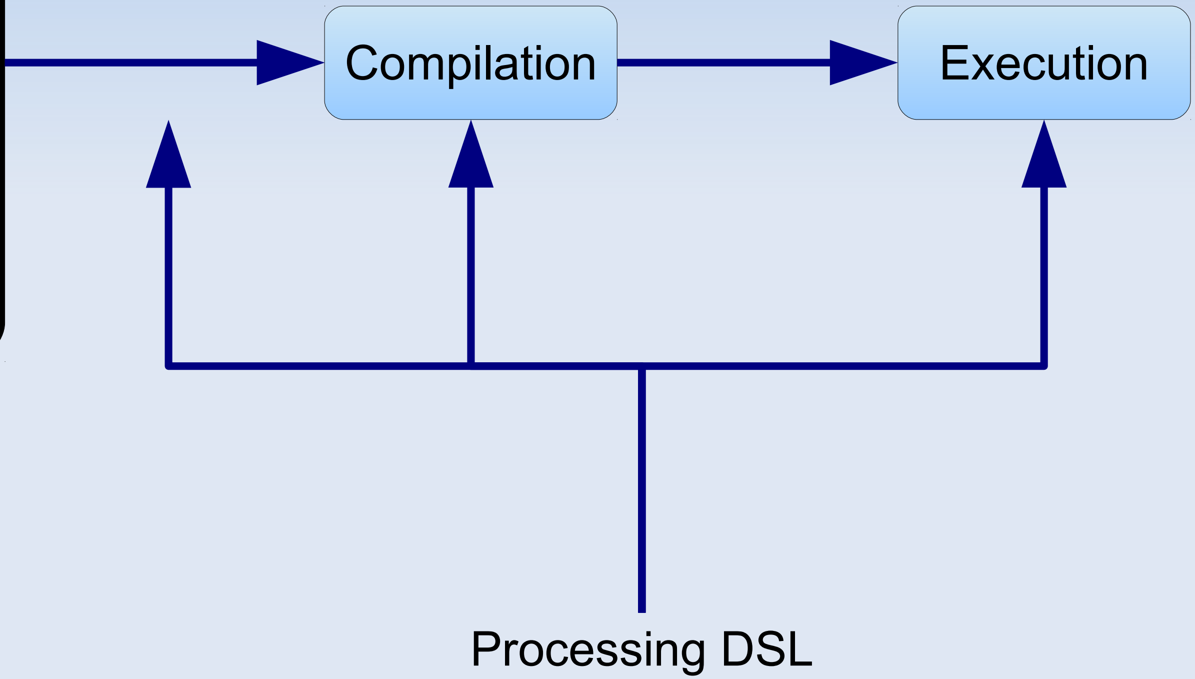
Compilation

Execution

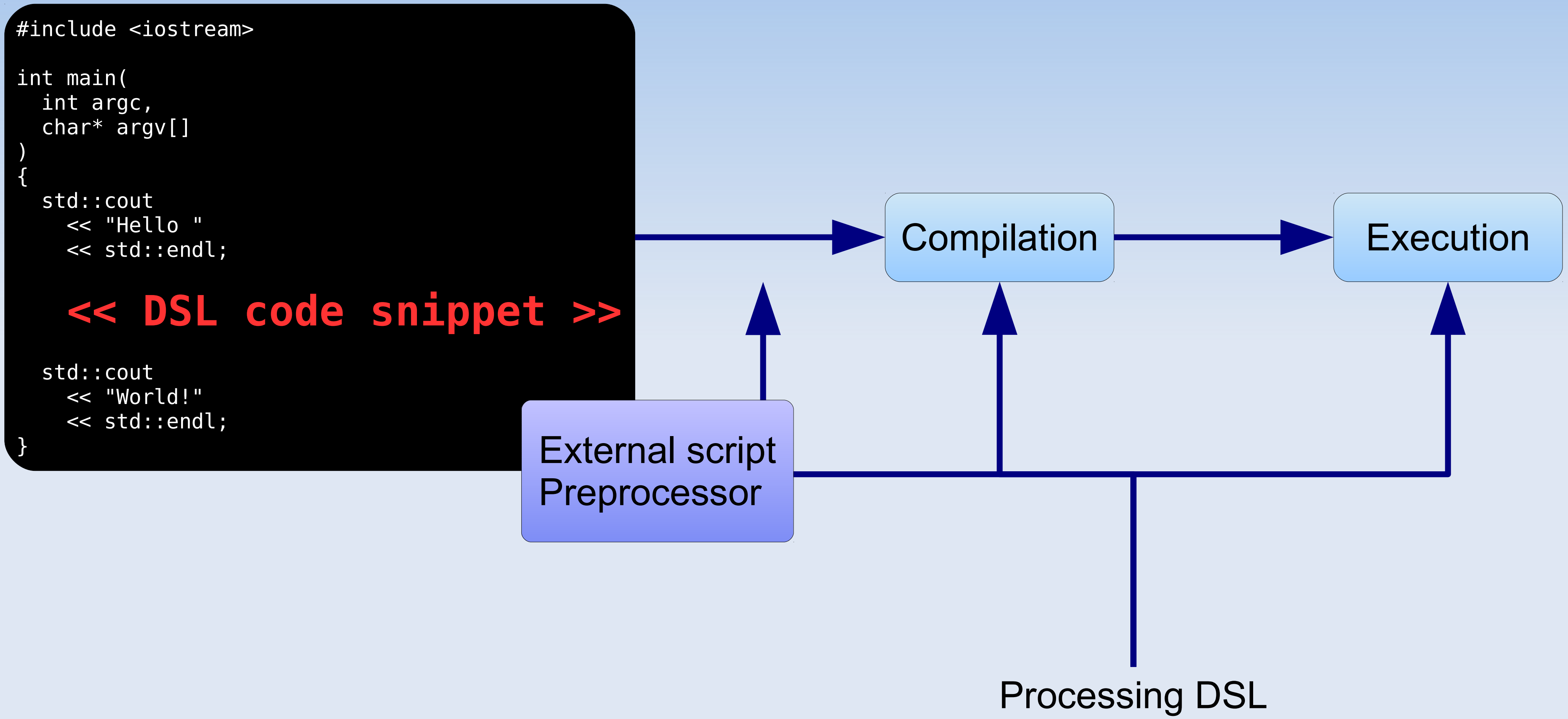
Processing DSL

Embedding a DSL

```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

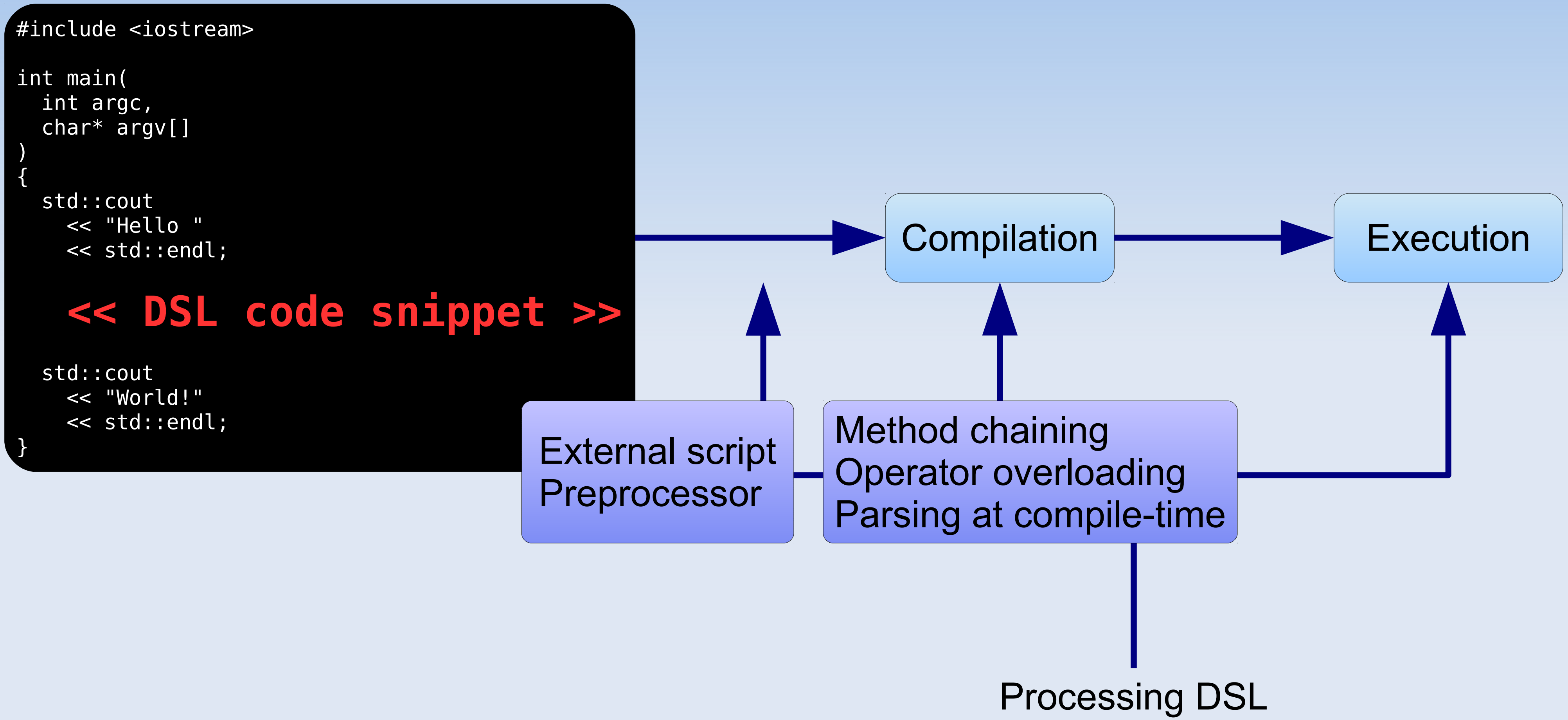


Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Compilation

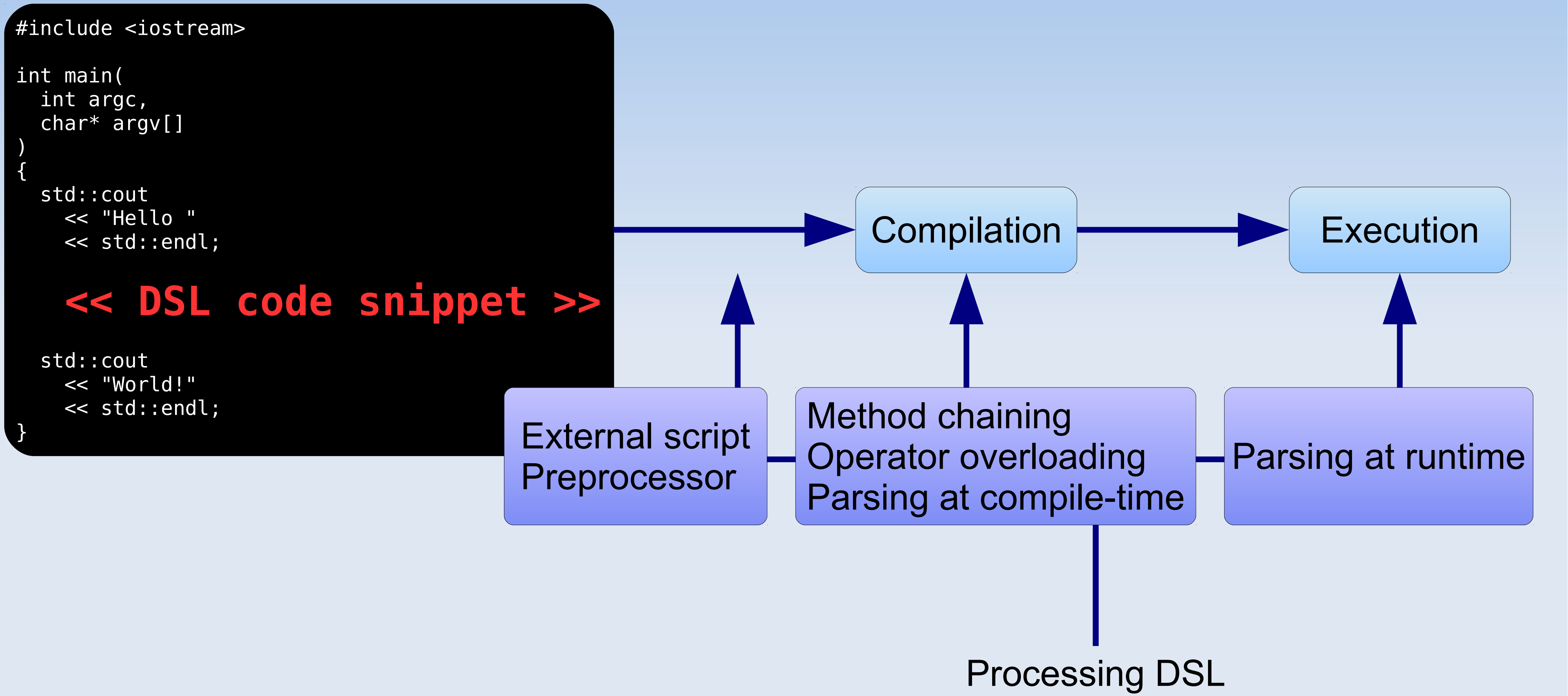
Execution

External script
Preprocessor

Method chaining
Operator overloading
Parsing at compile-time

Processing DSL

Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Compilation

Execution

External script
Preprocessor

Method chaining
Operator overloading
Parsing at compile-time

Parsing at runtime

Processing DSL

Processing before compilation

- External script
  - Qt: moc
  - Oracle Pro*C/C++
- Preprocessor
  - Boost.ConceptCheck
  - Boost.Foreach
  - Boost.StaticAssert
  - Unit testing libraries

Processing before compilation

- External script
  - Qt: moc
  - Oracle Pro*C/C++
- Preprocessor

```
class MyClass : public QObject
{
  Q_OBJECT
  Q_CLASSINFO("Author", "Oscar Peterson")
  Q_CLASSINFO("Status", "Active")

public:
  MyClass(QObject *parent = 0);
  ~MyClass();
};
```

Processing before c

- External script
    - Qt: moc
    - Oracle Pro*C/C++
- Preprocessor

```
class MyClass : public QObject
{
  Q_OBJECT
  Q_CLASSINFO("Author", "Oscar Peterso
  Q_CLASSINFO("Status", "Active")

public:
  MyClass(QObject *parent = 0);
  ~MyClass();
};
```

```
/****************************************************************************
** Meta object code from reading C++ file 'test.cpp'
**
** Created: Fri Apr 18 20:20:51 2014
**      by: The Qt Meta Object Compiler version 63 (Qt 4.8.4)
**
** WARNING! All changes made in this file will be lost!
*****************************************************************************/

#if !defined(Q_MOC_OUTPUT_REVISION)
#error "The header file 'test.cpp' doesn't include <QObject>."
#elif Q_MOC_OUTPUT_REVISION != 63
#error "This file was generated using the moc from 4.8.4. It"
#error "cannot be used with the include files from this version of Qt."
#error "(The moc has changed too much.)"
#endif

QT_BEGIN_MOC_NAMESPACE
static const uint qt_meta_data_MyClass[] = {

 // content:
       6,       // revision
       0,       // classname
       2,   14, // classinfo
       0,    0, // methods
       0,    0, // properties
       0,    0, // enums/sets
       0,    0, // constructors
       0,       // flags
       0,       // signalCount

 // classinfo: key, value
      23,    8,
      37,   30,

       0        // eod
};

static const char qt_meta_stringdata_MyClass[] = {
    "MyClass\0Oscar Peterson\0Author\0Active\0"
    "Status\0"
};

void MyClass::qt_static_metacall(QObject *_o, QMetaObject::Call _c, int _id, void **_a)
{
    Q_UNUSED(_o);
    Q_UNUSED(_id);
    Q_UNUSED(_c);
    Q_UNUSED(_a);
}

const QMetaObjectExtraData MyClass::staticMetaObjectExtraData = {
    0,  qt_static_metacall
};

const QMetaObject MyClass::staticMetaObject = {
    { &QObject::staticMetaObject, qt_meta_stringdata_MyClass,
      qt_meta_data_MyClass, &staticMetaObjectExtraData }
};

#ifdef Q_NO_DATA_RELOCATION
const QMetaObject &MyClass::getStaticMetaObject() { return staticMetaObject; }
#endif //Q_NO_DATA_RELOCATION

const QMetaObject *MyClass::metaObject() const
{
    return QObject::d_ptr->metaObject ? QObject::d_ptr->metaObject : &staticMetaObject;
}

void *MyClass::qt_metacast(const char *_clname)
{
    if (!_clname) return 0;
    if (!strcmp(_clname, qt_meta_stringdata_MyClass))
        return static_cast<void*>(const_cast< MyClass*>(this));
    return QObject::qt_metacast(_clname);
}

int MyClass::qt_metacall(QMetaObject::Call _c, int _id, void **_a)
{
    _id = QObject::qt_metacall(_c, _id, _a);
    if (_id < 0)
        return _id;
    return _id;
}
QT_END_MOC_NAMESPACE
```

Processing before compilation

```
BOOST_FOREACH( char ch, hello )
{
  std::cout << ch;
}
```

  - Qt: moc
  - Oracle Pro*C/C++
- Preprocessor
  - Boost.ConceptCheck
  - Boost.Foreach
  - Boost.StaticAssert
  - Unit testing libraries

Processing before

```
BOOST_FOREACH( char ch, hello )
{
  std::cout << ch;
}
```

  - Qt: moc
  - Oracle Pro*C/C++
- Preprocessor
  - Boost.ConceptCheck
  - Boost.Foreach
  - Boost.StaticAssert
  - Unit testing libraries

Preprocessor

```
if (bool _foreach_is_rvalue9 = false) {}
else if (
  boost::foreach_detail_::auto_any_t _foreach_col9 =
    boost::foreach_detail_::contain(
      (true ? boost::foreach_detail_::make_probe((hello), _foreach_is_rvalue9) : (hello)),
      (boost::foreach_detail_::should_copy_impl(
        true ? 0 :
          boost::foreach_detail_::or_(
            boost::foreach_detail_::is_array_(hello),
            boost_foreach_is_noncopyable(
              boost::foreach_detail_::to_ptr(hello),
              boost_foreach_argument_dependent_lookup_hack_value),
            boost::foreach_detail_::not_(boost::foreach_detail_::is_const_(hello))),
        true ? 0 :
          boost::foreach_detail_::and_(
            boost::foreach_detail_::not_(
              boost_foreach_is_noncopyable(
                boost::foreach_detail_::to_ptr(hello),
                boost_foreach_argument_dependent_lookup_hack_value)),
            boost_foreach_is_lightweight_proxy(
              boost::foreach_detail_::to_ptr(hello),
              boost_foreach_argument_dependent_lookup_hack_value)),
          &_foreach_is_rvalue9)))) {}
else if (
  boost::foreach_detail_::auto_any_t _foreach_cur9 =
    boost::foreach_detail_::begin(
      _foreach_col9,
      (true ? 0 : boost::foreach_detail_::encode_type(hello, boost::foreach_detail_::is_const_(hello))),
      (boost::foreach_detail_::should_copy_impl(
        true ? 0 :
          boost::foreach_detail_::or_(
            boost::foreach_detail_::is_array_(hello),
            boost_foreach_is_noncopyable(
              boost::foreach_detail_::to_ptr(hello),
              boost_foreach_argument_dependent_lookup_hack_value),
            boost::foreach_detail_::not_(boost::foreach_detail_::is_const_(hello))),
        true ? 0 :
          boost::foreach_detail_::and_(
            boost::foreach_detail_::not_(
              boost_foreach_is_noncopyable(
                boost::foreach_detail_::to_ptr(hello),
                boost_foreach_argument_dependent_lookup_hack_value)),
            boost_foreach_is_lightweight_proxy(
              boost::foreach_detail_::to_ptr(hello),
              boost_foreach_argument_dependent_lookup_hack_value)),
        &_foreach_is_rvalue9)))) {}
else if (
  boost::foreach_detail_::auto_any_t _foreach_end9 =
    boost::foreach_detail_::end(
      _foreach_col9,
      (true ? 0 : boost::foreach_detail_::encode_type(hello, boost::foreach_detail_::is_const_(hello))),
      (boost::foreach_detail_::should_copy_impl(
        true ? 0 :
          boost::foreach_detail_::or_(
            boost::foreach_detail_::is_array_(hello),
            boost_foreach_is_noncopyable(
              boost::foreach_detail_::to_ptr(hello),
              boost_foreach_argument_dependent_lookup_hack_value),
            boost::foreach_detail_::not_(boost::foreach_detail_::is_const_(hello))),
        true ? 0 :
          boost::foreach_detail_::and_(
            boost::foreach_detail_::not_(
              boost_foreach_is_noncopyable(
                boost::foreach_detail_::to_ptr(hello),
                boost_foreach_argument_dependent_lookup_hack_value)),
            boost_foreach_is_lightweight_proxy(
              boost::foreach_detail_::to_ptr(hello),
              boost_foreach_argument_dependent_lookup_hack_value)),
        &_foreach_is_rvalue9)))) {}
else for (
  bool _foreach_continue9 = true;
  _foreach_continue9 &&
    !boost::foreach_detail_::done(
      _foreach_cur9,
      _foreach_end9,
      (true ? 0 : boost::foreach_detail_::encode_type(hello, boost::foreach_detail_::is_const_(hello))));
  _foreach_continue9 ?
    boost::foreach_detail_::next(
      _foreach_cur9,
      (true ? 0 : boost::foreach_detail_::encode_type(hello, boost::foreach_detail_::is_const_(hello)))) :
    (void)0)
  if (boost::foreach_detail_::set_false(_foreach_continue9)) {}
  else for (
    char ch =
      boost::foreach_detail_::deref(
        _foreach_cur9,
        (true ? 0 : boost::foreach_detail_::encode_type(hello, boost::foreach_detail_::is_const_(hello))));
    !_foreach_continue9;
    _foreach_continue9 = true)
{
  std::cout << ch;
}
```

Processing at compile-time

- Method chaining
  - sqlpp11
  - Boost.Assign
- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XlXpressive

Processing at compile-time

- Method chaining
  - sqlpp11
  - Boost.Assign
- Operator overloading
  - Boos
  - Boos
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XIXpressive

```
select(foo.name, foo.hasFun)
```

Processing at compile-time

- Method chaining
  - sqlpp11
  - Boost.Assign
- Operator overloading
  - Boos
  - Boos
  - Boost.Phoenix

```
select(foo.name, foo.hasFun)
  .from(foo)
```

- Parsing at compile-time
  - Safe_printf
  - XlXpressive

Processing at compile-time

- Method chaining
  - sqlpp11
  - Boost.Assign
- Operator overloading
  - Boos
  - Boos
  - Boost.Phoenix

```
select(foo.name, foo.hasFun)
  .from(foo)
  .where(foo.id > 17 and foo.name.like("%bar%")))
```

- Parsing at compile-time
  - Safe_printf
  - XlXpressive

Processing at compile-time

```
    char_('.')
```

-
    - Boost.Assign
- Operator overloading
    - Boost.Xpressive
    - Boost.Spirit
    - Boost.Phoenix
- Parsing at compile-time
    - Safe_printf
    - XIXpressive

Processing at compile-time

```
    char_('.')          | char_("a-z")
```

  - Boost.Assign
- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XlXpressive

Processing at compile-time

```
(char_('.')        | char_("a-z")        ) >>  char_('*')
```

- Boost.Assign
- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XlXpressive

Processing at compile-time

```
(char_('.')        | char_("a-z")        ) >> -char_('*')
```

- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XlXpressive

Processing at compile-time

```
*((char_('.')         | char_("a-z")         ) >> -char_('*')      )
```

-
    - Boost.Assign
- Operator overloading
    - Boost.Xpressive
    - Boost.Spirit
    - Boost.Phoenix
- Parsing at compile-time
    - Safe_printf
    - XlXpressive

Processing at compile-time

```
*((char_('.')[a_any] | char_("a-z")[a_char]) >> -char_('*')[rep])
```

-
  - Boost.Assign
- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XlXpressive

Processing at compile-time

```
std::string s("foo bar");

boost::spirit::qi::parse(
  s.begin(), s.end(),
  *((char_('.')[a_any] | char_("a-z")[a_char]) >> -char_('*')[rep])
)
```

- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XIXpressive

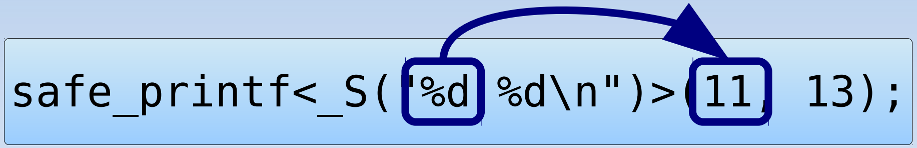
Processing at compile-time

- Method chaining
  - sqlpp11
  - Boost.Assign
- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XlXpressive

```
safe_printf<_S("%d %d\n")>(11, 13);
```

Processing at compile-time

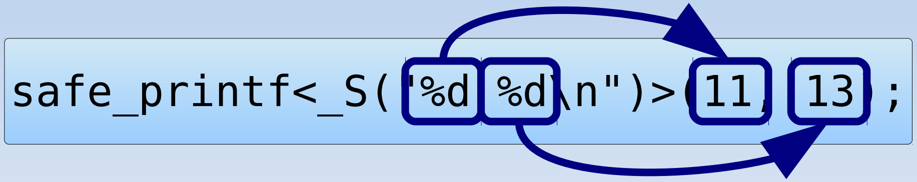
- Method chaining
  - sqlpp11
  - Boost.Assign
- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XlXpressive



