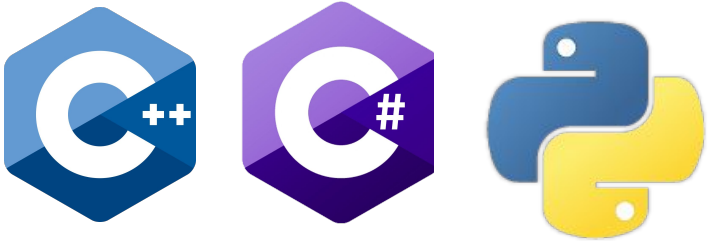
Monadic parsers

Alexander Granin
graninas@gmail.com

C++ Russia 2019, Moscow



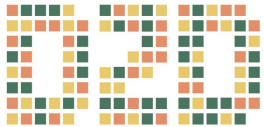
KASPERSKY lab

2GIS

JUSPAY

Restaumatic

DEVDAY
OT 2GIS



LambdaNsk

fpconf

C++ Siberia 2019, Keynote

The Present and the Future of Functional Programming in C++

C++ Russia 2018

Functional Approach to Software Transactional Memory in C++

C++ Russia 2016

Functional "Life": Parallel Cellular Automata and Comonads in C++

C++ Siberia 2015

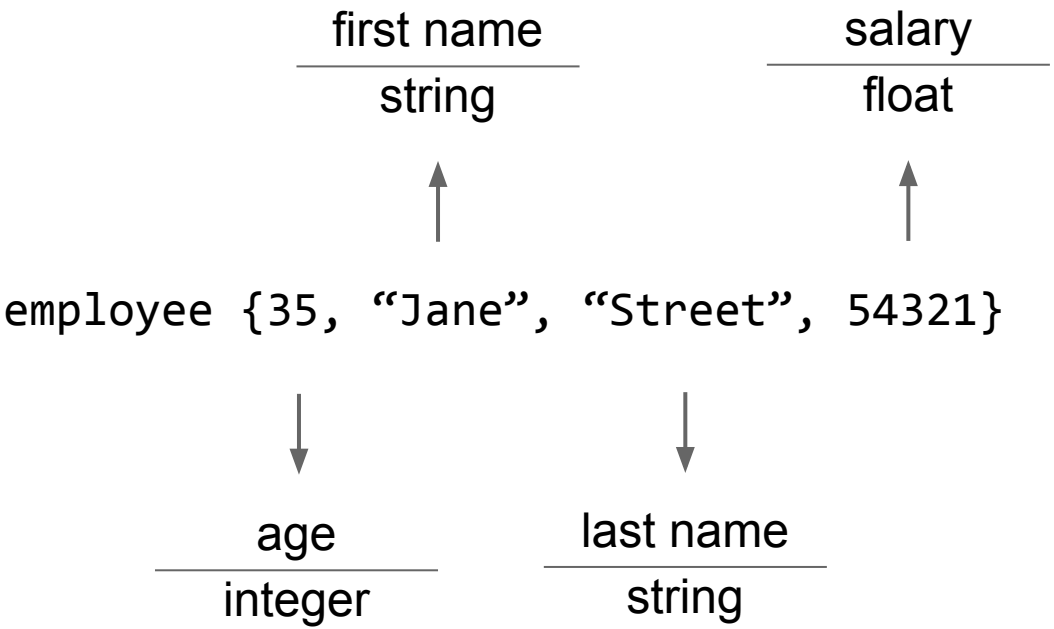
Functional Microscope: Lenses in C++

C++ User Group 2014

Functional and Declarative Design in C++

```
struct Presentation
{
    Introduction
        Parsing and Backus-Naur form
        boost::spirit (qi)
        cpp_parsec_free
    Monadic parsers
    State & context-sensitive grammars
    Conclusion
};
```

Parsing




```
struct Employee
{
    int    age;
    string firstName;
    string lastName;
    double salary;
};
```

Backus-Naur Form

```
employee ::=
  "employee" spaces
  "{" spaces
    age "," spaces
    firstName "," spaces
    lastName "," spaces
    salary spaces
  "}"
```

```
quotedString ::= '"' string '"'
age          ::= integer
firstName    ::= quotedString
lastName     ::= quotedString
salary       ::= double
```