```
safe_printf<_S("%d %d\n")>(11, 13);
```

Processing at compile-time

- Method chaining
  - sqlpp11
  - Boost.Assign
- Operator overloading
  - Boost.Xpressive
  - Boost.Spirit
  - Boost.Phoenix
- Parsing at compile-time
  - Safe_printf
  - XIXpressive



```
safe_printf<_S("%d %d\n")>(11, 13);
```

Parsing at runtime

- Text formatting
  - printf
- Regular expressions
  - Boost.Xpressive
  - std::regex
- SQL
  - SOCI
  - SQLAPI++
  - MySQL++

Parsing at runtime

- Text formatting
  - printf
- Regular expressions
  - Boost.Xpressive
  - std::regex
- SQL
  - SOCI
  - SQLAPI++
  - MySQL++

```
printf("%d %d\n", 11, 13);
```

Parsing at runtime

- Text formatting
  - printf
- Regular expressions
  - Boost.Xpressive
  - std::regex
- SQL
  - SOCI
  - SQLAPI++
  - MySQL++

```
std::regex("(sub)(.*)")
```

Parsing at runtime

- Text formatting
  - printf
- Regular expressions
  - Boost.Xpressive
  - std::regex
- SQL
  - SOCI
  - SQLAPI++
  - MySQL++

```
mysqlpp::Query query = conn.query("select item from stock");
```

Example

- Regular expressions

Example

- Regular expressions
  - a-z

Example

- Regular expressions
  - a-z
  - .

Example

- Regular expressions
  - a-z
  - .
  - *

Example

```
.      abc
x      ab\
a*
```

- Regular expressions
  - a-z
  - .
  - *

Example

```
.      abc
x      ab\
a*
```

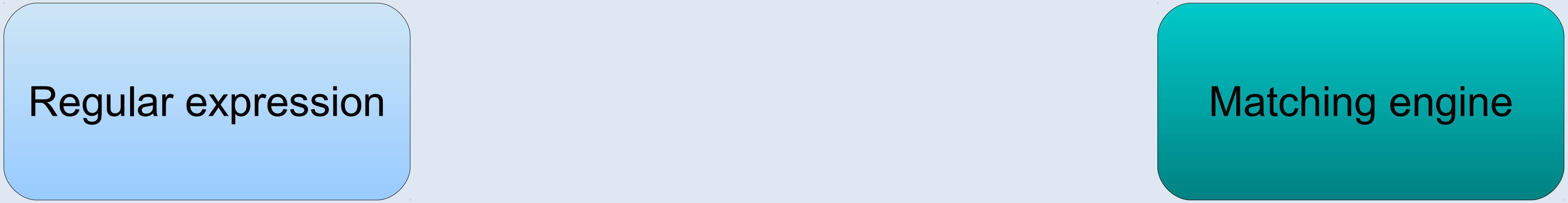
- Regular expressions
  - `a-z`
  - `.`
  - `*`

Matching engine

Example

```
.      abc
x      ab\
a*
```

- Regular expressions
  - a-z
  - .
  - *



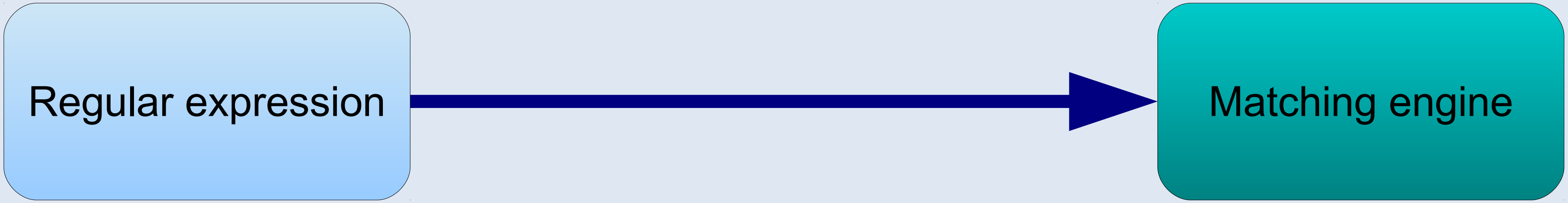
Regular expression

Matching engine

Example

```
.      abc
x      ab\
a*
```

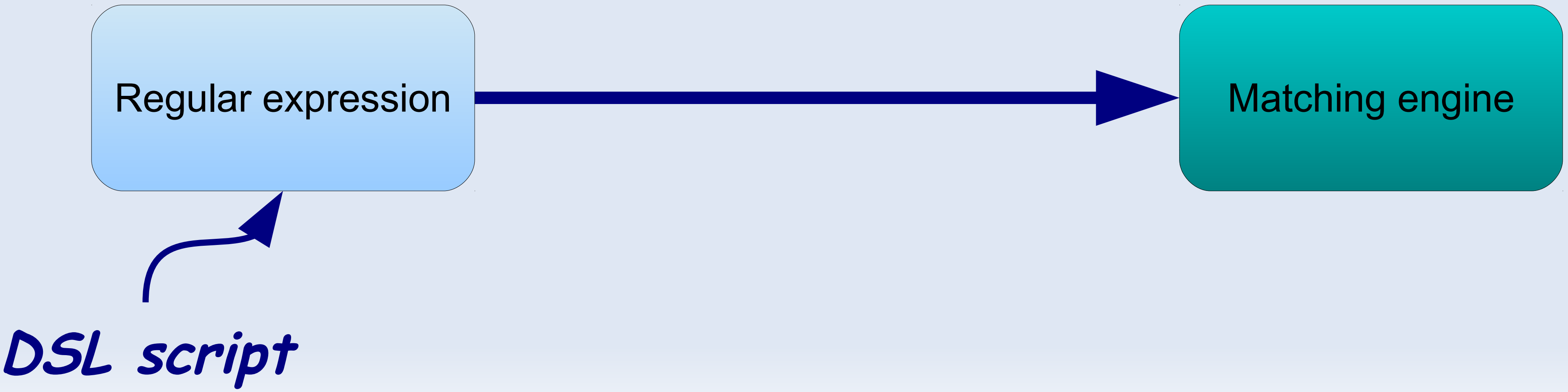
- Regular expressions
  - a-z
  - .
  - *



Example

```
.      abc
x      ab\
a*
```

- Regular expressions
  - a-z
  - .
  - *



Regular expression

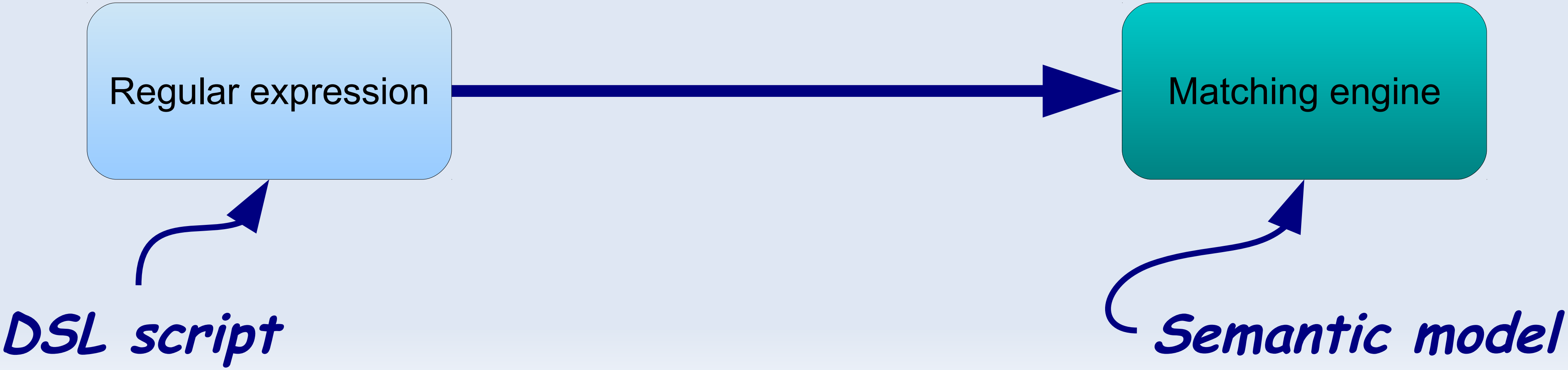
Matching engine

DSL script

Example

```
.      abc
x      ab\
a*
```

- Regular expressions
  - a-z
  - .
  - *



Regular expression

Matching engine

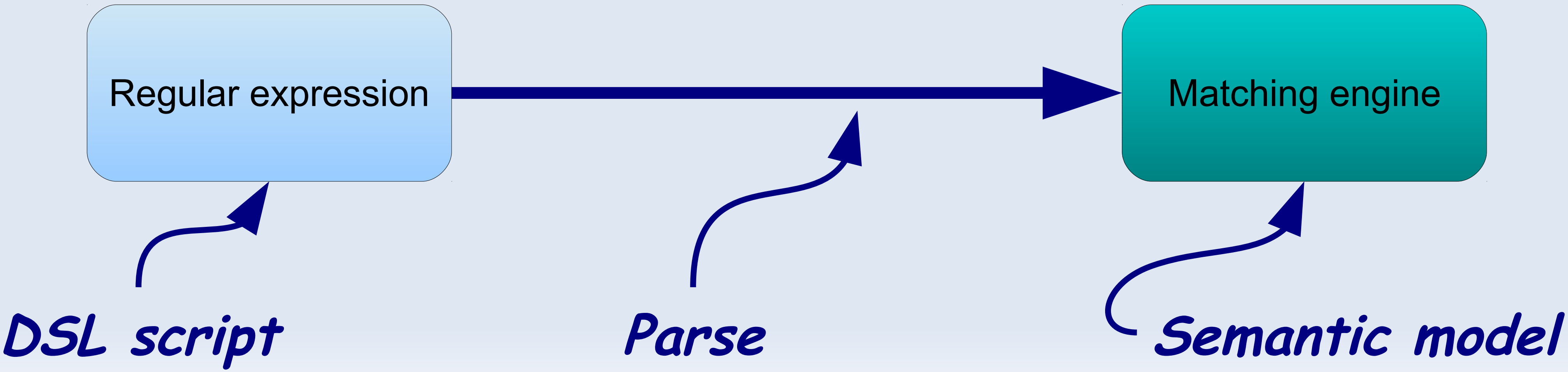
DSL script

Semantic model

Example

```
.      abc
x      ab\
a*
```

- Regular expressions
  - a-z
  - .
  - *



Regular expression

Matching engine

DSL script

Parse

Semantic model

Example

```
.     abc
x     ab\
a*
```

```
<< MATCHING_ENGINE >>                   re;
```

Example

```
.      abc
x      ab\
a*
```

```
  << MATCHING_ENGINE >>                     re;

std::string s("some text");
```

Example

```
.      abc
x      ab\
a*
```

```
  << MATCHING_ENGINE >>                  re;
std::string s("some text");
    auto i = re.match(s.begin(), s.end()))
```

Example

```
.     abc
x     ab\
a*
```

```
boost::optional<  std::string::iterator  >
```

```
  << MATCHING_ENGINE >>                    re;
std::string s("some text");
    auto i = re.match(s.begin(), s.end()))
```

Example

```
.      abc
x      ab\
a*
```

```
  << MATCHING_ENGINE >>                re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{                                                          }
```

Example

```
.     abc
x     ab\
a*
```

```
  << MATCHING_ENGINE >>                  re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
struct any    { /* … */ };
```

```
any                                         re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
struct any     { /* … */ };
```

```
struct any {




};
```

```
any                                         re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
struct any    { /* … */ };
```

```
struct any {
  template <class It>
                    match(It begin_, It end_) const {

  }
};
```

```
any                                     re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
struct any    { /* … */ };
```

```
struct any {
  template <class It>
  boost::optional<It> match(It begin_, It end_) const {

  }
};
```

```
any                                      re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
struct any    { /* … */ };
```

```
struct any {
  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    return begin_ == end_ ?                        :         ;
  }
};
```

```
any                                       re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
struct any    { /* … */ };
```

```
struct any {
  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    return begin_ == end_ ? boost::optional<It>() :         ;
  }
};
```

```
any                                    re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
struct any     { /* … */ };
```

```
struct any {
  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    return begin_ == end_ ? boost::optional<It>() : ++begin_;
  }
};
```

```
any                                         re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.    abc
x    ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
```

```
char_<'x'>                               re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
```

```
repeat<char_<'a'>>                       re(char_<'a'>());

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template
template
```

```
template <class E> struct repeat {
  repeat(E e_) : _e(e_) {}

  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    for (It i = begin_; i != end_; )
      if (auto j = _e.match(i, end_)) { i = *j; }
       else { return i; }
    return end_;
  }

  E _e;
};
```

```
repeat<char_<'a'>>                          re(char_<'a'>());

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>       struct char_   { /* … */ };
                        struct any     { /* … */ };
template <class    E>   struct repeat  { /* … */ };
template <class... Es>  struct seq     { /* … */ };
```

```
seq<char_<'a'>, char_<'b'>, char_<'c'>>  re(
  char_<'a'>(), char_<'b'>(), char_<'c'>());
std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>       struct char_  { /* … */ };
                        struct any    { /* … */ };
template <class     E>  struct repeat { /* … */ };
template <class... Es>  struct seq    { /* … */ };
// …
```

```
seq<char_<'a'>, char_<'b'>, char_<'c'>>  re(
  char_<'a'>(), char_<'b'>(), char_<'c'>());
std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>    struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

```
seq<char_<'a'>, char_<'b'>, char_<'\\'>> re(
  char_<'a'>(), char_<'b'>(), char_<'\\'>());
std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```cpp
class dyn_char {
public:
  dyn_char(char c_) : _c(c_) {}









private:
  char _c;
};
```

```
*/ };
*/ };
*/ };
*/ };
```

```cpp
  << MATCHING_ENGINE >>                    re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
class dyn_char {
public:
  dyn_char(char c_) : _c(c_) {}

  template <class It>
  boost::optional<It> match(It begin_, It end_) const
  {
    return
      begin_ != end_ && *begin_ == _c ?
        ++begin_ : boost::optional<It>();
  }
private:
  char _c;
};
```

```
*/ };
*/ };
*/ };
*/ };
```

```
  << MATCHING_ENGINE >>                          re;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
class dyn_char {
public:
  dyn_char(char c_) : _c(c_) {}

  template <class It>
  boost::optional<It> match(It begin_, It end_) const
  {
    return
      begin_ != end_ && *begin_ == _c ?
        ++begin_ : boost::optional<It>();
  }
private:
  char _c;
};
```

```
*/ };
*/ };
*/ };
*/ };
```

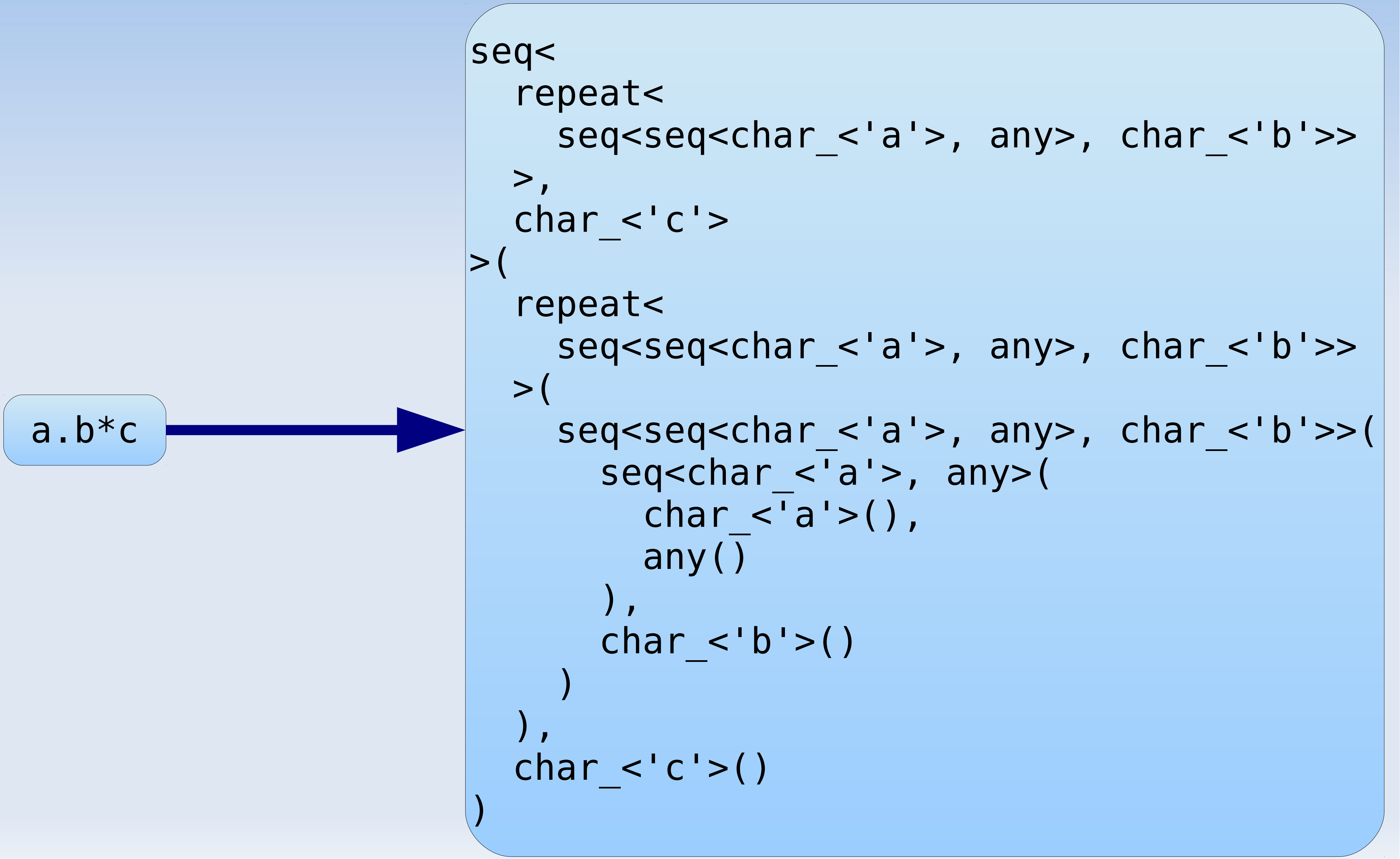
```
struct empty {
  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    return begin_;
  }
};
```

```
   << MATCHING_E

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

DSL embedding

```
seq<
  repeat<
    seq<seq<char_<'a'>, any>, char_<'b'>>
  >,
  char_<'c'>
>(
  repeat<
    seq<seq<char_<'a'>, any>, char_<'b'>>
  >(
    seq<seq<char_<'a'>, any>, char_<'b'>>(
      seq<char_<'a'>, any>(
        char_<'a'>(),
        any()
      ),
      char_<'b'>()
    )
  ),
  char_<'c'>()
)
```

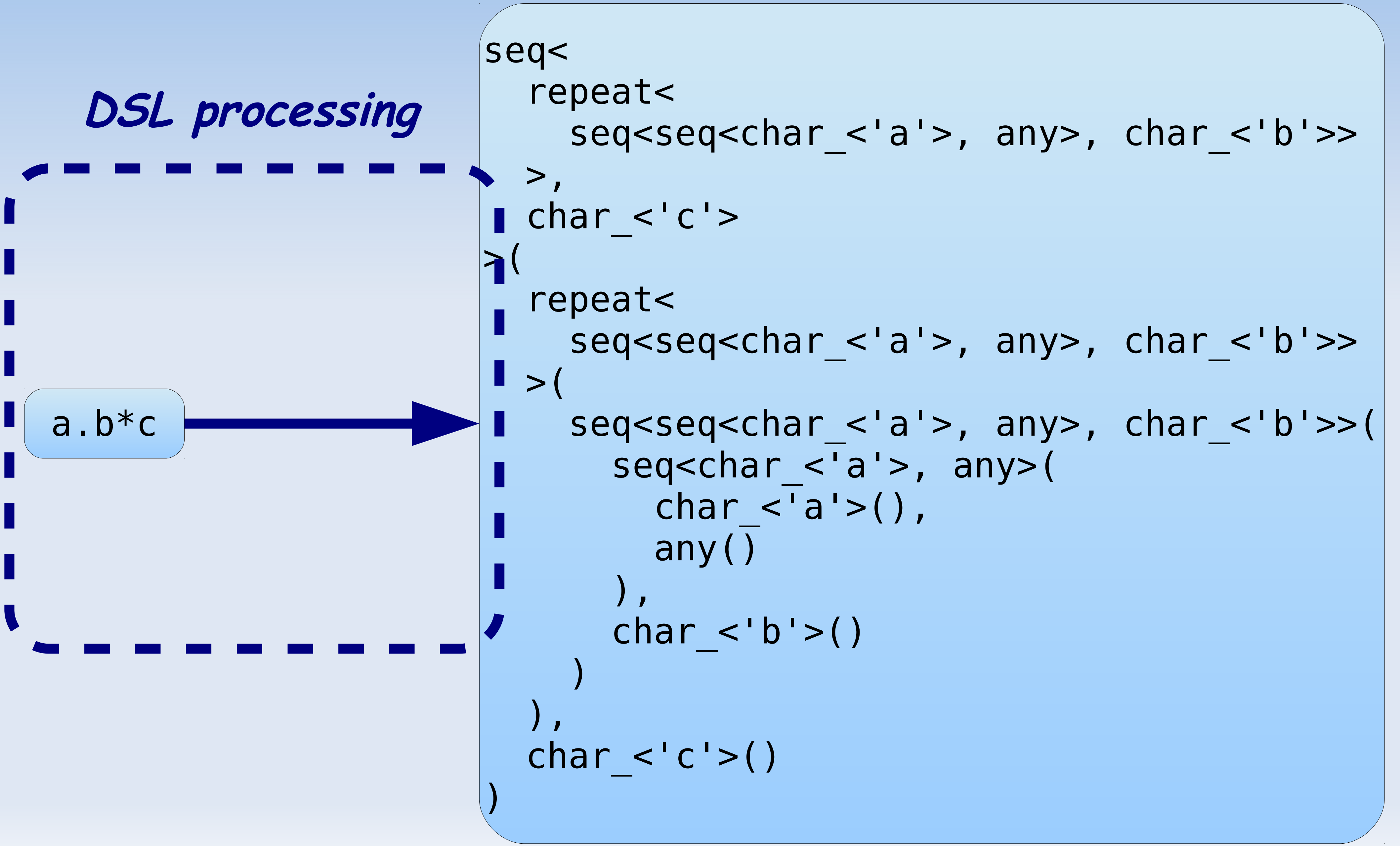
DSL embedding



a.b*c

```
seq<
  repeat<
    seq<seq<char_<'a'>, any>, char_<'b'>>
  >,
  char_<'c'>
>(
  repeat<
    seq<seq<char_<'a'>, any>, char_<'b'>>
  >(
    seq<seq<char_<'a'>, any>, char_<'b'>>(
      seq<char_<'a'>, any>(
        char_<'a'>(),
        any()
      ),
      char_<'b'>()
    )
  ),
  char_<'c'>()
)
```

DSL embedding



DSL processing

a.b*c

```
seq<
  repeat<
    seq<seq<char_<'a'>, any>, char_<'b'>>
  >,
  char_<'c'>
>(
  repeat<
    seq<seq<char_<'a'>, any>, char_<'b'>>
  >(
    seq<seq<char_<'a'>, any>, char_<'b'>>(
      seq<char_<'a'>, any>(
        char_<'a'>(),
        any()
      ),
      char_<'b'>()
    )
  ),
  char_<'c'>()
)
```

Evaluation

Using the DSL

Implementing the DSL

Evaluation

Using the DSL

No syntax changes

Implementing the DSL

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation

Implementing the DSL

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation
- Readable error messages

Implementing the DSL

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation
- Readable error messages
- Usable in library headers

Implementing the DSL

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation
- Readable error messages
- Usable in library headers
- Code completion

Implementing the DSL

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation
- Readable error messages
- Usable in library headers
- Code completion

Implementing the DSL

- Only standard C++

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation
- Readable error messages
- Usable in library headers
- Code completion

Implementing the DSL

- Only standard C++
- ”Normal” C++

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation
- Readable error messages
- Usable in library headers
- Code completion

Implementing the DSL

- Only standard C++
- ”Normal” C++
- No metaprogramming

Evaluation

Using the DSL

- No syntax changes
- Compile-time validation
- Readable error messages
- Usable in library headers
- Code completion

Implementing the DSL

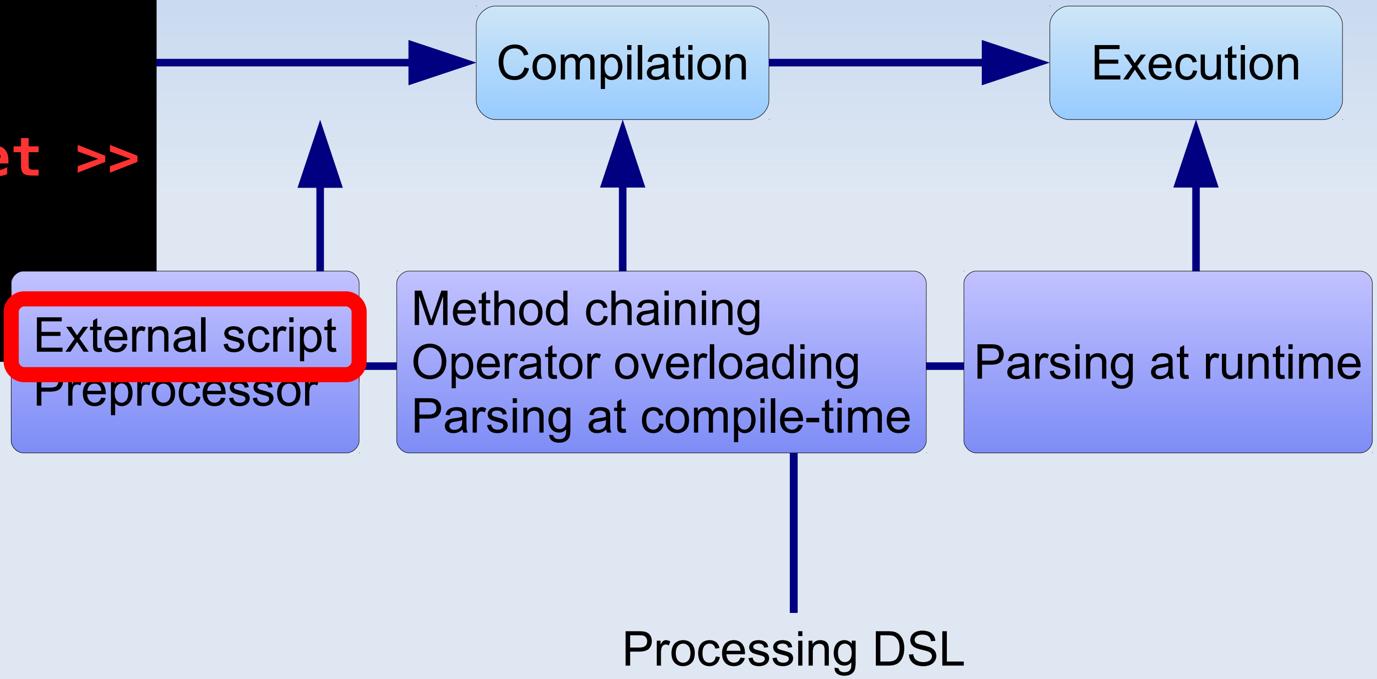
- Only standard C++
- ”Normal” C++
- No metaprogramming
- No build system support

Just one example...

- Compact notation
- No interaction with the host language
- One matching engine

Embedding a DSL

```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```



Compilation

Execution

External script

Preprocessor

Method chaining
Operator overloading
Parsing at compile-time

Parsing at runtime

Processing DSL

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class    E>    struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

```
auto re = REGEX(.);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>       struct char_  { /* … */ };
                        struct any    { /* … */ };
template <class    E>   struct repeat { /* … */ };
template <class... Es>  struct seq    { /* … */ };
// …
```

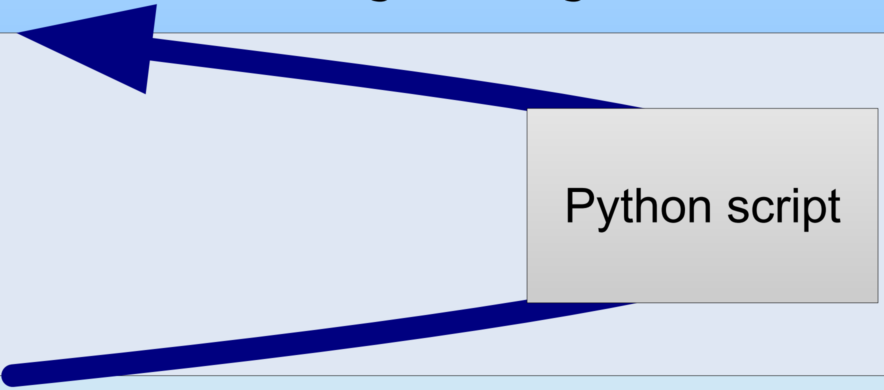
Python script

```
auto re = REGEX(.);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = any();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



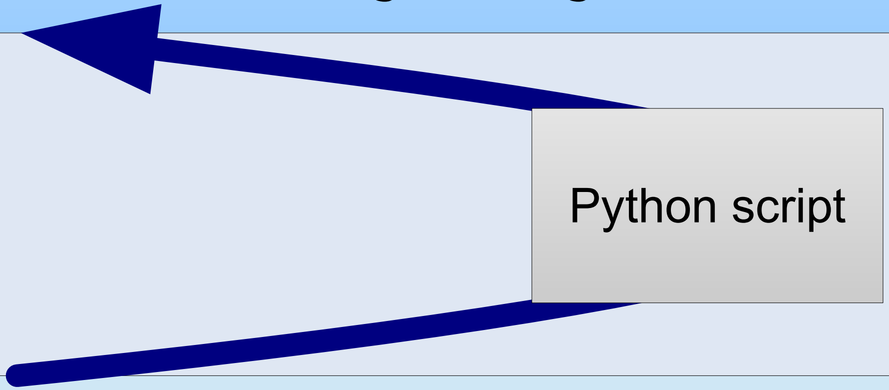
Python script

```
auto re = REGEX(.);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = char_<'x'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



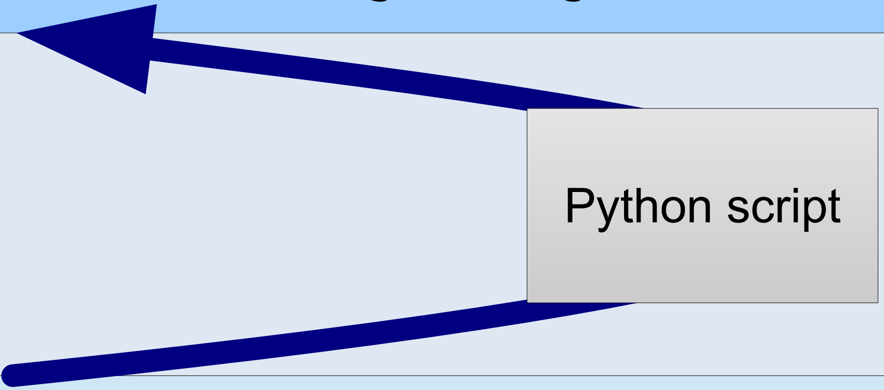
Python script

```
auto re = REGEX(x);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
auto re = repeat<char_<'a'>>(char_<'a'>());

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



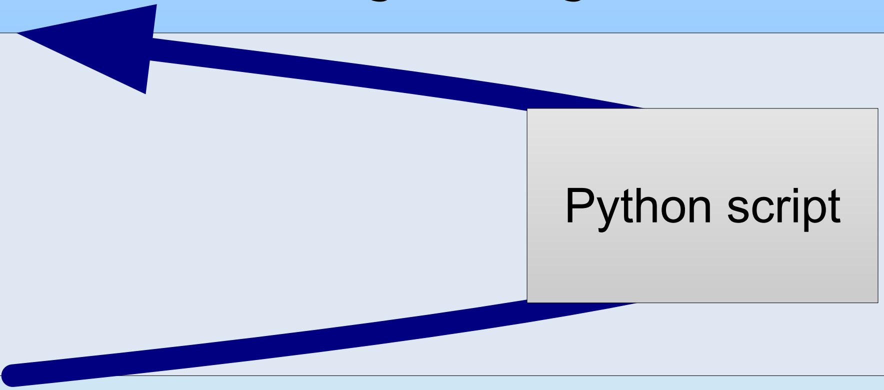
Python script

```
auto re = REGEX(a*);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.       abc
x       ab\
a*
```

```
auto re = seq<char_<'a'>, char_<'b'>, char_<'c'>>(
  char_<'a'>(), char_<'b'>(), char_<'c'>());
std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



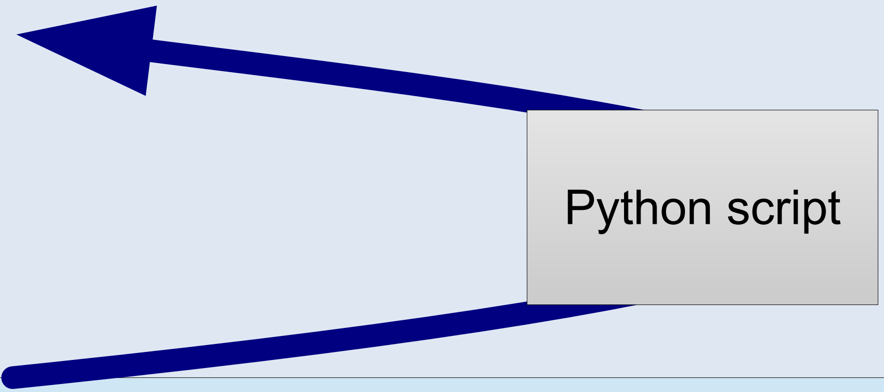
```
auto re = REGEX(abc);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class    Es> struct seq    { /* … */ };
// …
```

```
<stdin> 1:22 Invalid character: \
```



Python script

```
auto re = REGEX(ab\);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	
Readable error messages	
Usable in library headers	
Code completion	
Implementing the DSL	
Only standard C++	
”Normal” C++	
No metaprogramming	
No build system support	

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	
Usable in library headers	
Code completion	
Implementing the DSL	
Only standard C++	
”Normal” C++	
No metaprogramming	
No build system support	

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	✓
Usable in library headers	
Code completion	
Implementing the DSL	
Only standard C++	
”Normal” C++	
No metaprogramming	
No build system support	

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	✓
Usable in library headers	✗
Code completion	
Implementing the DSL	
Only standard C++	
”Normal” C++	
No metaprogramming	
No build system support	

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	✓
Usable in library headers	✗
Code completion	✗
Implementing the DSL	
Only standard C++	
”Normal” C++	
No metaprogramming	
No build system support	

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	✓
Usable in library headers	✗
Code completion	✗
Implementing the DSL	
Only standard C++	✗
”Normal” C++	
No metaprogramming	
No build system support	

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	✓
Usable in library headers	✗
Code completion	✗
Implementing the DSL	
Only standard C++	✗
”Normal” C++	✓
No metaprogramming	
No build system support	

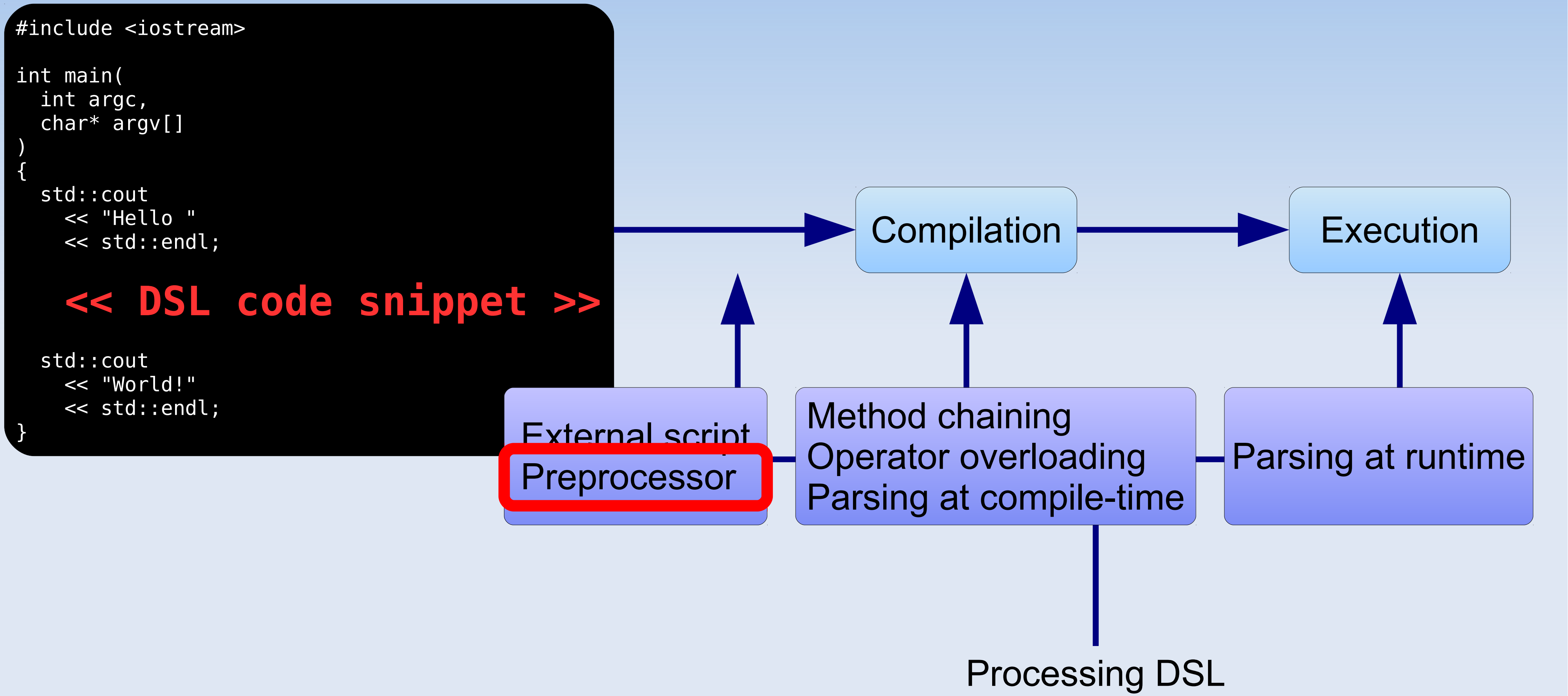
Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	✓
Usable in library headers	✗
Code completion	✗
Implementing the DSL	
Only standard C++	✗
”Normal” C++	✓
No metaprogramming	✓
No build system support	

Evaluation

	External script
Using the DSL	
No syntax changes	✓
Compile-time validation	✓
Readable error messages	✓
Usable in library headers	✗
Code completion	✗
Implementing the DSL	
Only standard C++	✗
”Normal” C++	✓
No metaprogramming	✓
No build system support	✗

Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class     E>   struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

```
auto re = DOT;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class     E> struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

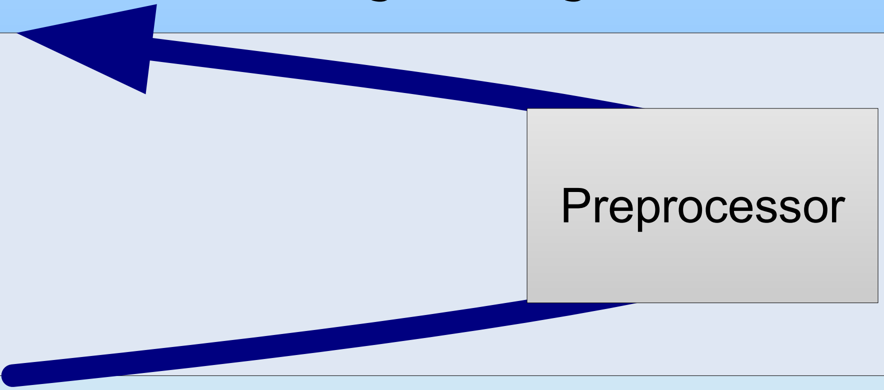
Preprocessor

```
auto re = DOT;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
auto re = any();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



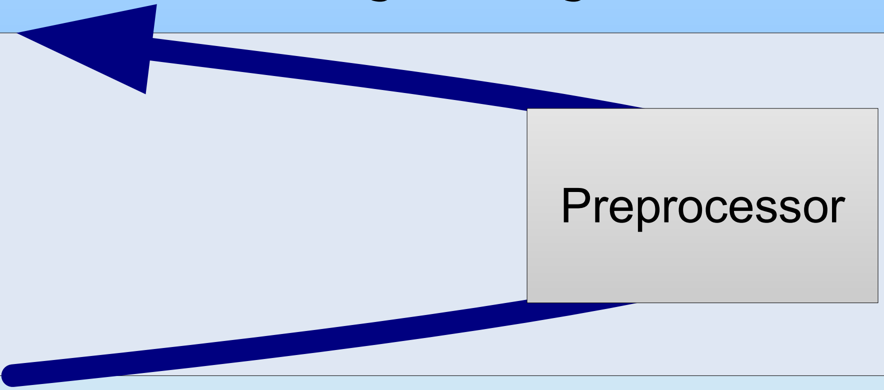
```
auto re = DOT;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.       abc
x       ab\
a*
```

```
auto re = any();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
#define DOT any()
```



Preprocessor

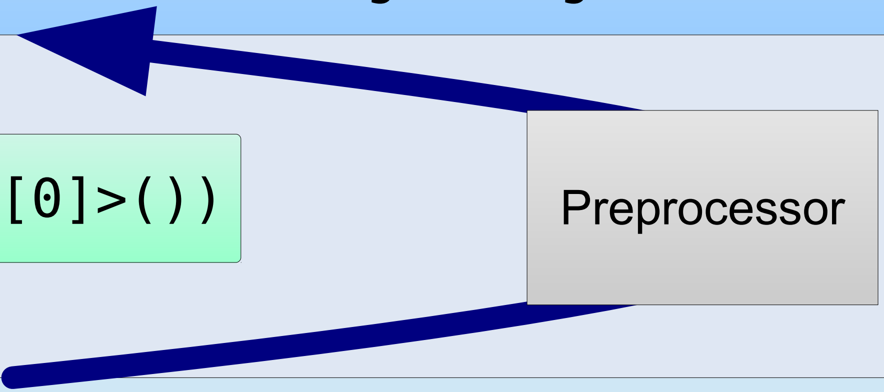
```
auto re = DOT;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = char_<'x'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
#define CHAR(c) (char_<#c[0]>())
```

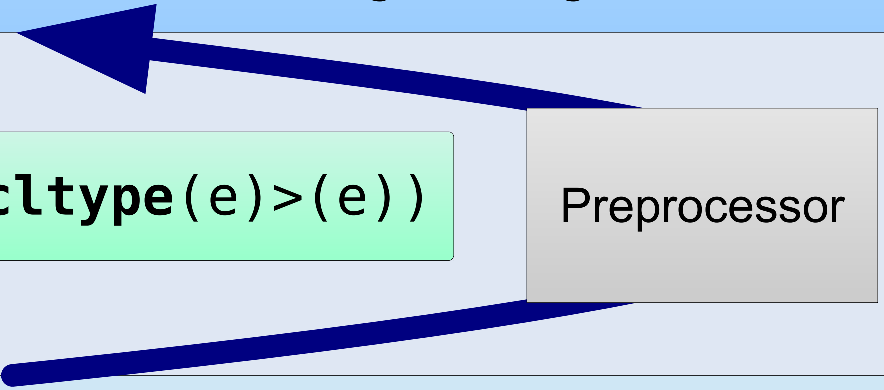


```
auto re = CHAR(x);

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = repeat<char_<'a'>>(char_<'a'>());

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



Preprocessor

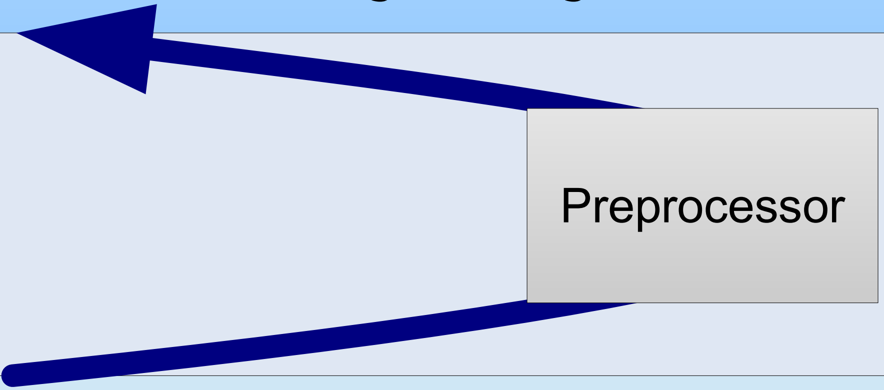
```
#define REPEAT(e) (repeat<decltype(e)>(e))
```

```
auto re = REPEAT(CHAR(a));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = seq<char_<'a'>, char_<'b'>, char_<'c'>>(
  char_<'a'>(), char_<'b'>(), char_<'c'>());
std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



Preprocessor

```
auto reg = SEQ(CHAR(a), CHAR(b), CHAR(c));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.       abc
x       ab\
a*
```

```
auto re = seq<char_<'a'>, char_<'b'>, char_<'c'>>(
  char_<'a'>(), char_<'b'>(), char_<'c'>());
std::string s("some text");
                                                            }
```

```
#define SEQ_ITEM(r, data, i, elem) BOOST_PP_COMMA_IF(i) decltype((elem))

#define SEQ(...) \
  ( \
    seq< \
      BOOST_PP_SEQ_FOR_EACH_I( \
        SEQ_ITEM, \
        ~, \
        BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__) \
      ) \
    >(__VA_ARGS__) \
  )
```

```
auto reg = SEQ(CHAR(a), CHAR(b), CHAR(c));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class    E>    struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

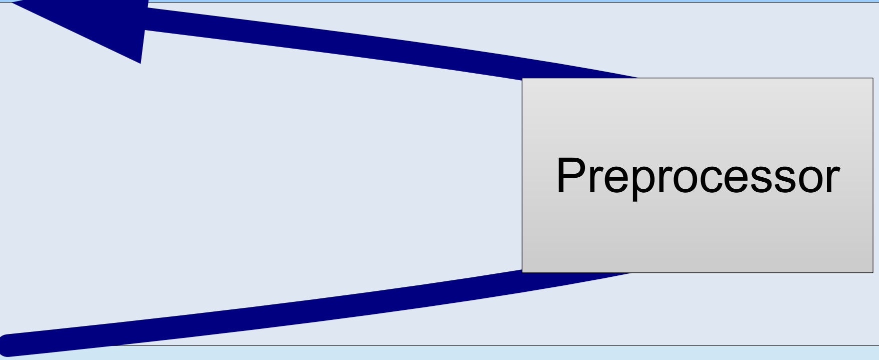
Preprocessor

```
auto re = SEQ(CHAR(a), CHAR(b), CHAR(\\));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = seq<char_<'a'>, char_<'b'>, char_<'\\'>>(
  char_<'a'>(), char_<'b'>(), char_<'\\'>());
std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



Preprocessor

```
auto re = SEQ(CHAR(a), CHAR(b), CHAR(\\));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

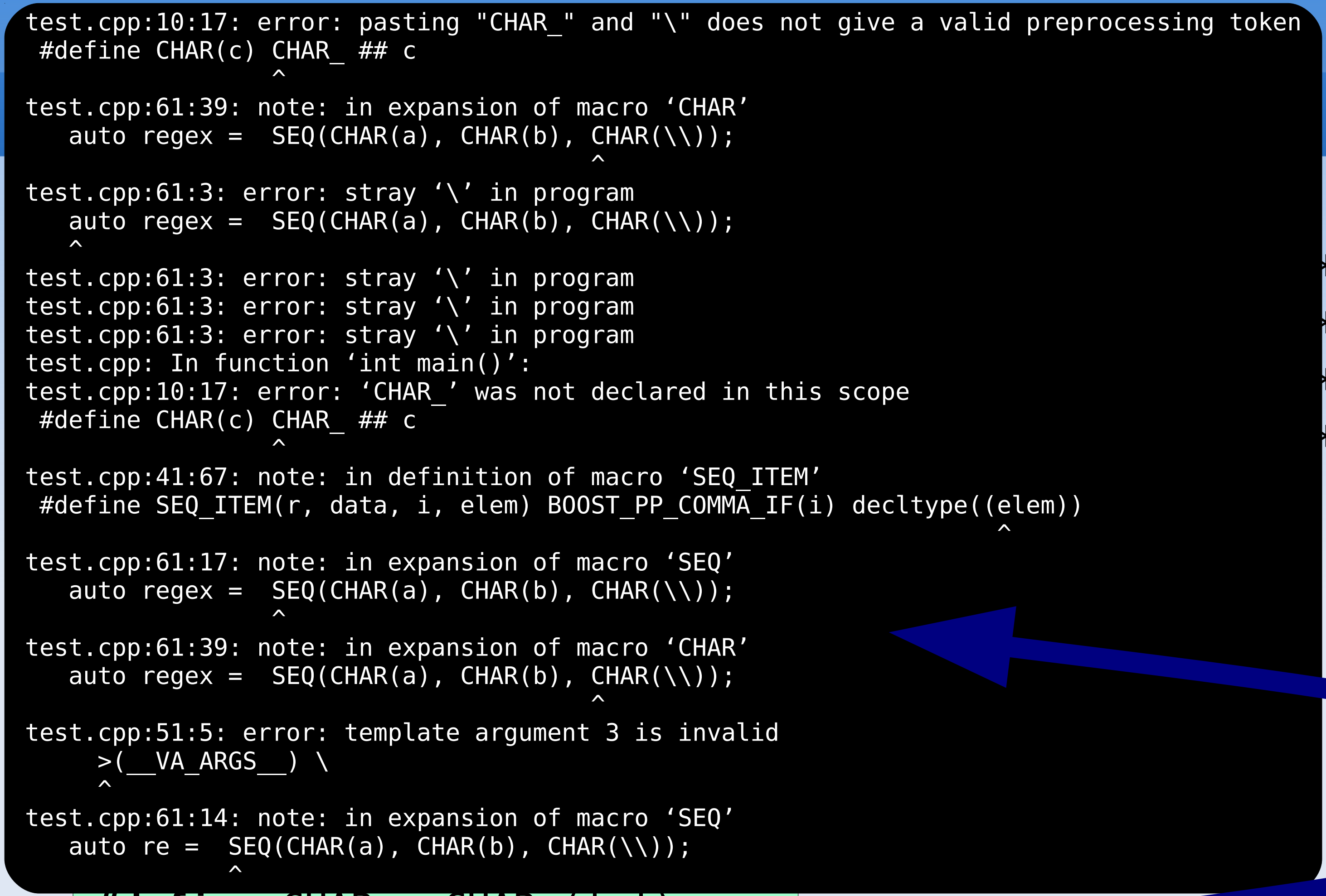
```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
//
```

```
#define CHAR(c) CHAR_ ## c
#define CHAR_(c) (char_<c>())
#define CHAR_a CHAR_('a')
#define CHAR_b CHAR_('b')

// ...

#define CHAR_y CHAR_('y')
#define CHAR_z CHAR_('z')
```