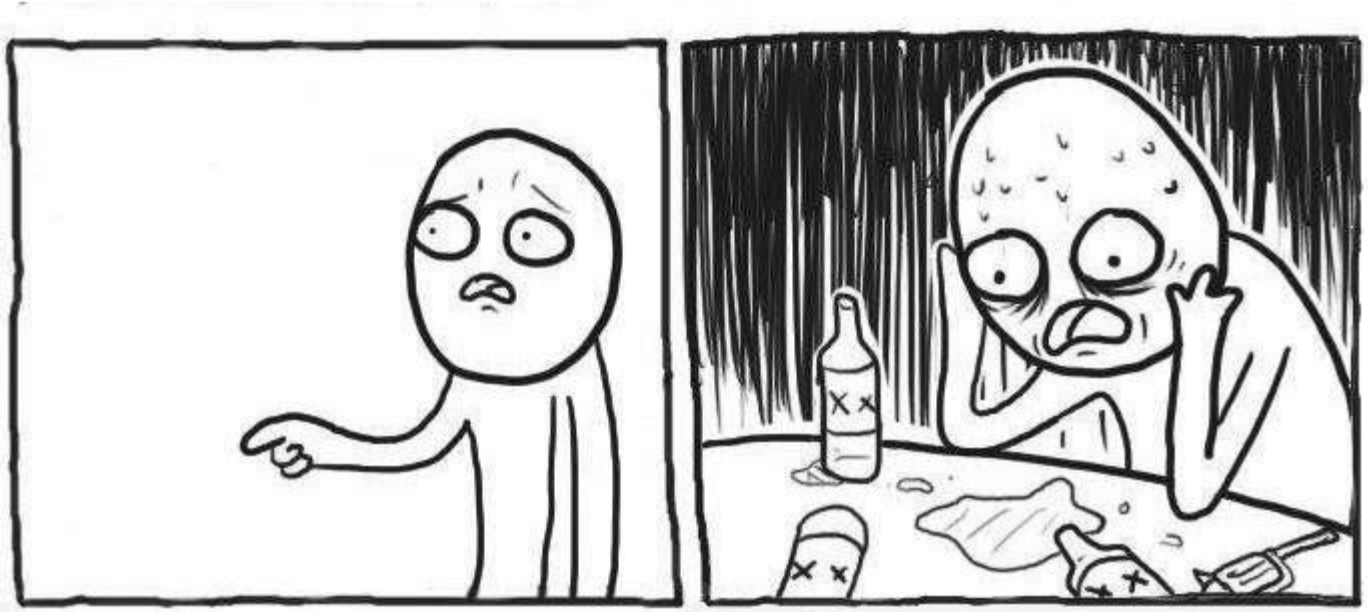
boost::spirit (Qi)

```
quoted_string %= lexeme['"' >> +(char_ - '"') >> '"'];

start %= lit("employee") >> '{'
            >>  int_ >> ','
            >>  quoted_string >> ','
            >>  quoted_string >> ','
            >>  double_ >>  '}';
}
```

```
quoted_string %= lexeme['"' >> +(char_ - '"') >> '"'];

start %= lit("employee") >> '{'
            >>  int_ >> ','
            >>  quoted_string >> ','
            >>  quoted_string >> ','
            >>  double_ >>  '}';
}

qi::rule<Iterator, string(),   ascii::space_type> quoted_string;
qi::rule<Iterator, employee(), ascii::space_type> start;
```

```
template <typename Iterator>
struct employee_parser : qi::grammar<Iterator, employee(), ascii::space_type>{
   employee_parser() : employee_parser::base_type(start) {

   quoted_string %= lexeme['"' >> +(char_ - '"') >> '"'];

   start %= lit("employee") >> '{'
             >>  int_ >> ','
             >>  quoted_string >> ','
             >>  quoted_string >> ','
             >>  double_ >>  '}';
   }

   qi::rule<Iterator, string(),   ascii::space_type> quoted_string;
   qi::rule<Iterator, employee(), ascii::space_type> start;
};
```

```
template <typename Iterator>
struct employee_parser : qi::grammar<Iterator, employee(), ascii::space_type>{
    employee_parser() : employee_parser::base_type(start) {

    quoted_string %= lexeme['"' >> +(char_ - '"') >> '"'];

    start %= lit("employee") >> '{'
                >>  int_ >> ','
                >>  quoted_string >> ','
                >>  quoted_string >> ','
                >>  double_ >>  '}';
    }

    qi::rule<Iterator, string(),   ascii::space_type> quoted_string;
    qi::rule<Iterator, employee(), ascii::space_type> start;
};
```



cpp_parsec_free

```
ParserT<Employee> employee =
    app<Employee, int, string, string, double>(
        mkEmployee,
        prefix >> intP,
        comma  >> quotedString,
        comma  >> quotedString,
        comma  >> (doubleP << postfix));
```

```
ParserT<Employee> employee =
   app<Employee, int, string, string, double>(
       mkEmployee,
       prefix >> intP,
       comma  >> quotedString,
       comma  >> quotedString,
       comma  >> (doubleP << postfix));

auto prefix  = lit("employee") >> between(spaces, symbol('{'));
auto postfix = between(spaces, symbol('}'));
auto comma   = symbol(',') >> spaces;

auto mkEmployee =
    [](int a, const string& sn, const string& fn, double s) {
        return Employee {a, sn, fn, s};
};
```

```
auto s = "employee {35, “Jane”, “Street”, 54321}";

ParserResult<Employee> result = parse(employee, s);
```

```
auto s = "employee {35, “Jane”, “Street”, 54321}";

ParserResult<Employee> result = parse(employee, s);

if (isLeft(result)) {
   std::cout << "Parse error: " << getError(result).message;
}
else {
   Employee employee = getParsed(result);
}
```