Preprocessor

```
auto re = SEQ(CHAR(a), CHAR(b), CHAR(\\));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
.      abc
x      ab\
a*
```



```
test.cpp:10:17: error: pasting "CHAR_" and "\" does not give a valid preprocessing token
 #define CHAR(c) CHAR_ ## c
                 ^
test.cpp:61:39: note: in expansion of macro ‘CHAR’
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
                                       ^
test.cpp:61:3: error: stray ‘\’ in program
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
   ^
test.cpp:61:3: error: stray ‘\’ in program
test.cpp:61:3: error: stray ‘\’ in program
test.cpp:61:3: error: stray ‘\’ in program
test.cpp: In function ‘int main()’:
test.cpp:10:17: error: ‘CHAR_’ was not declared in this scope
 #define CHAR(c) CHAR_ ## c
                 ^
test.cpp:41:67: note: in definition of macro ‘SEQ_ITEM’
 #define SEQ_ITEM(r, data, i, elem) BOOST_PP_COMMA_IF(i) decltype((elem))
                                                                   ^
test.cpp:61:17: note: in expansion of macro ‘SEQ’
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
                 ^
test.cpp:61:39: note: in expansion of macro ‘CHAR’
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
                                       ^
test.cpp:51:5: error: template argument 3 is invalid
     >(__VA_ARGS__) \
     ^
test.cpp:61:14: note: in expansion of macro ‘SEQ’
   auto re =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
              ^
```

```
*/ };
*/ };
*/ };
*/ };
```

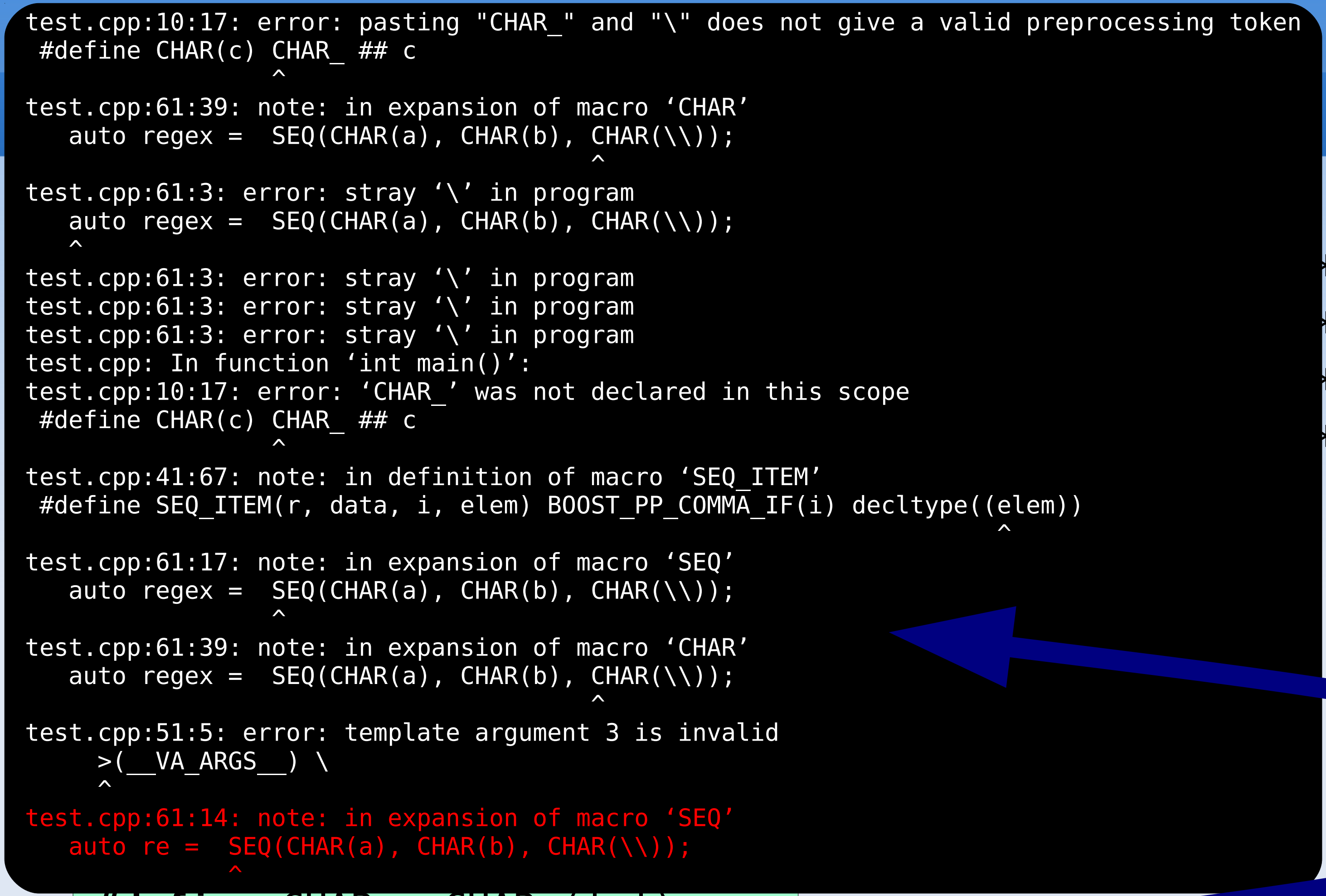
```
#define CHAR_z CHAR_('z')
```

Preprocessor

```
auto re = SEQ(CHAR(a), CHAR(b), CHAR(\\));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
.      abc
x      ab\
a*
```

```
*/ };
*/ };
*/ };
*/ };
```



```
test.cpp:10:17: error: pasting "CHAR_" and "\" does not give a valid preprocessing token
 #define CHAR(c) CHAR_ ## c
                 ^
test.cpp:61:39: note: in expansion of macro ‘CHAR’
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
                                       ^
test.cpp:61:3: error: stray ‘\’ in program
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
   ^
test.cpp:61:3: error: stray ‘\’ in program
test.cpp:61:3: error: stray ‘\’ in program
test.cpp:61:3: error: stray ‘\’ in program
test.cpp: In function ‘int main()’:
test.cpp:10:17: error: ‘CHAR_’ was not declared in this scope
 #define CHAR(c) CHAR_ ## c
                 ^
test.cpp:41:67: note: in definition of macro ‘SEQ_ITEM’
 #define SEQ_ITEM(r, data, i, elem) BOOST_PP_COMMA_IF(i) decltype((elem))
                                                                   ^
test.cpp:61:17: note: in expansion of macro ‘SEQ’
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
                 ^
test.cpp:61:39: note: in expansion of macro ‘CHAR’
   auto regex =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
                                       ^
test.cpp:51:5: error: template argument 3 is invalid
     >(__VA_ARGS__) \
     ^
test.cpp:61:14: note: in expansion of macro ‘SEQ’
   auto re =  SEQ(CHAR(a), CHAR(b), CHAR(\\));
              ^
```

Preprocessor

```
#define CHAR_z CHAR_('z')
```

```
auto re = SEQ(CHAR(a), CHAR(b), CHAR(\\));

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	
Readable error messages	✓	
Usable in library headers	✗	
Code completion	✗	
Implementing the DSL		
Only standard C++	✗	
”Normal” C++	✓	
No metaprogramming	✓	
No build system support	✗	

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	
Usable in library headers	✗	
Code completion	✗	
Implementing the DSL		
Only standard C++	✗	
”Normal” C++	✓	
No metaprogramming	✓	
No build system support	✗	

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	✗
Usable in library headers	✗	
Code completion	✗	
Implementing the DSL		
Only standard C++	✗	
”Normal” C++	✓	
No metaprogramming	✓	
No build system support	✗	

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	✗
Usable in library headers	✗	✓
Code completion	✗	
Implementing the DSL		
Only standard C++	✗	
”Normal” C++	✓	
No metaprogramming	✓	
No build system support	✗	

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	✗
Usable in library headers	✗	✓
Code completion	✗	✗
Implementing the DSL		
Only standard C++	✗	
”Normal” C++	✓	
No metaprogramming	✓	
No build system support	✗	

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	✗
Usable in library headers	✗	✓
Code completion	✗	✗
Implementing the DSL		
Only standard C++	✗	✓
”Normal” C++	✓	
No metaprogramming	✓	
No build system support	✗	

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	✗
Usable in library headers	✗	✓
Code completion	✗	✗
Implementing the DSL		
Only standard C++	✗	✓
”Normal” C++	✓	✗
No metaprogramming	✓	
No build system support	✗	

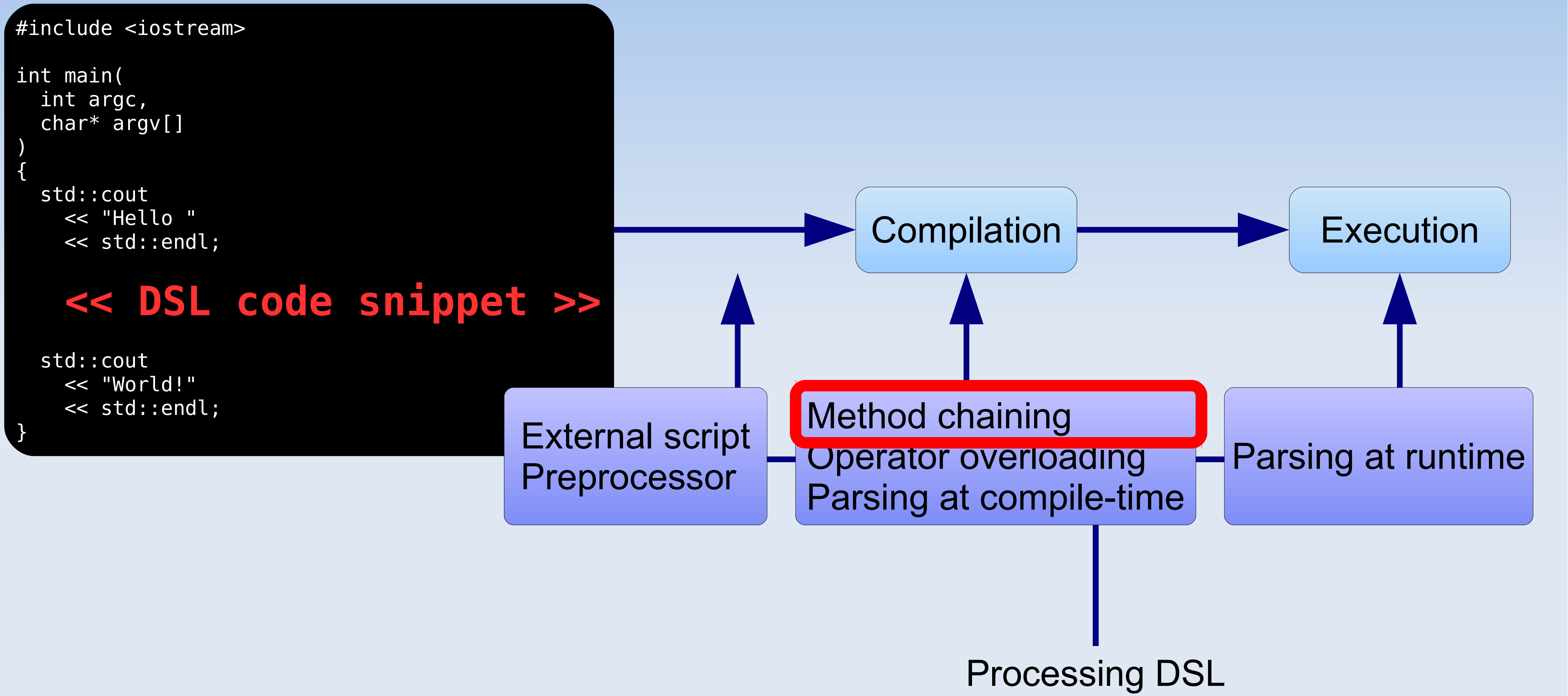
Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	✗
Usable in library headers	✗	✓
Code completion	✗	✗
Implementing the DSL		
Only standard C++	✗	✓
”Normal” C++	✓	✗
No metaprogramming	✓	✗
No build system support	✗	

Evaluation

	External script	Preprocessor
Using the DSL		
No syntax changes	✓	✗
Compile-time validation	✓	✓
Readable error messages	✓	✗
Usable in library headers	✗	✓
Code completion	✗	✗
Implementing the DSL		
Only standard C++	✗	✓
”Normal” C++	✓	✗
No metaprogramming	✓	✗
No build system support	✗	✓

Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Compilation

Execution

External script
Preprocessor

Method chaining
Operator overloading
Parsing at compile-time

Parsing at runtime

Processing DSL

Example

```
.       abc
x       ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {










};
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {









};
```

```
const regex_impl<empty> regex;
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  regex_impl(E e_) : _e(e_) {}









};
```

```
const regex_impl<empty> regex;
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  regex_impl(E e_) : _e(e_) {}

  template <class It>
  boost::optional<It> match(It begin_, It end_) const {

  }



};
```

```
const regex_impl<empty> regex;
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  regex_impl(E e_) : _e(e_) {}

  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    return _e.match(begin_, end_);
  }




};
```

```
const regex_impl<empty> regex;
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  regex_impl(E e_) : _e(e_) {}

  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    return _e.match(begin_, end_);
  }


  regex_impl<seq<E, any>> dot() const {                                  }

};
```

```
const regex_impl<empty> regex;
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  regex_impl(E e_) : _e(e_) {}

  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    return _e.match(begin_, end_);
  }


  regex_impl<seq<E, any>> dot() const { return seq<E, any>(_e, any()); }

};
```

```
const regex_impl<empty> regex;
```

```
auto re = regex.dot();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  // …

  template <char C>
  regex_impl<seq<E, char_<C>>> char_() const {

    return seq<E, ::char_<C>>(_e, ::char_<C>());
  }



};
```

```
auto re = regex.char_<'x'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  // …

  auto repeat() const -> decltype(repeat_last<E>::run(this->get())) {
    return repeat_last<E>::run(this->get());
  }





};
```

```
auto re = regex.char_<'a'>.repeat();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <class
class regex_imp
  E _e;
public:
  // …

  auto repeat()
    return repe
  }



};
```

```
template <int I, int N> struct set_nth {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<I - 1, N>::run(s_, nth_, s_.template get<I>(), as_...))
    { return set_nth<I - 1, N>::run(s_, nth_, s_.template get<I>(), as_...); }
};

template <int N> struct set_nth<N, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<N - 1, N>::run(s_, nth_, nth_, as_...))
    { return set_nth<N - 1, N>::run(s_, nth_, nth_, as_...); }
};

template <int N> struct set_nth<-1, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq&, NthT, As... as_)
    -> decltype(seq<As...>(as_...)) { return seq<As...>(as_...); }
};

template <class E> struct repeat_last {
  static repeat<E> run(E e_) { return repeat<E>(e_); }
};

template <class... Es> struct repeat_last<seq<Es...>> {
  static auto run(const seq<Es...>& s_) ->
    decltype(set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()))) {
    return set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()));
  }
};
```

```
auto re = reg

std::string s

if (auto i =
{ std::cout <
```

Example

```
.      abc
x      ab\
a*
```

```
template <class
class regex_imp
  E _e;
public:
  // …

  auto repeat()
    return repe
  }


};
```

```
auto re = reg

std::string s

if (auto i =
{ std::cout <
```

```
template <int I, int N> struct set_nth {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<I - 1, N>::run(s_, nth_, s_.template get<I>(), as_...))
    {                                          template get<I>(), as_...); }
};

template <int N> struct set_nth<N, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<N - 1, N>::run(s_, nth_, nth_, as_...))
    { return set_nth<N - 1, N>::run(s_, nth_, nth_, as_...); }
};

template <int N> struct set_nth<-1, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq&, NthT, As... as_)
    -> decltype(seq<As...>(as_...)) { return seq<As...>(as_...); }
};

template <class E> struct repeat_last {
  static repeat<E> run(E e_) { return repeat<E>(e_); }
};

template <class... Es> struct repeat_last<seq<Es...>> {
  static auto run(const seq<Es...>& s_) ->
    decltype(set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()))))) {
    return set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()));
  }
};
```

```
seq<e1, e2, …, e12, e13>
```

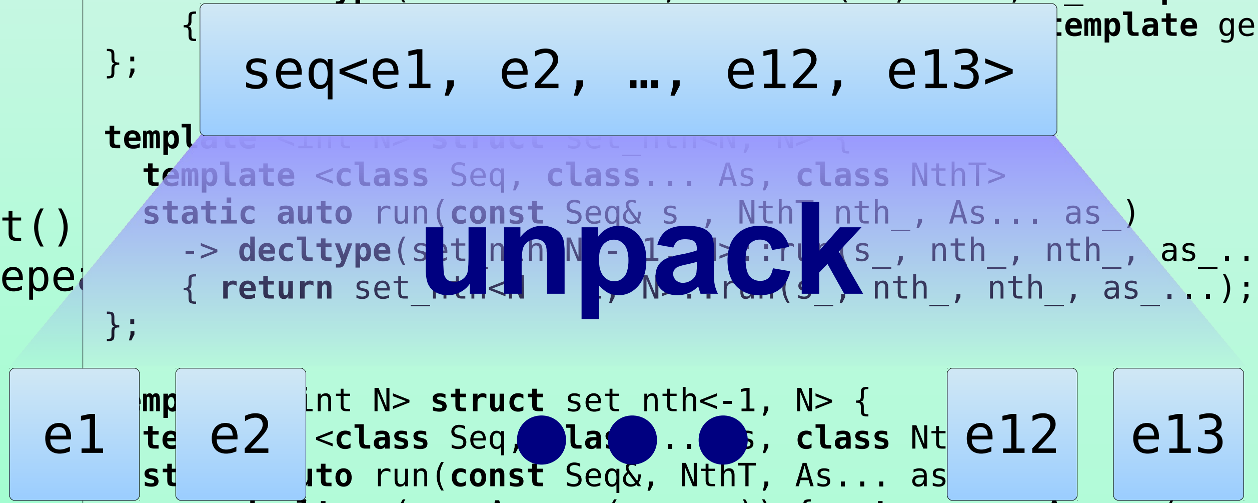
Example

```
.      abc
x      ab\
a*
```

```
template <class
class regex_imp
  E _e;
public:
  // …

  auto repeat()
    return repe
  }


};
```

```
auto re = reg

std::string s

if (auto i =
{ std::cout <
```

```
template <int I, int N> struct set_nth {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<I - 1, N>::run(s_, nth_, s_.template get<I>(), as_...))
    {                                                    template get<I>(), as_...); }
};

template <int N> struct set_nth<N, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<N - 1, N>::run(s_, nth_, nth_, as_...))
    { return set_nth<N - 1, N>::run(s_, nth_, nth_, as_...); }
};

template <int N> struct set_nth<-1, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq&, NthT, As... as_)
    -> decltype(seq<As...>(as_...)) { return seq<As...>(as_...); }
};

template <class E> struct repeat_last {
  static repeat<E> run(E e_) { return repeat<E>(e_); }
};

template <class... Es> struct repeat_last<seq<Es...>> {
  static auto run(const seq<Es...>& s_) ->
    decltype(set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()))) {
    return set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()));
  }
};
```



seq<e1, e2, …, e12, e13>

unpack

e1 e2 ● ● ● e12 e13

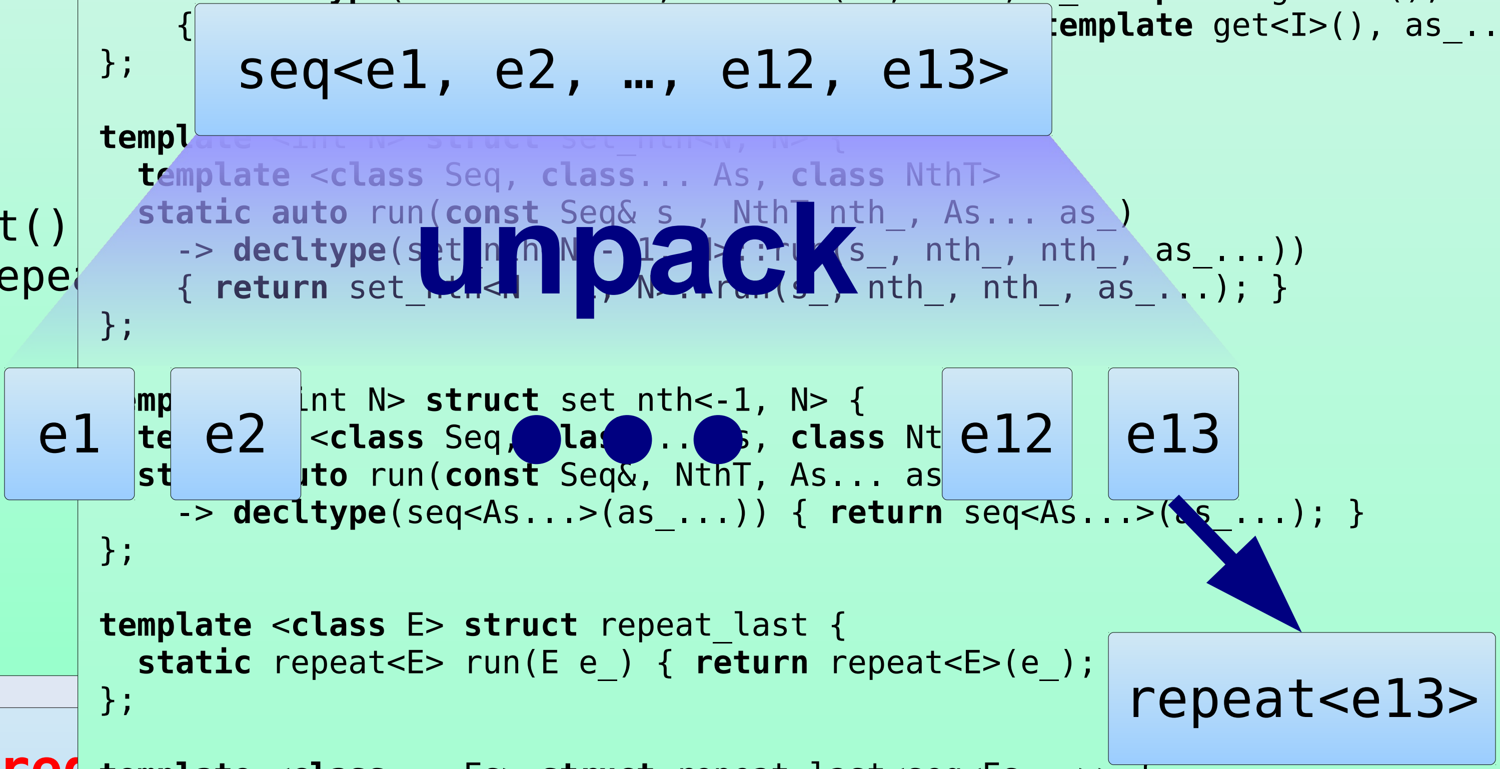
Example

```
.       abc
x       ab\
a*
```

```
template <class
class regex_imp
  E _e;
public:
  // …

  auto repeat()
    return repe
  }


};
```

```
auto re = reg

std::string s

if (auto i =
{ std::cout <
```

```
template <int I, int N> struct set_nth {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<I - 1, N>::run(s_, nth_, s_.template get<I>(), as_...))
    {                                              template get<I>(), as_...); }
};

template <int N> struct set_nth<N, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<N - 1, N>::run(s_, nth_, nth_, as_...))
    { return set_nth<N - 1, N>::run(s_, nth_, nth_, as_...); }
};

template <int N> struct set_nth<-1, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq&, NthT, As... as_)
    -> decltype(seq<As...>(as_...)) { return seq<As...>(as_...); }
};

template <class E> struct repeat_last {
  static repeat<E> run(E e_) { return repeat<E>(e_); }
};

template <class... Es> struct repeat_last<seq<Es...>> {
  static auto run(const seq<Es...>& s_) ->
    decltype(set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()))) {
    return set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()));
  }
};
```



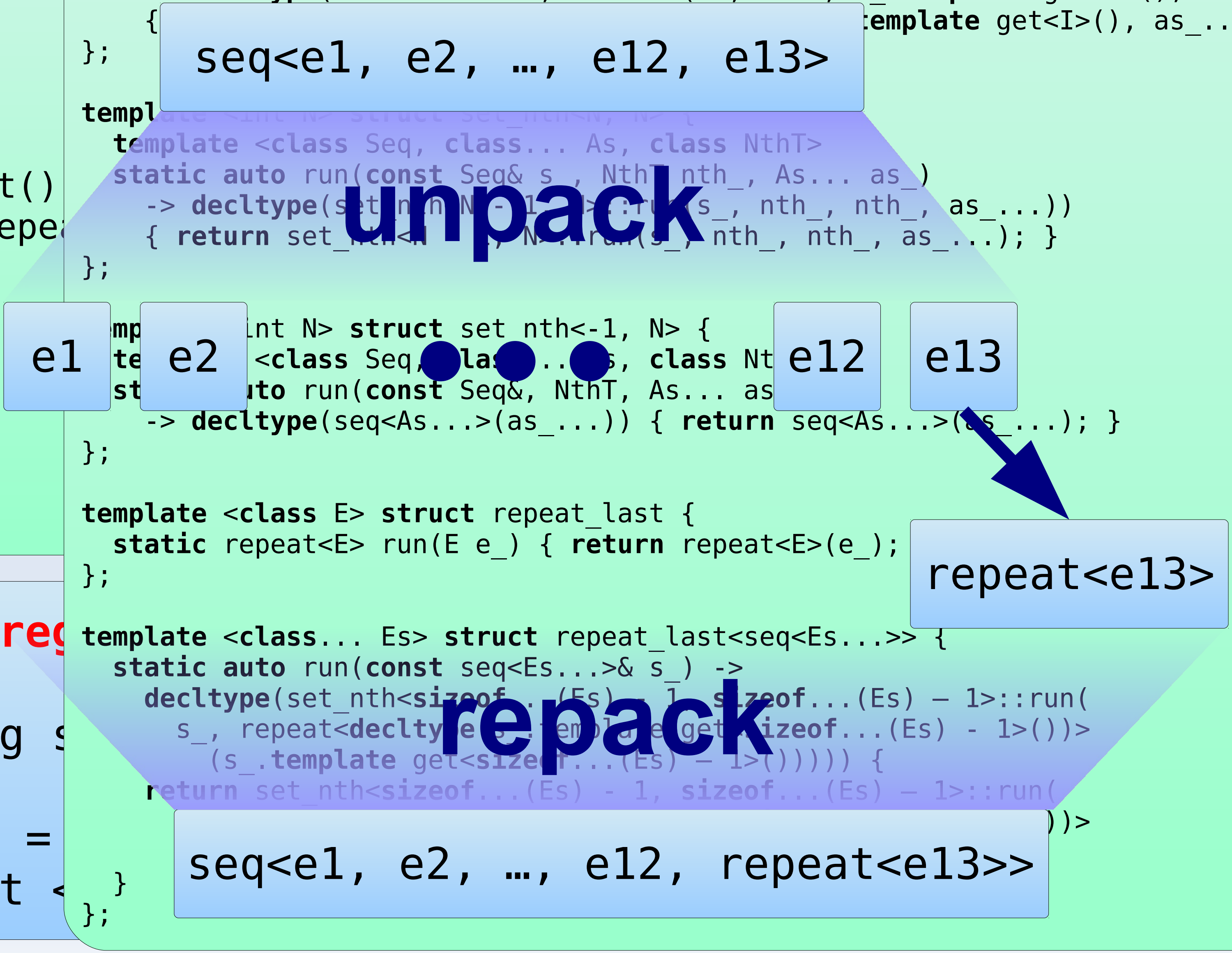
Example

```
.      abc
x      ab\
a*
```

```
template <class
class regex_imp
  E _e;
public:
  // …

  auto repeat()
    return repe
  }




};
```

```
auto re = reg

std::string s

if (auto i =
{ std::cout <
```

```
template <int I, int N> struct set_nth {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<I - 1, N>::run(s_, nth_, s_.template get<I>(), as_...))
    {                                                     template get<I>(), as_...); }
};

template <int N> struct set_nth<N, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq& s_, NthT nth_, As... as_)
    -> decltype(set_nth<N - 1, N>::run(s_, nth_, nth_, as_...))
    { return set_nth<N - 1, N>::run(s_, nth_, nth_, as_...); }
};

template <int N> struct set_nth<-1, N> {
  template <class Seq, class... As, class NthT>
  static auto run(const Seq&, NthT, As... as_)
    -> decltype(seq<As...>(as_...)) { return seq<As...>(as_...); }
};

template <class E> struct repeat_last {
  static repeat<E> run(E e_) { return repeat<E>(e_);
};

template <class... Es> struct repeat_last<seq<Es...>> {
  static auto run(const seq<Es...>& s_) ->
    decltype(set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
      s_, repeat<decltype(s_.template get<sizeof...(Es) - 1>())>
        (s_.template get<sizeof...(Es) – 1>()))) {
    return set_nth<sizeof...(Es) - 1, sizeof...(Es) – 1>::run(
                                                            ))>
  }
};
```



Example

```
.       abc
x       ab\
a*
```

```
template <char C>       struct char_  { /* … */ };
                        struct any    { /* … */ };
template <class    E>   struct repeat { /* … */ };
template <class... Es>  struct seq    { /* … */ };
// …
```

```
auto re = regex.char_<'a'>().char_<'b'>().char_<'c'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
auto re = regex.char_<'a'>().char_<'b'>().char_<'\\'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  // …

  template <char C>
  regex_impl<seq<E, char_<C>>> char_() const {

    return seq<E, ::char_<C>>(_e, ::char_<C>());
  }



};
```

```
auto re = regex.char_<'a'>().char_<'b'>().char_<'\\'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  // …

  template <char C>
  regex_impl<seq<E, char_<C>>> char_() const {
    static_assert(valid_char(C), "Invalid character");
    return seq<E, ::char_<C>>(_e, ::char_<C>());
  }



};
```

```
auto re = regex.char_<'a'>().char_<'b'>().char_<'\\'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {
  E _e;
public:
  // …

  template <char C>
  regex_impl<seq<E, char_<C>>> char_() const {
    static_assert(valid_char(C), "Invalid character");
    return seq<E, ::char_<C>>(_e, ::char_<C>());
  }


};
```

```
constexpr bool valid_char(char c_) { return c_ >= 'a' && c_ <= 'z'; }
```

```
auto re = regex.char_<'a'>().char_<'b'>().char_<'\\'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {
```

```
test.cpp: In instantiation of ‘regex_impl<seq<E, char_<C> > > regex_impl<E>::cha
r_() const [with char C = '\\'; E = seq<seq<empty, char_<'a'> >, char_<'b'> >]’:
test.cpp:110:57:   required from here
test.cpp:75:5: error: static assertion failed: Invalid character
     static_assert(valid_char(C), "Invalid character");
     ^
```

```
    static_assert(valid_char(C), "Invalid character");
    return seq<E, ::char_<C>>(_e, ::char_<C>());
  }

};
```

```
constexpr bool valid_char(char c_) { return c_ >= 'a' && c_ <= 'z'; }
```

```
auto re = regex.char_<'a'>().char_<'b'>().char_<'\\'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <class E>
class regex_impl {
```

```
test.cpp: In instantiation of ‘regex_impl<seq<E, char_<C> > > regex_impl<E>::cha
r_() const [with char C = '\\'; E = seq<seq<empty, char_<'a'> >, char_<'b'> >]’:
test.cpp:110:57:   required from here
test.cpp:75:5: error: static assertion failed: Invalid character
     static_assert(valid_char(C), "Invalid character");
     ^
```

```
    static_assert(valid_char(C), "Invalid character");
    return seq<E, ::char_<C>>(_e, ::char_<C>());
  }

};
```

```
constexpr bool valid_char(char c_) { return c_ >= 'a' && c_ <= 'z'; }
```

```
auto re = regex.char_<'a'>().char_<'b'>().char_<'\\'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	
Readable error messages	✓	✗	
Usable in library headers	✗	✓	
Code completion	✗	✗	
Implementing the DSL			
Only standard C++	✗	✓	
”Normal” C++	✓	✗	
No metaprogramming	✓	✗	
No build system support	✗	✓	

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	
Usable in library headers	✗	✓	
Code completion	✗	✗	
Implementing the DSL			
Only standard C++	✗	✓	
”Normal” C++	✓	✗	
No metaprogramming	✓	✗	
No build system support	✗	✓	

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	✗
Usable in library headers	✗	✓	
Code completion	✗	✗	
Implementing the DSL			
Only standard C++	✗	✓	
”Normal” C++	✓	✗	
No metaprogramming	✓	✗	
No build system support	✗	✓	

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	✗
Usable in library headers	✗	✓	✓
Code completion	✗	✗	
Implementing the DSL			
Only standard C++	✗	✓	
”Normal” C++	✓	✗	
No metaprogramming	✓	✗	
No build system support	✗	✓	

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	✗
Usable in library headers	✗	✓	✓
Code completion	✗	✗	✓
Implementing the DSL			
Only standard C++	✗	✓	
”Normal” C++	✓	✗	
No metaprogramming	✓	✗	
No build system support	✗	✓	

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	✗
Usable in library headers	✗	✓	✓
Code completion	✗	✗	✓
Implementing the DSL			
Only standard C++	✗	✓	✓
”Normal” C++	✓	✗	
No metaprogramming	✓	✗	
No build system support	✗	✓	

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	✗
Usable in library headers	✗	✓	✓
Code completion	✗	✗	✓
Implementing the DSL			
Only standard C++	✗	✓	✓
”Normal” C++	✓	✗	✗
No metaprogramming	✓	✗	
No build system support	✗	✓	

Evaluation

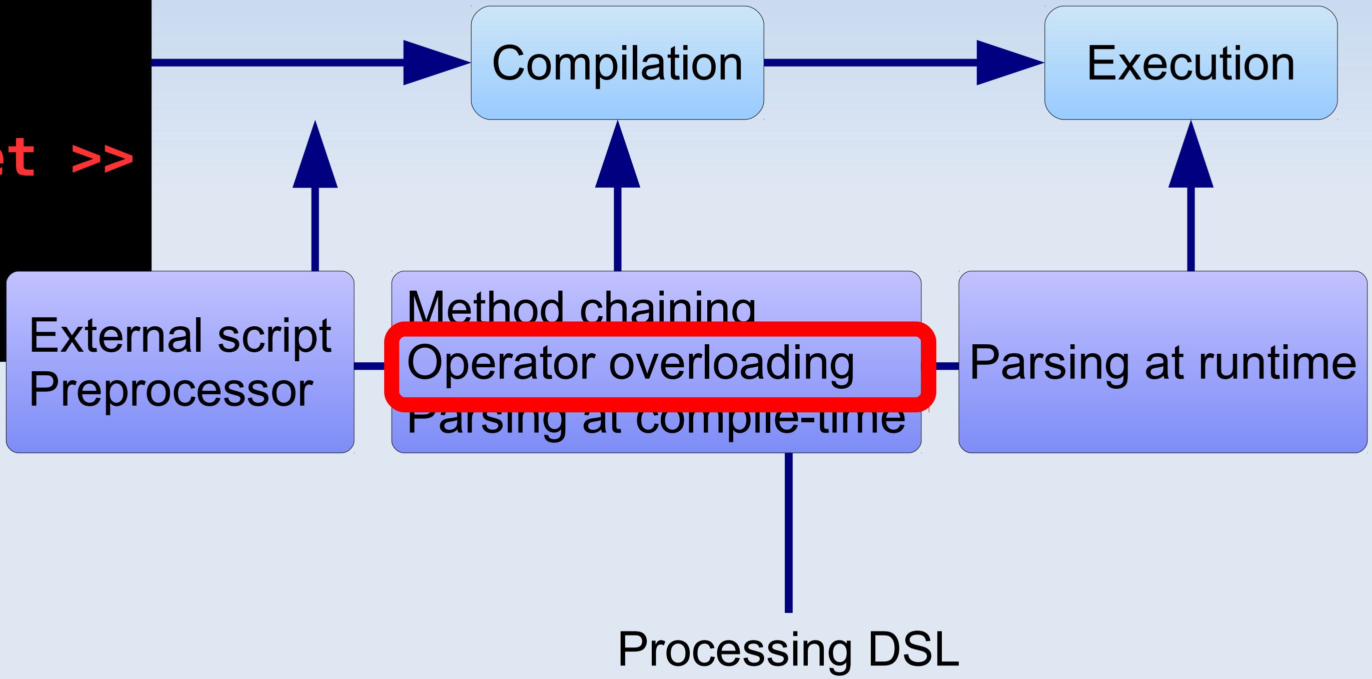
	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	✗
Usable in library headers	✗	✓	✓
Code completion	✗	✗	✓
Implementing the DSL			
Only standard C++	✗	✓	✓
”Normal” C++	✓	✗	✗
No metaprogramming	✓	✗	✗
No build system support	✗	✓	

Evaluation

	External script	Preprocessor	Method chaining
Using the DSL			
No syntax changes	✓	✗	✗
Compile-time validation	✓	✓	✓
Readable error messages	✓	✗	✗
Usable in library headers	✗	✓	✓
Code completion	✗	✗	✓
Implementing the DSL			
Only standard C++	✗	✓	✓
”Normal” C++	✓	✗	✗
No metaprogramming	✓	✗	✗
No build system support	✗	✓	✓

Embedding a DSL

```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```



Example

```
.      abc
x      ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>    struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

```
auto re = dot;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
const any dot;
```

```
auto re = dot;

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>         struct char_  { /* … */ };
                          struct any    { /* … */ };
template <class     E>    struct repeat { /* … */ };
template <class... Es>    struct seq    { /* … */ };
// …
```

```
auto re = ch<'x'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class    E>    struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

```
template <char C>
char_<C> ch() {

  return char_<C>();
}
```

```
auto re = ch<'x'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>    struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

```
template <char C>
char_<C> ch() {
  static_assert(valid_char(C), "Invalid character");
  return char_<C>();
}
```

```
auto re = ch<'x'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>   struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>          struct char_   { /* … */ };
                           struct any     { /* … */ };
template <class     E>   struct repeat { /* … */ };
template <class... Es>   struct seq      { /* … */ };
// …
```

```
template <class E>
                                                          repeat<E>
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>         struct char_   { /* … */ };
                          struct any     { /* … */ };
template <class     E>    struct repeat  { /* … */ };
template <class... Es>    struct seq     { /*   */ };
// …
```

```
std::vector<int> v;

*v
```

```
template <class E>
                                            repeat<E>
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>       struct char_  { /* … */ };
                        struct any    { /* … */ };
template <class    E>   struct repeat { /* … */ };
template <class... Es>  struct seq    { /* … */ };
// …
```

```
template <class E>
typename std::enable_if<                                  , repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>        struct char   { /*    */ };

te
te
//
```

```
template <class T> struct is_regex                                    ;
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>      struct char_     { /*    */ };

te
te
//
```

```
template <class T> struct is_regex              : std::false_type {};
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>         struct char_    { /*    */ };
te
te
//
```

```
template <class T> struct is_regex             : std::false_type {};

template <>        struct is_regex<any>        : std::true_type  {};
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>         struct char_    { /*    */ };

te
te
//
```

```
template <class T> struct is_regex              : std::false_type {};

template <>        struct is_regex<any>         : std::true_type  {};
template <char C>  struct is_regex<char_<C>>    : std::true_type  {};
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>        struct char_  { /*   */ };

te
te
//
```

```
template <class T> struct is_regex                 : std::false_type {};

template <>          struct is_regex<any>            : std::true_type  {};
template <char C>    struct is_regex<char_<C>>       : std::true_type  {};
template <class E> struct is_regex<repeat<E>> : is_regex<E>       {};
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>         struct char    { /*   */ };

te
te
//
```

```
template <class T> struct is_regex                   : std::false_type {};

template <>           struct is_regex<any>           : std::true_type  {};
template <char C>     struct is_regex<char_<C>>      : std::true_type  {};
template <class E> struct is_regex<repeat<E>> : is_regex<E>        {};
// …
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
.       abc
x       ab\
a*
```

Example

```
template <class E>
concept bool Regex() { return is_regex<E>::type::value; }
```

```
template <char C>          struct char_    { /*    */ };

te
te
//
```

```
template <class T> struct is_regex                : std::false_type {};

template <>        struct is_regex<any>           : std::true_type  {};
template <char C>  struct is_regex<char_<C>>      : std::true_type  {};
template <class E> struct is_regex<repeat<E>>     : is_regex<E>     {};
// …
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
auto re = *ch<'a'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
```

```
template <class E>
concept bool Regex() { return is_regex<E>::type::value; }
```

```
template <char C>      struct char_   { /*   */ };
```

```
te
te
//
```

```
template <class T> struct is_regex                : std::false_type {};

template <>        struct is_regex<any>           : std::true_type  {};
template <char C>  struct is_regex<char_<C>>      : std::true_type  {};
template <class E> struct is_regex<repeat<E>>     : is_regex<E>     {};
// …
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, repeat<E>>::type
operator*(E e_) { return repeat<E>(e_); }
```

```
template <Regex E>
repeat<E> operator*(E e_) { return repeat<E>(e_); }
```

```
au

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) ->

  seq<          E1           ,           E2           >
        {
  return seq<          E1           ,           E2           >
    (         e1_ ,          e2_ );
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>       struct char_  { /* … */ };
                        struct any    { /* … */ };
template <class     E>  struct repeat { /* … */ };
template <class... Es>  struct seq    { /* … */ };
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value && is_regex<E2>::type::value,
  seq<          E1           ,           E2           >
>::type {
  return seq<          E1           ,           E2           >
    (         e1_ ,           e2_ );
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value || is_regex<E2>::type::value,
  seq<          E1            ,           E2           >
>::type {
  return seq<          E1            ,           E2           >
    (         e1_ ,           e2_ );
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class    E>    struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value || is_regex<E2>::type::value,
  seq<          E1          ,          E2          >
>::type {
  return seq<          E1          ,          E2          >
    (to_regex(e1_), to_regex(e2_));
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
.      abc
x      ab\
a*
```

Example

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class     E>   struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value || is_regex<E2>::type::value,
  seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
>::type {
  return seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
    (to_regex(e1_), to_regex(e2_));
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>    struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value || is_regex<E2>::type::value,
  seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
>::type {
  return seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
    (to_regex(e1_), to_regex(e2_));
}
```

```
au


st


if
{                                                                          ; }
```

```
template <class E1, class E2> requires Regex<E1>() || Regex<E2>()
auto operator>>(E1 e1_, E2 e2_) ->
  seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
{
  return seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
    (to_regex(e1_), to_regex(e2_));
}
```

Example

.	abc
x	ab\

```
dyn_char to_regex(char c_) {
                        return dyn_char(c_);
}
```

```
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value || is_regex<E2>::type::value,
  seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
>::type {
  return seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
    (to_regex(e1_), to_regex(e2_));
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.       abc
x       ab\
```

```
template <class E>
                                                  E
to_regex(E e_) { return e_; }

dyn_char to_regex(char c_) {
                        return dyn_char(c_);

}
```

// …

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value || is_regex<E2>::type::value,
  seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
>::type {
  return seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
    (to_regex(e1_), to_regex(e2_));
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.       abc
x       ab\
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, E>::type
to_regex(E e_) { return e_; }

dyn_char to_regex(char c_) {
                      return dyn_char(c_);

}
```

```
// …
```

```
template <class E1, class E2>
auto operator>>(E1 e1_, E2 e2_) -> typename std::enable_if<
  is_regex<E1>::type::value || is_regex<E2>::type::value,
  seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
>::type {
  return seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
    (to_regex(e1_), to_regex(e2_));
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

.	abc
x	ab\

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, E>::type
to_regex(E e_) { return e_; }
```

```
template <Regex E>
E to_regex(E e_) { return e_; }
```

```
  seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
>::type {
  return seq<decltype(to_regex(e1_)), decltype(to_regex(e2_))>
    (to_regex(e1_), to_regex(e2_));
}
```

```
auto re = ch<'a'>() >> 'b' >> 'c';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

.	abc
x	ab\

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, E>::type
to_regex(E e_) { return e_; }

dyn_char to_regex(char c_) {
                      return dyn_char(c_);

}
```

```
// …
```

```
auto re = ch<'a'>() >> 'b' >> '\\';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

.	abc
x	ab\

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, E>::type
to_regex(E e_) { return e_; }

dyn_char to_regex(char c_) {
  if (valid_char(c_)) { return dyn_char(c_); }
   else { throw regex_error(std::string("Invalid character ") + c_); }
}
```

```
// …
```

```
auto re = ch<'a'>() >> 'b' >> '\\';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
```

```
template <class E>
typename std::enable_if<is_regex<E>::type::value, E>::type
to_regex(E e_) { return e_; }

dyn_char to_regex(char c_) {
  if (valid_char(c_)) { return dyn_char(c_); }
   else { throw regex_error(std::string("Invalid character ") + c_); }
}
```

```
// …
```

```
terminate called after throwing an instance of 'regex_error'
  what():  Invalid character \
```

```
auto re = ch<'a'>() >> 'b' >> '\\';

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	
Readable error messages	✓	✗	✗	
Usable in library headers	✗	✓	✓	
Code completion	✗	✗	✓	
Implementing the DSL				
Only standard C++	✗	✓	✓	
”Normal” C++	✓	✗	✗	
No metaprogramming	✓	✗	✗	
No build system support	✗	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	
Usable in library headers	✗	✓	✓	
Code completion	✗	✗	✓	
Implementing the DSL				
Only standard C++	✗	✓	✓	
”Normal” C++	✓	✗	✗	
No metaprogramming	✓	✗	✗	
No build system support	✗	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	✓
Usable in library headers	✗	✓	✓	
Code completion	✗	✗	✓	
Implementing the DSL				
Only standard C++	✗	✓	✓	
”Normal” C++	✓	✗	✗	
No metaprogramming	✓	✗	✗	
No build system support	✗	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	✓
Usable in library headers	✗	✓	✓	✓
Code completion	✗	✗	✓	
Implementing the DSL				
Only standard C++	✗	✓	✓	
”Normal” C++	✓	✗	✗	
No metaprogramming	✓	✗	✗	
No build system support	✗	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	✓
Usable in library headers	✗	✓	✓	✓
Code completion	✗	✗	✓	✗
Implementing the DSL				
Only standard C++	✗	✓	✓	
”Normal” C++	✓	✗	✗	
No metaprogramming	✓	✗	✗	
No build system support	✗	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	✓
Usable in library headers	✗	✓	✓	✓
Code completion	✗	✗	✓	✗
Implementing the DSL				
Only standard C++	✗	✓	✓	✓
”Normal” C++	✓	✗	✗	
No metaprogramming	✓	✗	✗	
No build system support	✗	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	✓
Usable in library headers	✗	✓	✓	✓
Code completion	✗	✗	✓	✗
Implementing the DSL				
Only standard C++	✗	✓	✓	✓
”Normal” C++	✓	✗	✗	✗
No metaprogramming	✓	✗	✗	
No build system support	✗	✓	✓	

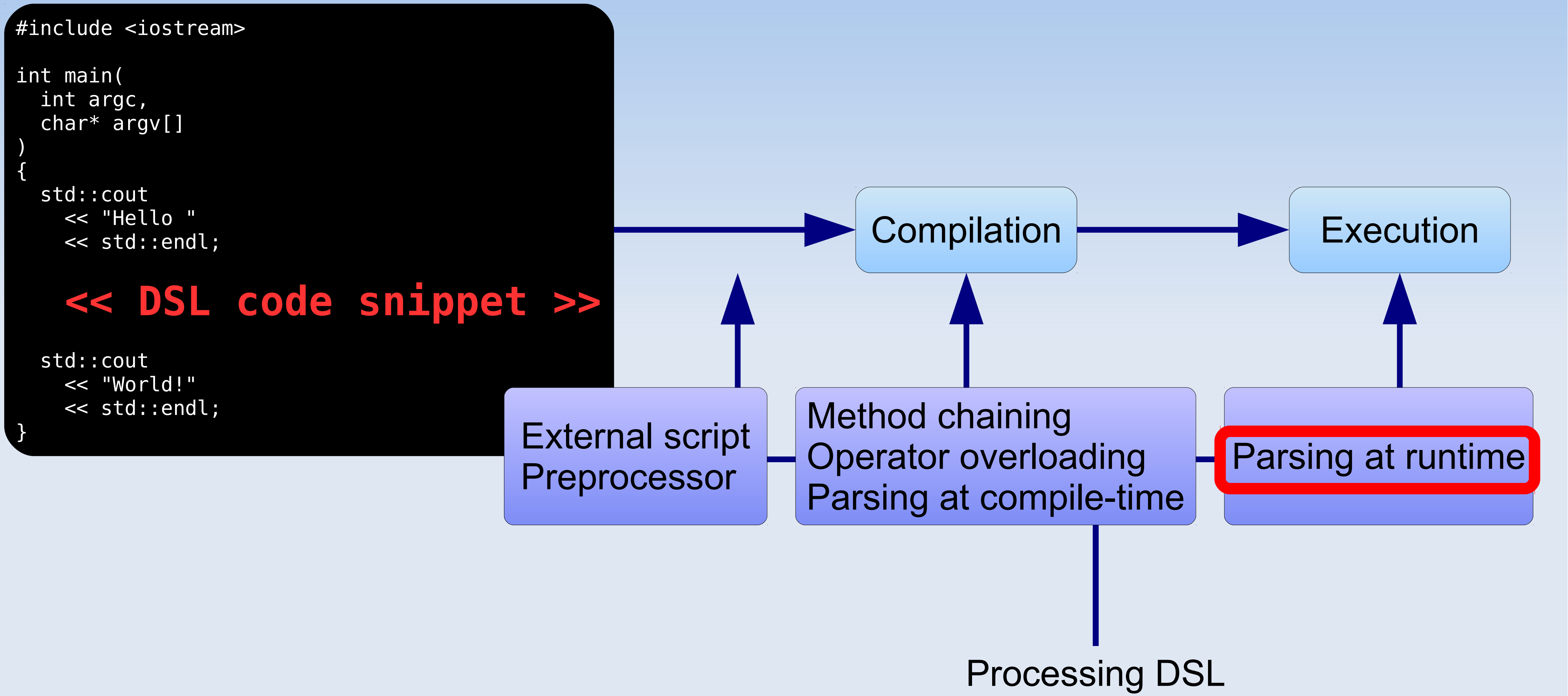
Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	✓
Usable in library headers	✗	✓	✓	✓
Code completion	✗	✗	✓	✗
Implementing the DSL				
Only standard C++	✗	✓	✓	✓
”Normal” C++	✓	✗	✗	✗
No metaprogramming	✓	✗	✗	✗
No build system support	✗	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading
Using the DSL				
No syntax changes	✓	✗	✗	✗
Compile-time validation	✓	✓	✓	✗
Readable error messages	✓	✗	✗	✓
Usable in library headers	✗	✓	✓	✓
Code completion	✗	✗	✓	✗
Implementing the DSL				
Only standard C++	✗	✓	✓	✓
”Normal” C++	✓	✗	✗	✗
No metaprogramming	✓	✗	✗	✗
No build system support	✗	✓	✓	✓

Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Compilation

Execution

External script
Preprocessor

Method chaining
Operator overloading
Parsing at compile-time

Parsing at runtime

Processing DSL

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class    E>    struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

```
regex re(".");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>         struct char_  { /* … */ };
                          struct any    { /* … */ };
template <class    E>     struct repeat { /* … */ };
template <class... Es>    struct seq    { /* … */ };
// …
```

```
regex parse(const std::string& re_) {
  // …
}
```

```
regex re(".");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
class regex_interface {
public:
  virtual ~regex_interface() {}

  virtual boost::optional<const char*> match(
    const char* begin_,
    const char* end_
  ) const = 0;

  virtual regex_interface* clone() const = 0;
};
```

```
{ /* … */ };
{ /* … */ };
{ /* … */ };
{ /* … */ };
```

```
regex re(".");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
class regex_interface {
public:
  virtual ~regex_interface() {}

  virtual boost::optional<const char*> match(
    const char* begin_,
    const char* end_
  ) const = 0;

  virtual regex_interface* clone() const = 0;
};
```

```
{ /* … */ };
{ /* … */ };
t { /* … */ };
{ /* … */ };
```

```
template <class E> class regex_impl :
  public regex_interface {



};
```

```
regex re(".");

std::string s("some text

if (auto i = re.match(s.
{ std::cout << "matched:     << std::string(s.begin(), i); }
```

Example

```
.       abc
x       ab\
a*
```

```
class regex_interface {
public:
  virtual ~regex_interface() {}

  virtual boost::optional<const char*> match(
    const char* begin_,
    const char* end_
  ) const = 0;

  virtual regex_interface* clone() const = 0;
};
```

```
{ /* … */ };
{ /* … */ };
t { /* … */ };
{ /* … */ };
```

```
template <class E> class regex_impl :
  public regex_interface {
public:
  regex_impl(E e_) : _e(e_) {}

private:
  E _e;
};
```

```
regex re(".");

std::string s("some text

if (auto i = re.match(s.
{ std::cout << "matched: " << std::string(s.begin(), i); }
```

Example

```
.      abc
x      ab\
a*
```

```
class regex_interface {
public:
  virtual ~regex_interface() {}

  virtual boost::optional<const char*> match(
    const char* begin_,
    const char* end_
  ) const = 0;

  virtual regex_interface* clone() const = 0;
};
```

```
{ /* … */ };
{ /* … */ };
t { /* … */ };
{ /* … */ };
```

```
template <class E> class regex_impl :
  public regex_interface {
public:
  regex_impl(E e_) : _e(e_) {}

  virtual boost::optional<const char*> match(
    const char* begin_,
    const char* end_
  ) const { return _e.match(begin_, end_); }

  virtual regex_interface* clone() const
    { return new regex_impl(*this); }
private:
  E _e;
};
```

```
regex re(".");

std::string s("some text

if (auto i = re.match(s.
{ std::cout << "matched:      << std::string(s.begin(), i); }
```

Example

```
.      abc
x      ab\
a*
```

```
class regex_interface {
public:
  virtual ~regex_interface() {}
}
```

```
class regex {
public:
  template <class E> regex(E e_) : _body(new regex_impl<E>(e_)) {}
  regex(const regex& e_) : _body(e_._body->clone()) {}
  regex& operator=(re e_) { swap(e_); return *this; }

  template <class It>
  boost::optional<It> match(It begin_, It end_) const {
    const std::string s(begin_, end_);
    if (auto i = _body->match(s.c_str(), s.c_str() + s.length()))
    {
      std::advance(begin_, *i - s.c_str());
      return begin_;
    } else { return boost::optional<It>(); }
  }

  void swap(re& e_) { _body.swap(e_._body); }

private:
  std::unique_ptr<regex_interface> _body;
};
```

```
> match(
nd_); }
nst
private:
  E _e;
};
```

```
if (auto i = re.match(s.
{ std::cout << "matched:                  << std::string(s.begin(), i)); }
```

Example

```
.     abc
x     ab\
a*
```

```
void append_char(regex& r_, char c_) {r_=seq<regex,regex>(r_, dyn_char(c_));}
void append_any(regex& r_) { r_ = seq<regex, regex>(r_, any()); }

void repeat_last(regex& r_) {
  const auto& s = r_.get<seq<regex, regex>>();
  r_ = seq<regex, regex>(s.get<0>(), repeat<regex>(s.get<1>()));
}

regex parse(const std::string& e_) {
  using boost::spirit::qi::char_;

  regex r{empty()};

  auto a_char = boost::bind(append_char, boost::ref(r), _1);
  auto a_any  = boost::bind(append_any,  boost::ref(r));
  auto rep    = boost::bind(repeat_last, boost::ref(r));

  std::string::const_iterator i = e_.begin();
  if (boost::spirit::qi::parse(i, e_.end(),
      *((char_('.')[a_any] | char_("a-z")[a_char]) >> -char_('*')[rep])))
  { if (i == e_.end()) { return r; } else { throw regex_error(/* … */); } }
  else { throw regex_error(/* … */); }
}
```

```
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
void append_char(regex& r_, char c_) {r_=seq<regex,regex>(r_, dyn_char(c_));}
void append_any(regex& r_) { r_ = seq<regex, regex>(r_, any()); }

void repeat_last(regex& r_) {
  const auto& s = r_.get<seq<regex, regex>>();
  r_ = seq<regex, regex>(s.get<0>(), repeat<regex>(s.get<1>()));
}

regex parse(const std::string& e_) {
  using boost::spirit::qi::char_;

  regex r{empty()};

  auto a_char = boost::bind(append_char, boost::ref(r), _1);
  auto a_any  = boost::bind(append_any,  boost::ref(r));
  auto rep    = boost::bind(repeat_last, boost::ref(r));

  std::string::const_iterator i = e_.begin();
  if (boost::spirit::qi::parse(i, e_.end(),
      *((char_('.')[a_any] | char_("a-z")[a_char]) >> -char_('*')[rep])))
  { if (i == e_.end()) { return r; } else { throw regex_error(/* … */); } }
  else { throw regex_error(/* … */); }
}
```

```
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
void append_char(regex& r_, char c_) {r_=seq<regex,regex>(r_, dyn_char(c_));}
void append_any(regex& r_) { r_ = seq<regex, regex>(r_, any()); }

void repeat_last(regex& r_) {
  const auto& s = r_.get<seq<regex, regex>>();
  r_ = seq<regex, regex>(s.get<0>(), repeat<regex>(s.get<1>()));
}

regex parse(const std::string& e_) {
  using boost::spirit::qi::char_;

  regex r{empty()};

  auto a_char = boost::bind(append_char, boost::ref(r), _1);
  auto a_any  = boost::bind(append_any,  boost::ref(r));
  auto rep    = boost::bind(repeat_last, boost::ref(r));

  std::string::const_iterator i = e_.begin();
  if (boost::spirit::qi::parse(i, e_.end(),
      *((char_('.')[a_any] | char_("a-z")[a_char]) >> -char_('*')[rep])))
  { if (i == e_.end()) { return r; } else { throw regex_error(/* … */); } }
  else { throw regex_error(/* … */); }
}
```

```
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.       abc
x       ab\
a*
```

```
void append_char(regex& r_, char c_) {r_=seq<regex,regex>(r_, dyn_char(c_));}
void append_any(regex& r_) { r_ = seq<regex, regex>(r_, any()); }

void repeat_last(regex& r_) {
  const auto& s = r_.get<seq<regex, regex>>();
  r_ = seq<regex, regex>(s.get<0>(), repeat<regex>(s.get<1>()));
}

regex parse(const std::string& e_) {
  using boost::spirit::qi::char_;

  regex r{empty()};

  auto a_char = boost::bind(append_char, boost::ref(r), _1);
  auto a_any  = boost::bind(append_any,  boost::ref(r));
  auto rep    = boost::bind(repeat_last, boo

  std::string::const_iterator i = e_.begin();
  if (boost::spirit::qi::parse(i, e_.end(),
      *((char_('.')[a_any] | char_("a-z")[a_char]) >> -char_('*')[rep])))
  { if (i == e_.end()) { return r; } else { throw regex_error(/* … */); } }
  else { throw regex_error(/* … */); }
}
```

```
(('.' | 'a'..'z') '*'?)*
```

```
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class    E>    struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

```
regex re(".");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_   { /* … */ };
                         struct any     { /* … */ };
template <class    E>    struct repeat  { /* … */ };
template <class... Es>   struct seq     { /* … */ };
// …
```

```
regex re("x");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>      struct char_  { /* … */ };
                       struct any    { /* … */ };
template <class    E>  struct repeat { /* … */ };
template <class... Es> struct seq    { /* … */ };
// …
```

```
regex re("a*");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
template <char C>       struct char_  { /* … */ };
                        struct any    { /* … */ };
template <class    E>   struct repeat { /* … */ };
template <class... Es>  struct seq    { /* … */ };
// …
```

```
regex re("abc");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>    struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

```
terminate called after throwing an instance of 'regex_error'
  what():  Invalid regular expression (char 3) ab1
```

```
regex re("ab\\");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	
Readable error messages	✓	✗	✗	✓	
Usable in library headers	✗	✓	✓	✓	
Code completion	✗	✗	✓	✗	
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	
”Normal” C++	✓	✗	✗	✗	
No metaprogramming	✓	✗	✗	✗	
No build system support	✗	✓	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	
Usable in library headers	✗	✓	✓	✓	
Code completion	✗	✗	✓	✗	
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	
”Normal” C++	✓	✗	✗	✗	
No metaprogramming	✓	✗	✗	✗	
No build system support	✗	✓	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	✓
Usable in library headers	✗	✓	✓	✓	
Code completion	✗	✗	✓	✗	
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	
”Normal” C++	✓	✗	✗	✗	
No metaprogramming	✓	✗	✗	✗	
No build system support	✗	✓	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	✓
Usable in library headers	✗	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	
”Normal” C++	✓	✗	✗	✗	
No metaprogramming	✓	✗	✗	✗	
No build system support	✗	✓	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	✓
Usable in library headers	✗	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	
”Normal” C++	✓	✗	✗	✗	
No metaprogramming	✓	✗	✗	✗	
No build system support	✗	✓	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	✓
Usable in library headers	✗	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗	
No metaprogramming	✓	✗	✗	✗	
No build system support	✗	✓	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	✓
Usable in library headers	✗	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗	✓
No metaprogramming	✓	✗	✗	✗	
No build system support	✗	✓	✓	✓	

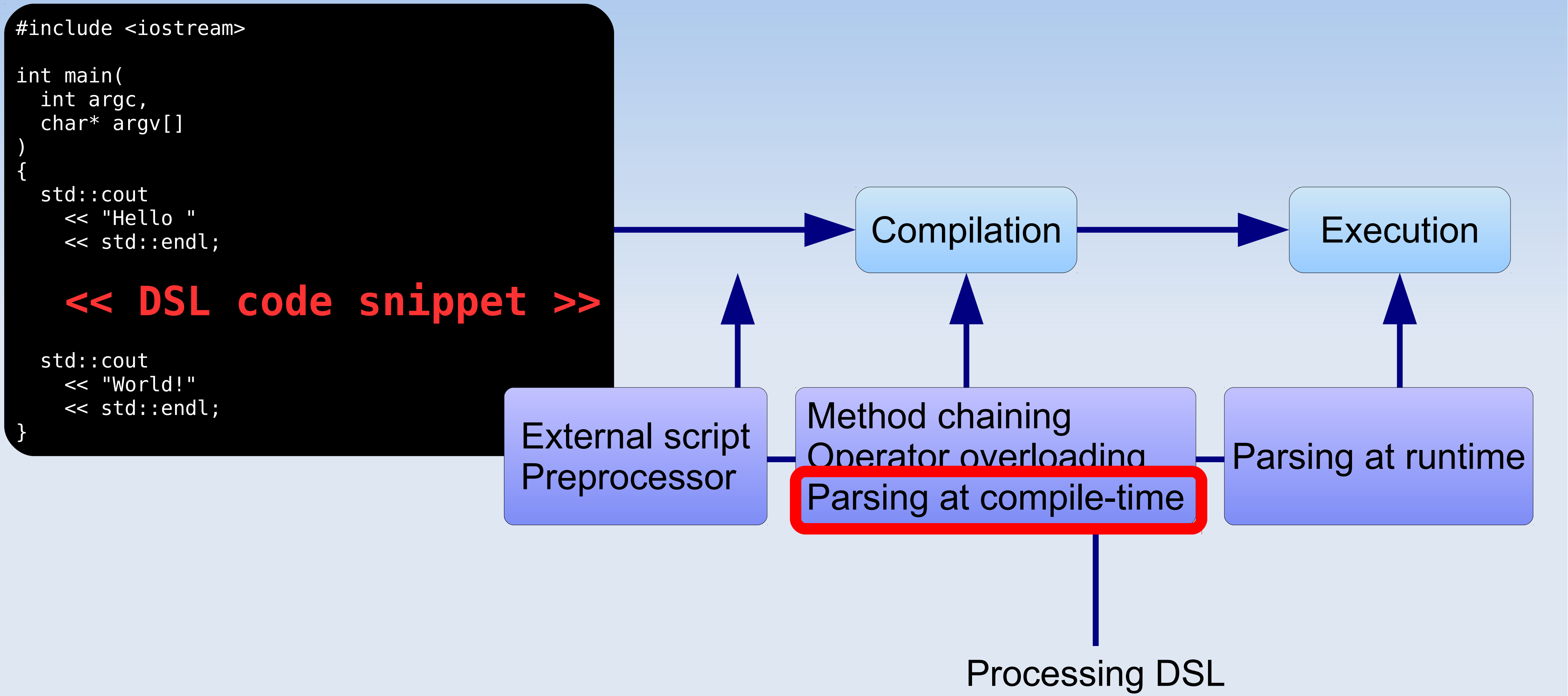
Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	✓
Usable in library headers	✗	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗	✓
No metaprogramming	✓	✗	✗	✗	✓
No build system support	✗	✓	✓	✓	

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at runtime
Using the DSL					
No syntax changes	✓	✗	✗	✗	✓
Compile-time validation	✓	✓	✓	✗	✗
Readable error messages	✓	✗	✗	✓	✓
Usable in library headers	✗	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗
Implementing the DSL					
Only standard C++	✗	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗	✓
No metaprogramming	✓	✗	✗	✗	✓
No build system support	✗	✓	✓	✓	✓

Embedding a DSL



```
#include <iostream>

int main(
  int argc,
  char* argv[]
)
{
  std::cout
    << "Hello "
    << std::endl;

    << DSL code snippet >>

  std::cout
    << "World!"
    << std::endl;
}
```

Compilation

Execution

External script
Preprocessor

Method chaining
Operator overloading
Parsing at compile-time

Parsing at runtime

Processing DSL

```
.     abc
x     ab\
a*
```

Example

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>    struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

```
auto re = regex<MPLLIBS_STRING(".")>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
template <char C>        struct char_  { /* … */ };
                         struct any    { /* … */ };
template <class    E>    struct repeat { /* … */ };
template <class... Es>   struct seq    { /* … */ };
// …
```

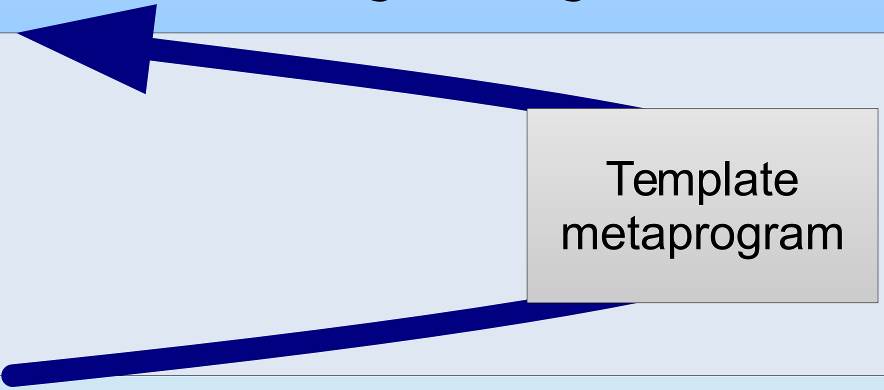
Template metaprogram

```
auto re = regex<MPLLIBS_STRING(".")>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = any();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(), i); }
```

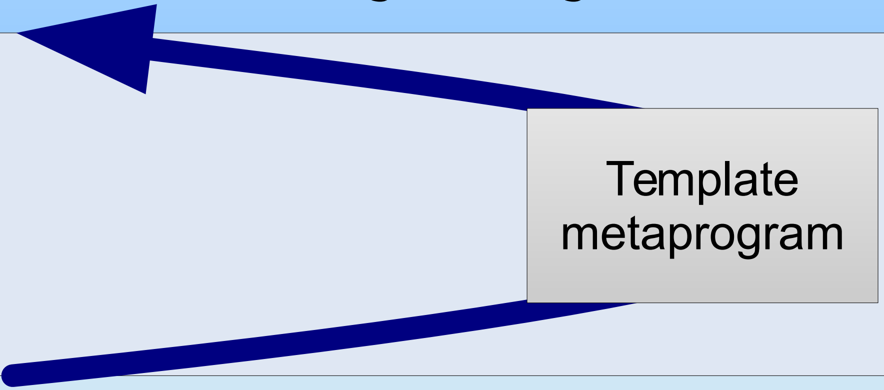


```
auto re = regex<MPLLIBS_STRING(".")>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = any();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(), i); }
```



Template metaprogram

```
auto re = REGEX(".");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
.     abc
x     ab\
a*
```

Example

```
auto re = any();

std::string s("some text");

in(), i); }
```

> *”How would you know that you have gone too far with metaprogramming? One warning sign that I use is an urge to use macros to hide ”details” that have become too ugly to deal with directly.”*
>
> Bjarne Stroustrup, The C++ programming language, Fourth edition

Template
etaprogram

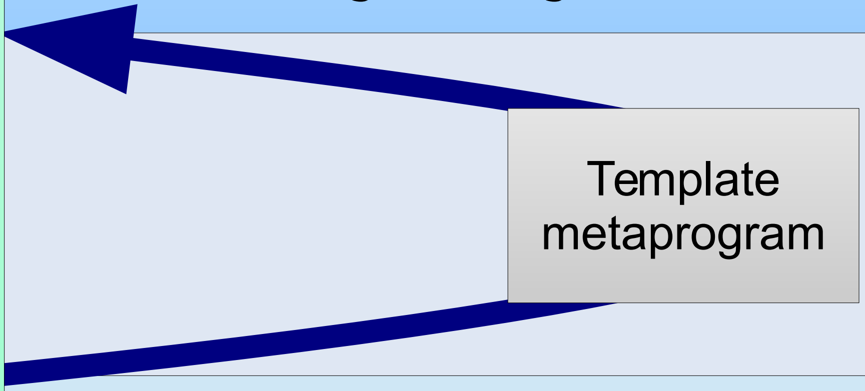
```
auto re = REGEX(".");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
struct invalid_regex_char {
  typedef invalid_regex_char type
  static std::string get_value() { return std::string("Invalid regex char "); }
};

template <class T> struct returns { typedef T type; };

template <class E> struct default_construct : returns<default_construct<E>> {
  template <class> struct apply : returns<default_construct<E>> {};
  static auto run() RETURNS( E() )
};

template <class E, char C> struct build_repeated_impl;
template <class E> struct build_repeated_impl<E, '*'> :
  returns<build_repeated_impl<E, '*'>>
{ static auto run() RETURNS( repeat<decltype(E::run())>(E::run()) ) };

template <class E> struct build_repeated_impl<E, 'x'> : E {};

struct build_seq {
  template <class A, class B> struct apply : returns<apply<A, B>> {
    static auto run()
    RETURNS(seq<decltype(B::run()), decltype(A::run())>(B::run(), A::run()))
  };
};

struct build_repeated : returns<build_repeated> {
  template <class Seq> struct apply :
  build_repeated_impl<typename front<Seq>::type, back<Seq>::type::value> {};
};

struct char_to_regex : returns<char_to_regex>
{ template <class C> struct apply : default_construct<char_<C::type::value>> {}; };

typedef transform<lit_c<'.'>, default_construct<any>> dot;
typedef transform<range_c<'a', 'z'>, char_to_regex> ch;

typedef transform<
  sequence<
    one_of<dot, ch>,
    one_of<lit_c<'*'>, return_<boost::mpl::char_<'x'>>>
  >,
  build_repeated
> repeated;

typedef entire_input<
  foldl<repeated, default_construct<empty>, build_seq>,
  invalid_regex_char
> regex_grammar;

typedef mpllibs::metaparse::build_parser<regex_grammar> regex_parser;

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

```
(), s.end()))
 std::string(s.begin(), i); }
```



Template metaprogram

```
if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
struct invalid_regex_char {
  typedef invalid_regex_char type
  static std::string get_value() { return std::string("Invalid regex char "); }
};

template <class T> struct returns { typedef T type; };

template <class E> struct default_construct : returns<default_construct<E>> {
  template <clas
  static auto r
};

template <class
template <class
  returns<build
{ static auto r

template <class

struct build_se
  template <cla
    static auto
    RETURNS(seq
  };
};

struct build_re
  template <cla
  build_repeate
};

struct char_to_
{ template <cla

typedef transfo
typedef transfo

typedef transfo
  sequence<
    one_of<dot,
    one_of<lit_
  >,
  build_repeate
> repeated;

typedef entire_
  foldl<repeate
  invalid_regex
> regex_grammar

typedef mpllibs

#define REGEX(s
```

```
typedef transform<lit_c<'.'>, default_construct<any>> dot;

typedef transform<range_c<'a', 'z'>, char_to_regex> ch;

typedef
  transform<
    sequence<
      one_of<dot, ch>,
      one_of<lit_c<'*'>, return_<boost::mpl::char_<'x'>>>
    >,
    build_repeated
  >
  repeated;

typedef
  entire_input<
    foldl<repeated, default_construct<empty>, build_seq>,
    invalid_regex_char
  >
  regex_grammar;
```

```
if (a
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
struct invalid_regex_char {
  typedef invalid_regex_char type
  static std::string get_value() { return std::string("Invalid regex char "); }
};

template <class T> struct returns { typedef T type; };

template <class E> struct default_construct : returns<default_construct<E>> {
  template <clas
  static auto r
};

template <class
template <class
  returns<build
{ static auto r

template <class

struct build_se
  template <cla
    static auto
    RETURNS(seq
  };
};

struct build_re
  template <cla
  build_repeate
};

struct char_to_
{ template <cla

typedef transfo
typedef transfo

typedef transfo
  sequence<
    one_of<dot,
    one_of<lit_
  >,
  build_repeate
> repeated;

typedef entire_
  foldl<repeate
  invalid_regex
> regex_grammar

typedef mpllibs

#define REGEX(s
```

```
typedef transform<lit_c<'.'>, default_construct<any>> dot;

typedef transform<range_c<'a', 'z'>, char_to_regex> ch;

typedef
  transform<
    sequence<
      one_of<dot, ch>,
      one_of<lit_c<'*'>, return_<boost::mpl::char_<'x'>>>
    >,
    build_repeated
  >
  repeated;

typedef
  entire_input<
    foldl<repeated, default_construct<empty>, build_seq>,
    invalid_regex_char
  >
  regex_grammar;
```

```
if (a
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.       abc
x       ab\
a*
```

```
struct invalid_regex_char {
  typedef invalid_regex_char type
  static std::string get_value() { return std::string("Invalid regex char "); }
};

template <class T> struct returns { typedef T type; };

template <class E> struct default_construct : returns<default_construct<E>> {
  template <clas
  static auto r
};

template <class
template <class
  returns<build
{ static auto r

template <class

struct build_se
  template <cla
    static auto
    RETURNS(seq
  };
};

struct build_re
  template <cla
  build_repeate
};

struct char_to_
{ template <cla

typedef transfo
typedef transfo

typedef transfo
  sequence<
    one_of<dot,
    one_of<lit_
  >,
  build_repeate
> repeated;

typedef entire_
  foldl<repeate
  invalid_regex
> regex_grammar

typedef mpllibs

#define REGEX(s
```

```
typedef transform<lit_c<'.'>, default_construct<any>> dot;

typedef transform<range_c<'a', 'z'>, char_to_regex> ch;

typedef
  transform<
    sequence<
      one_of<dot, ch>,
      one_of<lit_c<'*'>, return_<boost::mpl::char_<'x'>>>
    >,
    build_repeated
  >
  repeated;

typedef
  entire_input<
    foldl<repeated, default_construct<empty>, build_seq>,
    invalid_regex_char
  >
  regex_grammar;
```

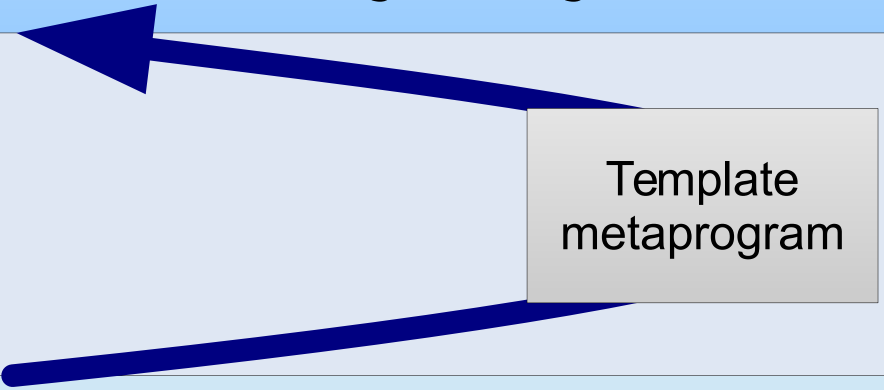
```
dot           ::= '.'
ch            ::= 'a'..'z'
repeated      ::= (dot | ch) '*'?
regex_grammar ::= repeated*
```

```
if (a
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
auto re = char_<'x'>();

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



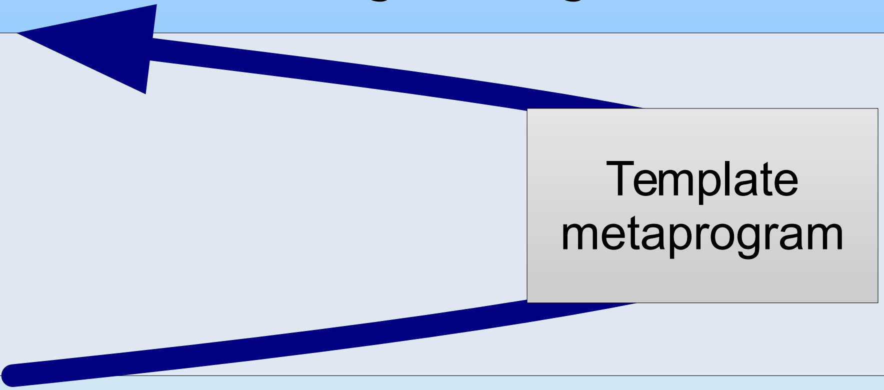
Template metaprogram

```
auto re = REGEX("x");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.     abc
x     ab\
a*
```

```
auto re = repeat<char_<'a'>>(char_<'a'>());

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



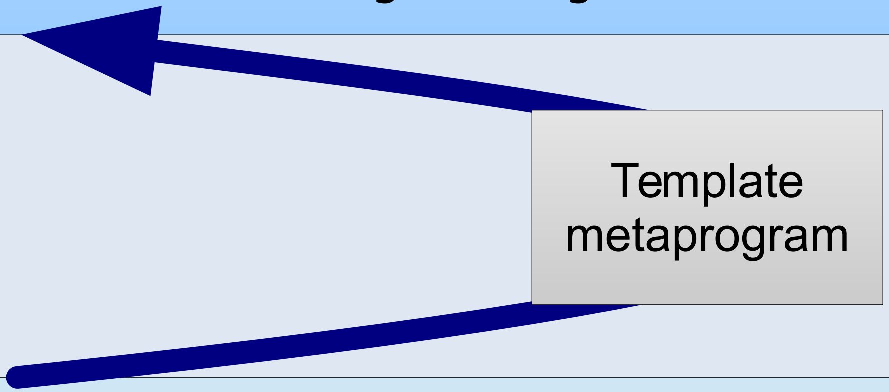
Template metaprogram

```
auto re = REGEX("a*");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

```
.      abc
x      ab\
a*
```

```
auto re = seq<char_<'a'>, char_<'b'>, char_<'c'>>(
  char_<'a'>(), char_<'b'>(), char_<'c'>());
std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```



Template metaprogram

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

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Example

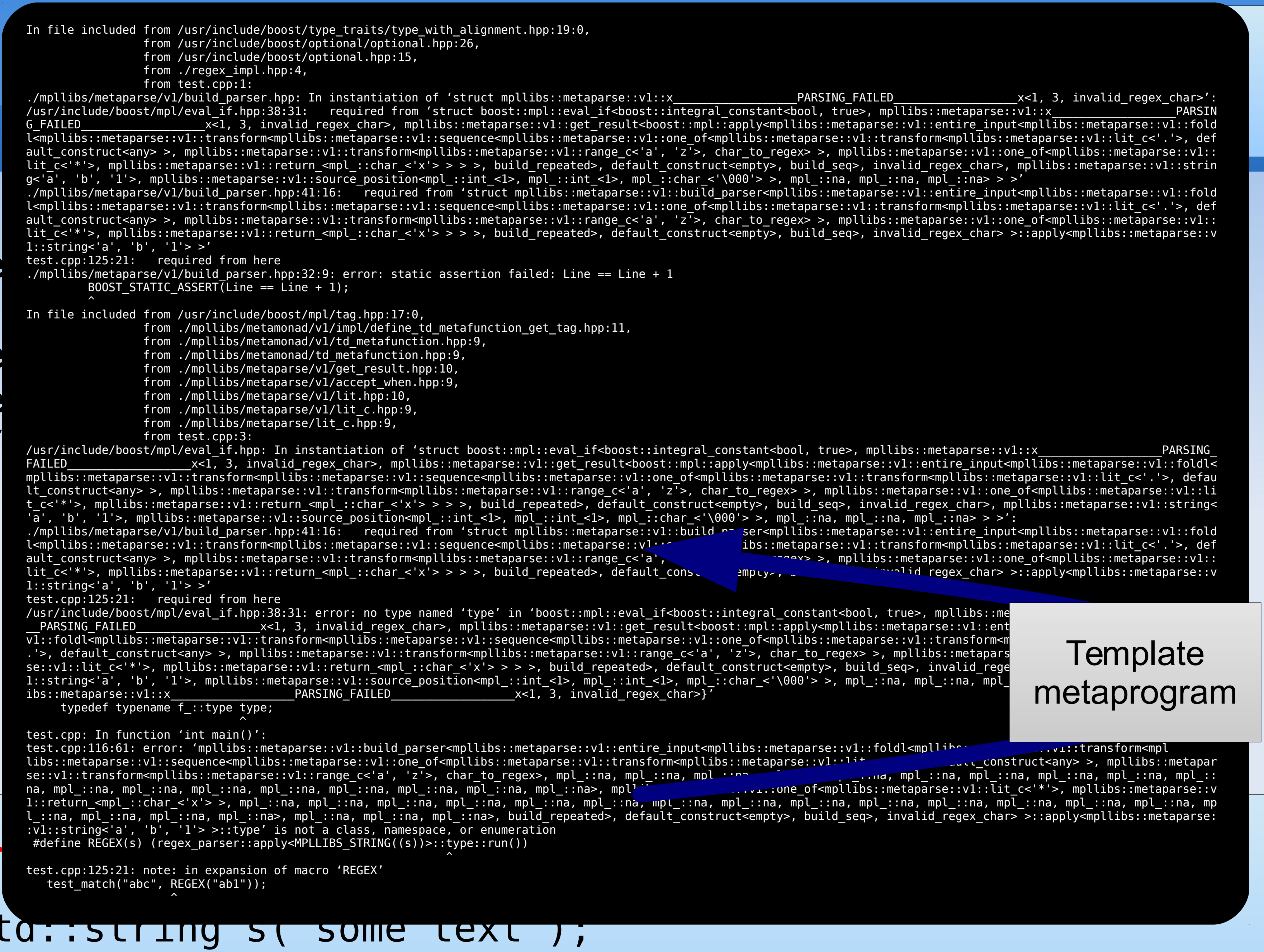
```
.     abc
x     ab\
a*
```

```
template <char C>       struct char_  { /* … */ };
                        struct any    { /* … */ };
template <class    E>   struct repeat { /* … */ };
template <class... Es>  struct seq    { /* … */ };
// …
```

Template metaprogram

```
auto re = REGEX("ab\\");

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

```
abc
ab\
```



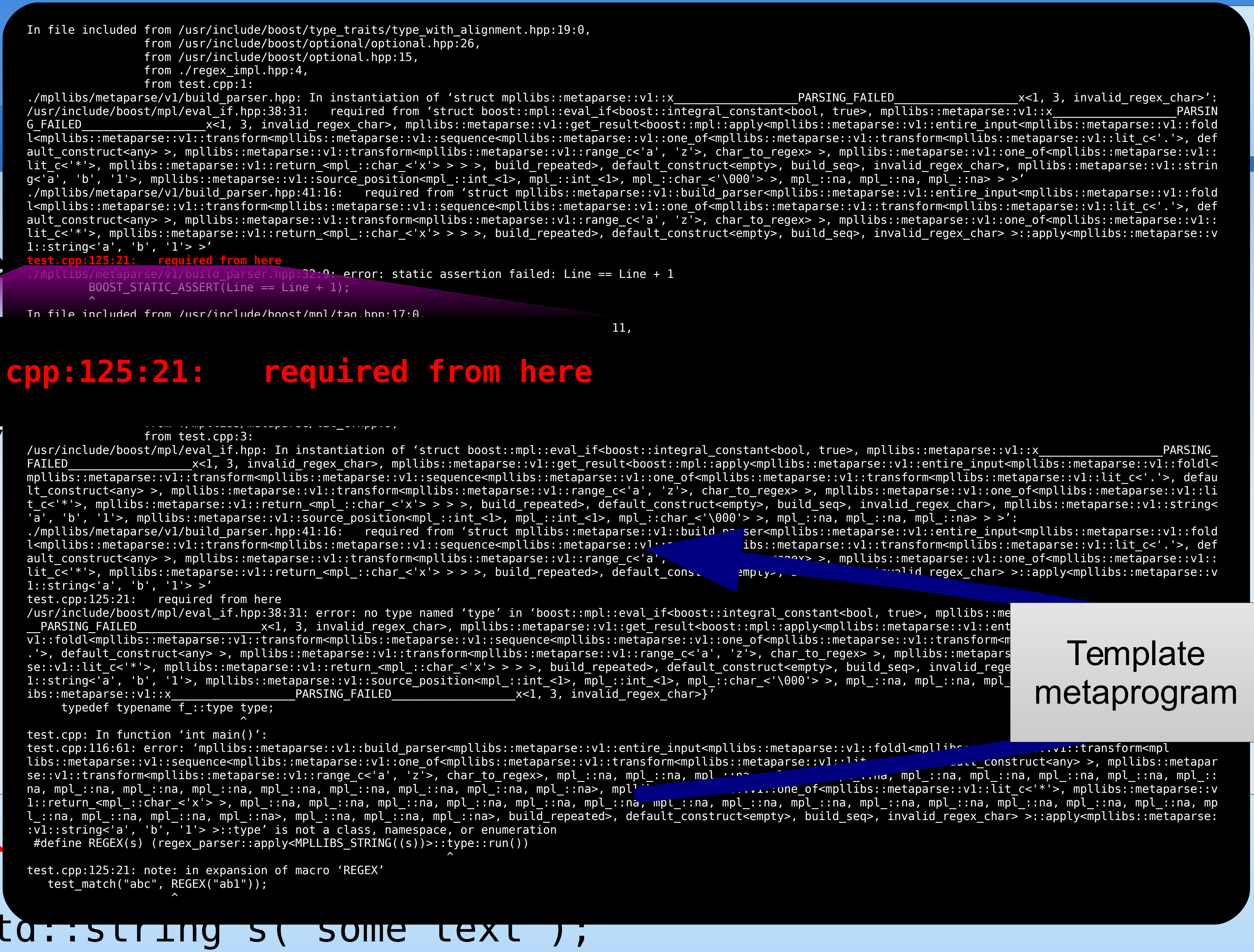
```
In file included from /usr/include/boost/type_traits/type_with_alignment.hpp:19:0,
                 from /usr/include/boost/optional/optional.hpp:26,
                 from /usr/include/boost/optional.hpp:15,
                 from ./regex_impl.hpp:4,
                 from test.cpp:1:
./mpllibs/metaparse/v1/build_parser.hpp: In instantiation of ‘struct mpllibs::metaparse::v1::x__________________PARSING_FAILED__________________x<1, 3, invalid_regex_char>’:
/usr/include/boost/mpl/eval_if.hpp:38:31:   required from ‘struct boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::metaparse::v1::x__________________PARSIN
G_FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char>, mpllibs::metaparse::v1::strin
g<'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_::na> > >’
./mpllibs/metaparse/v1/build_parser.hpp:41:16:   required from ‘struct mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char> >::apply<mpllibs::metaparse::v
1::string<'a', 'b', '1'> >’
test.cpp:125:21:   required from here
./mpllibs/metaparse/v1/build_parser.hpp:32:9: error: static assertion failed: Line == Line + 1
         BOOST_STATIC_ASSERT(Line == Line + 1);
         ^
In file included from /usr/include/boost/mpl/tag.hpp:17:0,
                 from ./mpllibs/metamonad/v1/impl/define_td_metafunction_get_tag.hpp:11,
                 from ./mpllibs/metamonad/v1/td_metafunction.hpp:9,
                 from ./mpllibs/metamonad/td_metafunction.hpp:9,
                 from ./mpllibs/metaparse/v1/get_result.hpp:10,
                 from ./mpllibs/metaparse/v1/accept_when.hpp:9,
                 from ./mpllibs/metaparse/v1/lit.hpp:10,
                 from ./mpllibs/metaparse/v1/lit_c.hpp:9,
                 from ./mpllibs/metaparse/lit_c.hpp:9,
                 from test.cpp:3:
/usr/include/boost/mpl/eval_if.hpp: In instantiation of ‘struct boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::metaparse::v1::x__________________PARSING_
FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::foldl<
mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, defau
lt_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::li
t_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char>, mpllibs::metaparse::v1::string<
'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_::na> > >’:
./mpllibs/metaparse/v1/build_parser.hpp:41:16:   required from ‘struct mpllibs::metaparse::v1::build_    ser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1            ibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a'                   >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_cons      empty>,             valid_regex_char> >::apply<mpllibs::metaparse::v
1::string<'a', 'b', '1'> >’
test.cpp:125:21:   required from here
/usr/include/boost/mpl/eval_if.hpp:38:31: error: no type named ‘type’ in ‘boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::me
__PARSING_FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::ent
v1::foldl<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<m
.'>, default_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metapars
se::v1::lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_rege
1::string<'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_
ibs::metaparse::v1::x__________________PARSING_FAILED__________________x<1, 3, invalid_regex_char>}’
     typedef typename f_::type type;
                               ^

test.cpp: In function ‘int main()’:
test.cpp:116:61: error: ‘mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::foldl<mpllibs                v1::transform<mpl
libs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit              _construct<any> >, mpllibs::metapar
se::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex>, mpl_::na, mpl_::na, mpl                  na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::
na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na>, mpl               v1::one_of<mpllibs::metaparse::v1::lit_c<'*'>, mpllibs::metaparse::v
1::return_<mpl_::char_<'x'> >, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mp
l_::na, mpl_::na, mpl_::na, mpl_::na>, mpl_::na, mpl_::na, mpl_::na>, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char> >::apply<mpllibs::metaparse:
:v1::string<'a', 'b', '1'> >::type’ is not a class, namespace, or enumeration
 #define REGEX(s) (regex_parser::apply<MPLLIBS_STRING((s))>::type::run())
                                                                 ^

test.cpp:125:21: note: in expansion of macro ‘REGEX’
   test_match("abc", REGEX("ab1"));
                     ^
```

Template metaprogram

```
te

te
te
//

au

std::string s("some text");

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

abc
ab\

te



```
In file included from /usr/include/boost/type_traits/type_with_alignment.hpp:19:0,
                 from /usr/include/boost/optional/optional.hpp:26,
                 from /usr/include/boost/optional.hpp:15,
                 from ./regex_impl.hpp:4,
                 from test.cpp:1:
./mpllibs/metaparse/v1/build_parser.hpp: In instantiation of ‘struct mpllibs::metaparse::v1::x__________________PARSING_FAILED__________________x<1, 3, invalid_regex_char>’:
/usr/include/boost/mpl/eval_if.hpp:38:31:   required from ‘struct boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::metaparse::v1::x__________________PARSIN
G_FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char>, mpllibs::metaparse::v1::strin
g<'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_::na> > >’
./mpllibs/metaparse/v1/build_parser.hpp:41:16:   required from ‘struct mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char> >::apply<mpllibs::metaparse::v
1::string<'a', 'b', '1'> >’
test.cpp:125:21:   required from here
./mpllibs/metaparse/v1/build_parser.hpp:32:9: error: static assertion failed: Line == Line + 1
         BOOST_STATIC_ASSERT(Line == Line + 1);
         ^
In file included from /usr/include/boost/mpl/tag.hpp:17:0,
                                                                                         11,
                 from test.cpp:3:
/usr/include/boost/mpl/eval_if.hpp: In instantiation of ‘struct boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::metaparse::v1::x__________________PARSING_
FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::foldl<
mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, defau
lt_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::li
t_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char>, mpllibs::metaparse::v1::string<
'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_::na> > >’:
./mpllibs/metaparse/v1/build_parser.hpp:41:16:   required from ‘struct mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char> >::apply<mpllibs::metaparse::v
1::string<'a', 'b', '1'> >’
test.cpp:125:21:   required from here
/usr/include/boost/mpl/eval_if.hpp:38:31: error: no type named ‘type’ in ‘boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::me
__PARSING_FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::ent
v1::foldl<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<m
.'>, default_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metapars
se::v1::lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_rege
1::string<'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_
ibs::metaparse::v1::x__________________PARSING_FAILED__________________x<1, 3, invalid_regex_char>}’
     typedef typename f_::type type;
                               ^

test.cpp: In function ‘int main()’:
test.cpp:116:61: error: ‘mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::foldl<mpllibs              ::v1::transform<mpl
libs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit              ault_construct<any> >, mpllibs::metapar
se::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex>, mpl_::na, mpl_::na, mpl_::na              na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::
na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na>, mpll              ::v1::one_of<mpllibs::metaparse::v1::lit_c<'*'>, mpllibs::metaparse::v
1::return_<mpl_::char_<'x'> >, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mp
l_::na, mpl_::na, mpl_::na, mpl_::na>, mpl_::na, mpl_::na, mpl_::na>, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char> >::apply<mpllibs::metaparse:
:v1::string<'a', 'b', '1'> >::type’ is not a class, namespace, or enumeration
 #define REGEX(s) (regex_parser::apply<MPLLIBS_STRING((s))>::type::run())
                                                              ^

test.cpp:125:21: note: in expansion of macro ‘REGEX’
    test_match("abc", REGEX("ab1"));
                    ^
```

test.cpp:125:21:   required from here

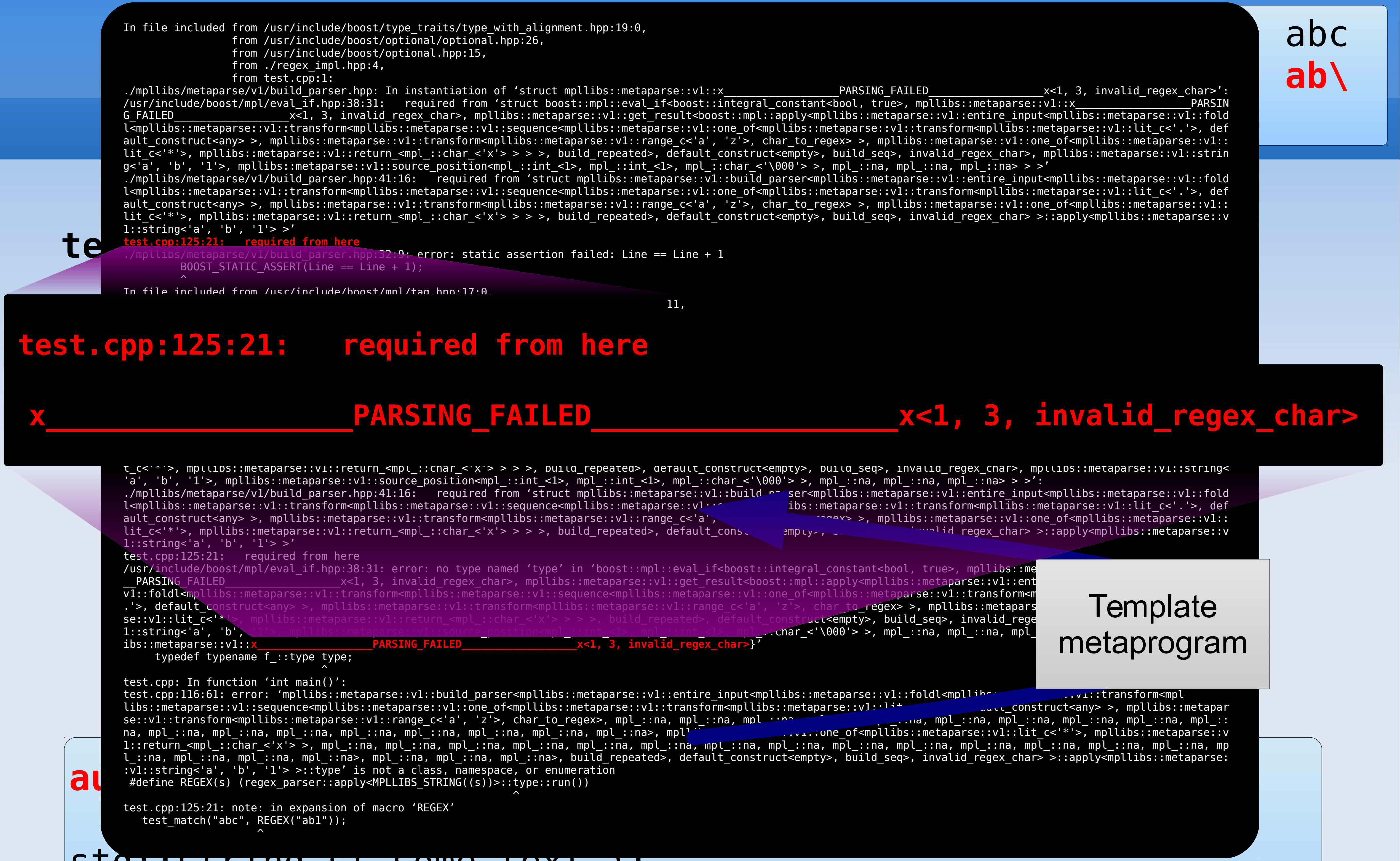
Template metaprogram

//

au

```
std::string s( some text );

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

abc
ab\



```
In file included from /usr/include/boost/type_traits/type_with_alignment.hpp:19:0,
                 from /usr/include/boost/optional/optional.hpp:26,
                 from /usr/include/boost/optional.hpp:15,
                 from ./regex_impl.hpp:4,
                 from test.cpp:1:
./mpllibs/metaparse/v1/build_parser.hpp: In instantiation of ‘struct mpllibs::metaparse::v1::x__________________PARSING_FAILED__________________x<1, 3, invalid_regex_char>’:
/usr/include/boost/mpl/eval_if.hpp:38:31:   required from ‘struct boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::metaparse::v1::x__________________PARSIN
G_FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char>, mpllibs::metaparse::v1::strin
g<'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_::na> > >’
./mpllibs/metaparse/v1/build_parser.hpp:41:16:   required from ‘struct mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char> >::apply<mpllibs::metaparse::v
1::string<'a', 'b', '1'> >’
test.cpp:125:21:   required from here
./mpllibs/metaparse/v1/build_parser.hpp:32:9: error: static assertion failed: Line == Line + 1
         BOOST_STATIC_ASSERT(Line == Line + 1);
         ^
In file included from /usr/include/boost/mpl/tag.hpp:17:0,
                                                                                     11,
```

test.cpp:125:21:   required from here

x________________________PARSING_FAILED________________________x<1, 3, invalid_regex_char>

```
t_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char>, mpllibs::metaparse::v1::string<
'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_::na> > >’:
./mpllibs/metaparse/v1/build_parser.hpp:41:16:   required from ‘struct mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::fold
l<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::                 ibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit_c<'.'>, def
ault_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a',                  gex> >, mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::
lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_cons      empty>,           valid_regex_char> >::apply<mpllibs::metaparse::v
1::string<'a', 'b', '1'> >’
test.cpp:125:21:   required from here
/usr/include/boost/mpl/eval_if.hpp:38:31: error: no type named ‘type’ in ‘boost::mpl::eval_if<boost::integral_constant<bool, true>, mpllibs::me
__PARSING_FAILED__________________x<1, 3, invalid_regex_char>, mpllibs::metaparse::v1::get_result<boost::mpl::apply<mpllibs::metaparse::v1::ent
v1::foldl<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<m
.'>, default_construct<any> >, mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex> >, mpllibs::metapars
se::v1::lit_c<'*'>, mpllibs::metaparse::v1::return_<mpl_::char_<'x'> > > >, build_repeated>, default_construct<empty>, build_seq>, invalid_rege
1::string<'a', 'b', '1'>, mpllibs::metaparse::v1::source_position<mpl_::int_<1>, mpl_::int_<1>, mpl_::char_<'\000'> >, mpl_::na, mpl_::na, mpl_
ibs::metaparse::v1::x__________________PARSING_FAILED__________________x<1, 3, invalid_regex_char>}’
     typedef typename f_::type type;
                               ^

test.cpp: In function ‘int main()’:
test.cpp:116:61: error: ‘mpllibs::metaparse::v1::build_parser<mpllibs::metaparse::v1::entire_input<mpllibs::metaparse::v1::foldl<mpllibs           ::v1::transform<mpl
libs::metaparse::v1::sequence<mpllibs::metaparse::v1::one_of<mpllibs::metaparse::v1::transform<mpllibs::metaparse::v1::lit           ult_construct<any> >, mpllibs::metapar
se::v1::transform<mpllibs::metaparse::v1::range_c<'a', 'z'>, char_to_regex>, mpl_::na, mpl_::na, mpl_::na           _::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::
na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na>, mpl           ::v1::one_of<mpllibs::metaparse::v1::lit_c<'*'>, mpllibs::metaparse::v
1::return_<mpl_::char_<'x'> >, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mp
l_::na, mpl_::na, mpl_::na, mpl_::na>, mpl_::na, mpl_::na, mpl_::na>, build_repeated>, default_construct<empty>, build_seq>, invalid_regex_char> >::apply<mpllibs::metaparse:
:v1::string<'a', 'b', '1'> >::type’ is not a class, namespace, or enumeration
 #define REGEX(s) (regex_parser::apply<MPLLIBS_STRING((s))>::type::run())
                                                                 ^

test.cpp:125:21: note: in expansion of macro ‘REGEX’
   test_match("abc", REGEX("ab1"));
                    ^
```

Template metaprogram

```
std::string s( some text );

if (auto i = re.match(s.begin(), s.end()))
{ std::cout << "matched: " << std::string(s.begin(),*i); }
```

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗		✗
Readable error messages	✓	✗	✗	✓		✓
Usable in library headers	✗	✓	✓	✓		✓
Code completion	✗	✗	✓	✗		✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓		✓
”Normal” C++	✓	✗	✗	✗		✓
No metaprogramming	✓	✗	✗	✗		✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓		✓
Usable in library headers	✗	✓	✓	✓		✓
Code completion	✗	✗	✓	✗		✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓		✓
”Normal” C++	✓	✗	✗	✗		✓
No metaprogramming	✓	✗	✗	✗		✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓	✗	✓
Usable in library headers	✗	✓	✓	✓		✓
Code completion	✗	✗	✓	✗		✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓		✓
”Normal” C++	✓	✗	✗	✗		✓
No metaprogramming	✓	✗	✗	✗		✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓	✗	✓
Usable in library headers	✗	✓	✓	✓	✓	✓
Code completion	✗	✗	✓	✗		✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓		✓
”Normal” C++	✓	✗	✗	✗		✓
No metaprogramming	✓	✗	✗	✗		✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓	✗	✓
Usable in library headers	✗	✓	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗	✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓		✓
”Normal” C++	✓	✗	✗	✗		✓
No metaprogramming	✓	✗	✗	✗		✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓	✗	✓
Usable in library headers	✗	✓	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗	✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗		✓
No metaprogramming	✓	✗	✗	✗		✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓	✗	✓
Usable in library headers	✗	✓	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗	✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗	✗	✓
No metaprogramming	✓	✗	✗	✗		✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓	✗	✓
Usable in library headers	✗	✓	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗	✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗	✗	✓
No metaprogramming	✓	✗	✗	✗	✗	✓
No build system support	✗	✓	✓	✓		✓

Evaluation

	External script	Preprocessor	Method chaining	Operator overloading	Parsing at compile-time	Parsing at runtime
Using the DSL						
No syntax changes	✓	✗	✗	✗	✓	✓
Compile-time validation	✓	✓	✓	✗	✓	✗
Readable error messages	✓	✗	✗	✓	✗	✓
Usable in library headers	✗	✓	✓	✓	✓	✓
Code completion	✗	✗	✓	✗	✗	✗
Implementing the DSL						
Only standard C++	✗	✓	✓	✓	✓	✓
”Normal” C++	✓	✗	✗	✗	✗	✓
No metaprogramming	✓	✗	✗	✗	✗	✓
No build system support	✗	✓	✓	✓	✓	✓

How fast is it?

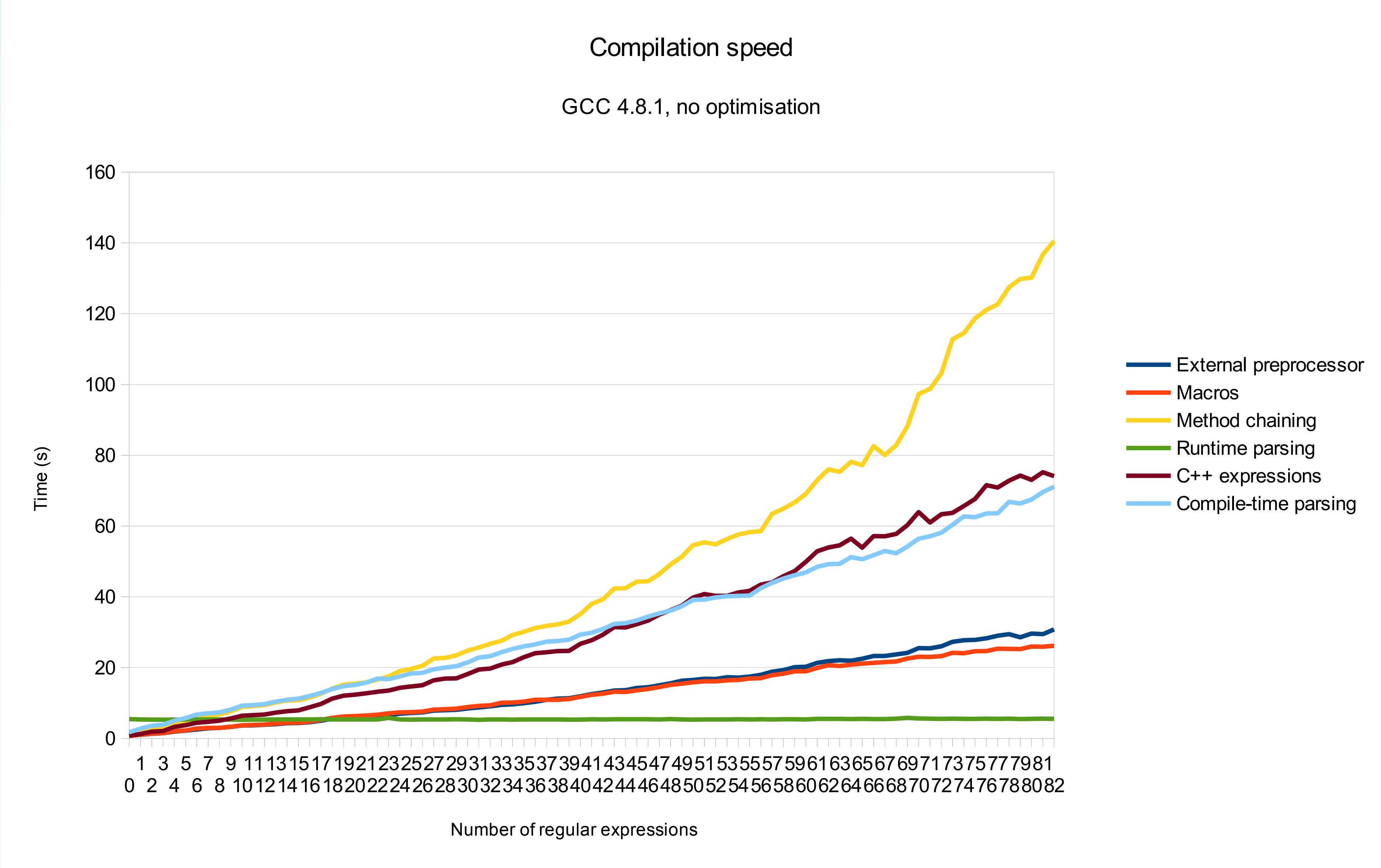
- GCC 4.8.1
- Ubuntu 13.10
- Memory: 4 GB
- Processor: Intel Core i5 3337U

How fast is it?

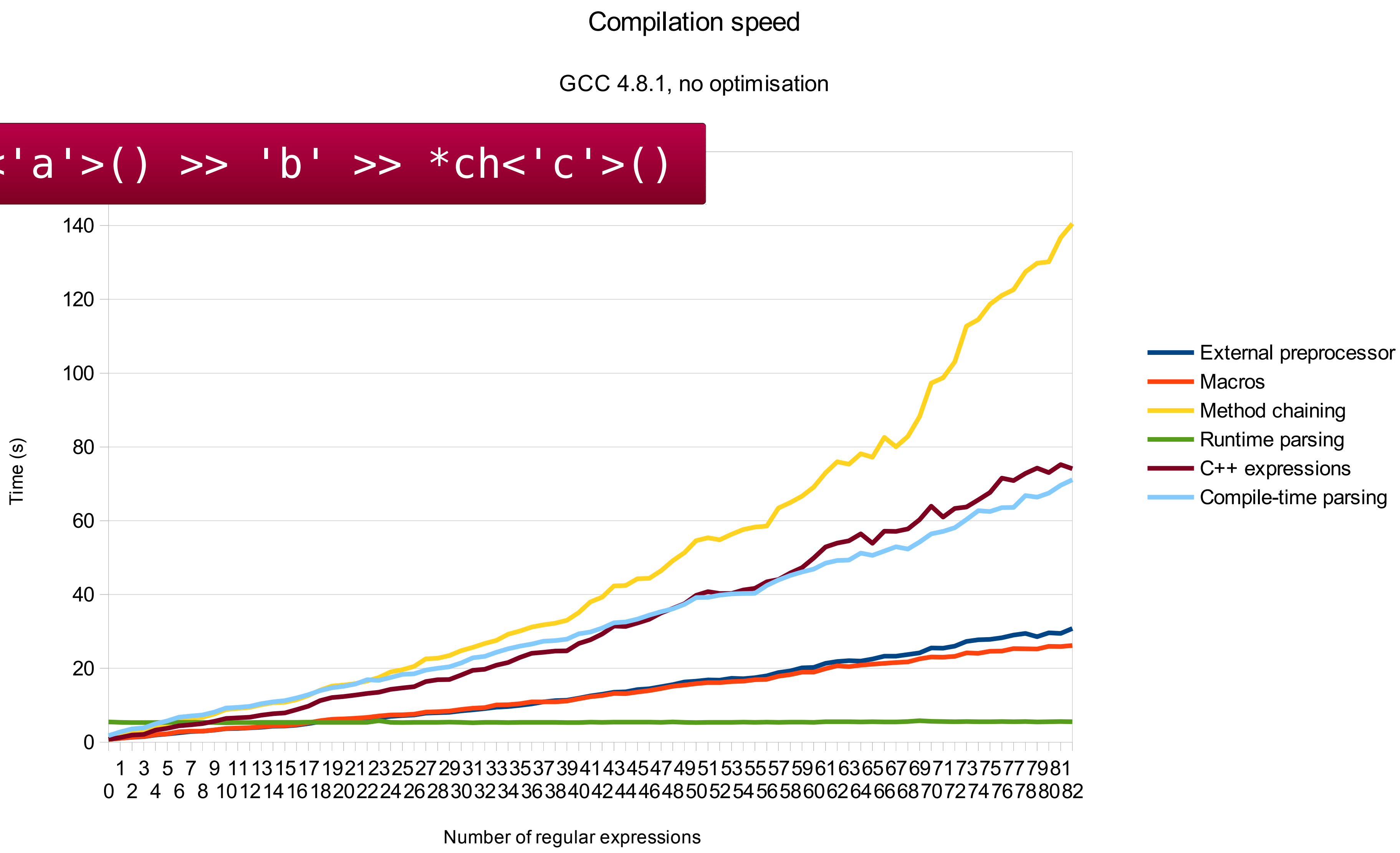
- GCC 4.8.1
- Ubuntu 13.10
- Memory: 4 GB
- Processor: Intel Core i5 3337U

- Create *n* regular expressions
- Try matching one string

How fast does it compile?



How fast does it compile?



```
ch<'a'>() >> 'b' >> *ch<'c'>()
```

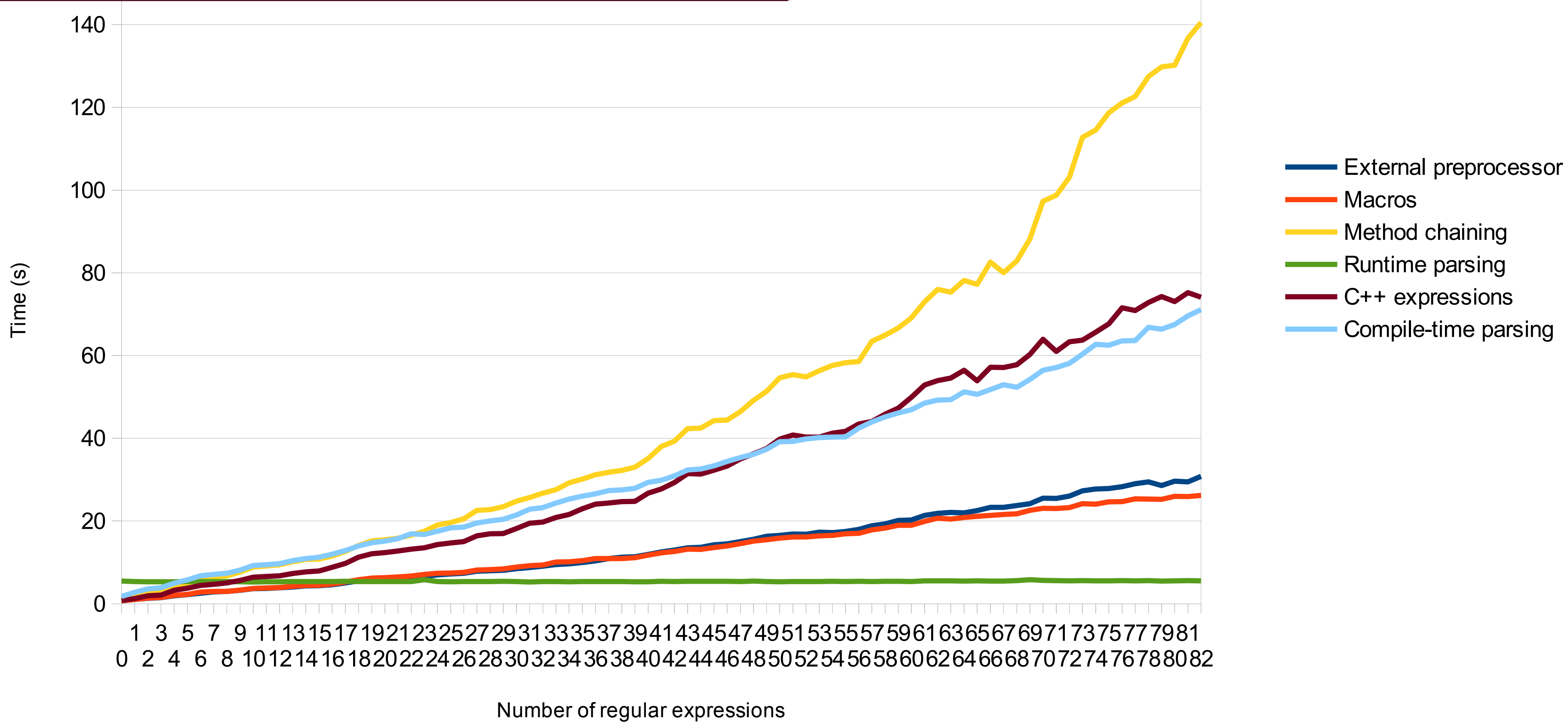
How fast does it compile?

```
regex
  .char_<'a'>()
  .char_<'b'>()
  .char_<'c'>().repeat();
```

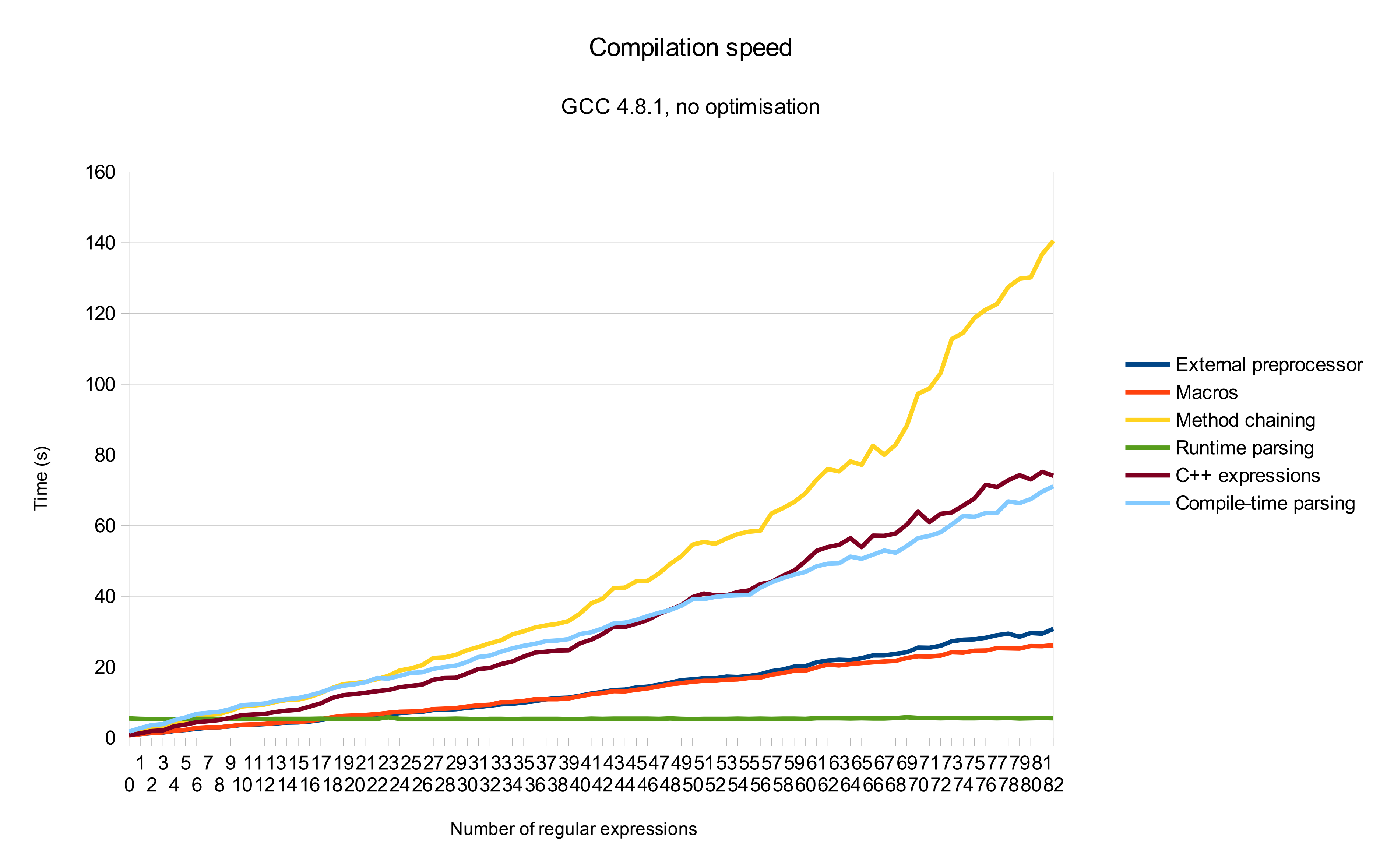
```
ch<'a'>() >> 'b' >> *ch<'c'>()
```

Compilation speed

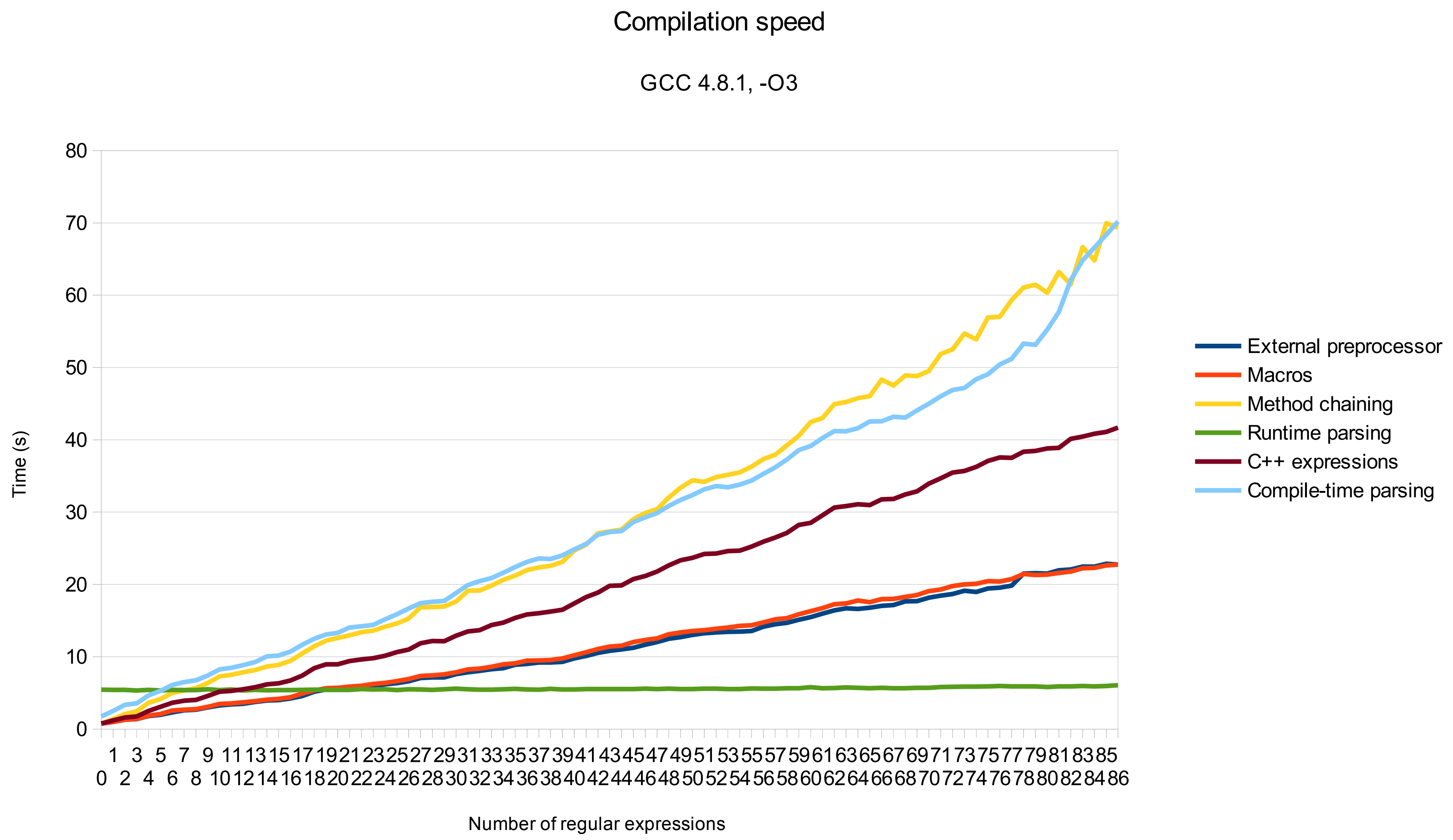
GCC 4.8.1, no optimisation



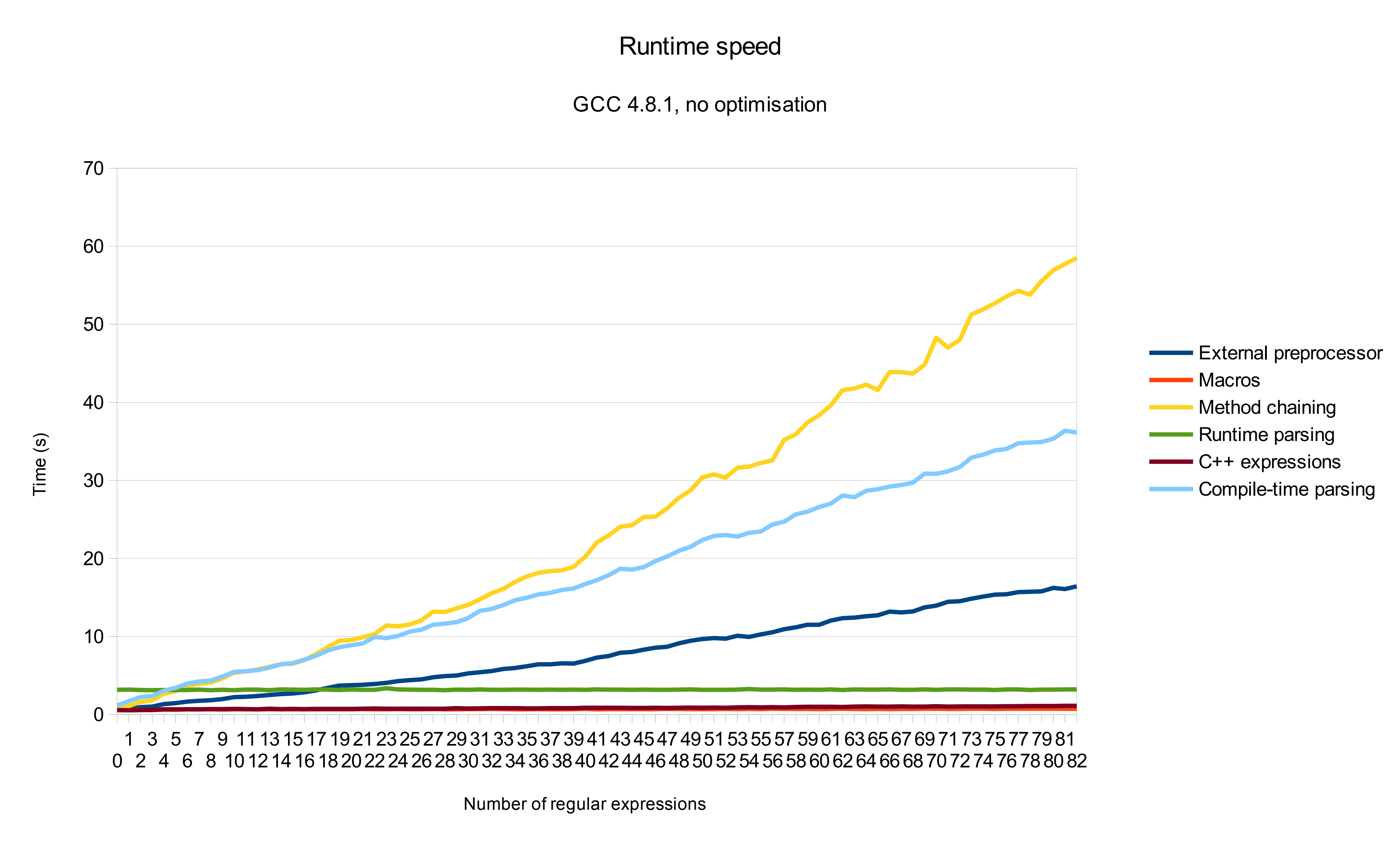
How fast does it compile?



How fast does it compile?



How fast does it run?



Summary

- Embedding domain-specific languages
- Different methods
  - Before compilation
  - During compilation
  - At runtime

Q & A

http://abel.sinkovics.hu
abel@sinkovics.hu

http://github.com/sabel83