Parser combinators

Basic parsers

```
ParserT<int>    intP;                   // integer number
ParserT<double> doubleP;                // double precision float

ParserT<Char>   digit;                  // 0..9
ParserT<Char>   lower;                  // a..z
ParserT<Char>   upper;                  // A..Z
ParserT<Char>   letter;                 // a..z A..Z
ParserT<Char>   alphaNum;               // 0..9 a..z A..Z
ParserT<Char>   symbol(Char ch);        // symbol

ParserT<string> lit(const string&);     // string
```

Parser combinators

```
template <typename A, typename B>
ParserT<A> fst(const ParserT<A>& p1, const ParserT<B>& p2);

template <typename A, typename B>
ParserT<B> snd(const ParserT<A>& p1, const ParserT<B>& p2);
```

Parser combinators

```
template <typename A, typename B>
ParserT<A> fst(const ParserT<A>& p1, const ParserT<B>& p2);

template <typename A, typename B>
ParserT<B> snd(const ParserT<A>& p1, const ParserT<B>& p2);

// fst
template <typename A, typename B>
ParserT<A> operator<< (const ParserT<A>& l, const ParserT<B>& r);

// snd
template <typename A, typename B>
ParserT<B> operator>> (const ParserT<A>& l, const ParserT<B>& r);
```

Parsing “Hello world”

```
ParserT<string> helloWorld =
    snd(lit("Hello"), snd(symbol(' '), lit("world")));
```

```
ParserT<string> helloWorld =
    lit("Hello") >> symbol(' ') >> lit("world");
```

More combinators

```
template <typename A>
ParserT<Many<A>> many(const ParserT<A>& p);          // 0 or many

template <typename A>
ParserT<Many<A>> many1(const ParserT<A>& p);         // 1 or many

template <typename A, typename B>
ParserT<B> between(const ParserT<A>& bracketP,       // between
                   const ParserT<B>& p);
```

Alternative

```
template <typename A>
ParserT<A> alt(const ParserT<A>& l, const ParserT<A>& r);

template <typename A>
ParserT<A> operator| (const ParserT<A>& l, const ParserT<A>& r);
```

Alternative

```
template <typename A>
ParserT<A> alt(const ParserT<A>& l, const ParserT<A>& r);

template <typename A>
ParserT<A> operator| (const ParserT<A>& l, const ParserT<A>& r);
```

```
ParserT<string> boolean  = alt(lit("true"), lit("false"));

ParserT<string> currency = lit("EUR") | lit("USD") | lit("GBP");
```

Monadic parsers

Functor

```
fmap :: (A -> B) -> ParserT<A> -> ParserT<B>
```

Applicative

```
app :: (A -> B -> C) -> ParserT<A> -> ParserT<B> -> ParserT<C>
```

Monad

```
bind :: ParserT<A> -> (A -> ParserT<B>)
```

Functor

```
fmap :: (A -> B) -> ParserT<A> -> ParserT<B>
```

```
template <typename A, typename B>
ParserT<B> fmap(const function<B(A)>& f, const ParserT<A>& p);
```

Functor

```
fmap :: (A -> B) -> ParserT<A> -> ParserT<B>
```

```
template <typename A, typename B>
ParserT<B> fmap(const function<B(A)>& f, const ParserT<A>& p);
```

```
ParserT<string> strBoolean = lit("true") | lit("false");
```

Functor

```
fmap :: (A -> B) -> ParserT<A> -> ParserT<B>
```

```
template <typename A, typename B>
ParserT<B> fmap(const function<B(A)>& f, const ParserT<A>& p);
```

```
ParserT<string> strBoolean = lit("true") | lit("false");

ParserT<bool>   boolean = fmap<string, bool>(f, strBoolean);

function<bool(string)> f = [](const string& s) {
    return s == "true";
};
```

Functor

```
fmap :: (A -> B) -> ParserT<A> -> ParserT<B>
```

Applicative

```
app :: (A -> B -> C) -> ParserT<A> -> ParserT<B> -> ParserT<C>
```

Monad

```
bind :: ParserT<A> -> (A -> ParserT<B>)
```

Applicative

```
app :: (A -> B -> C) -> ParserT<A> -> ParserT<B> -> ParserT<C>
```

```
template <typename A, typename B, typename C>
ParserT<C> app(const function<C(A, B)>& f,
               const ParserT<A>& a, const ParserT<B>& b);
```

Applicative

```
app :: (A -> B -> C) -> ParserT<A> -> ParserT<B> -> ParserT<C>
```

```
template <typename A, typename B, typename C>
ParserT<C> app(const function<C(A, B)>& f,
               const ParserT<A>& a, const ParserT<B>& b);
```

```
ParserT<string> helloUsername = app<string, string>(
    mkHello,
    letters,
    spaces >> letters);
```

```
string mkHello = [](const string& firstName, const string& secondName) {
    return string("Hello, ") + firstName + " " + secondName + "!"; };
```

Applicative is combinable

```
ParserT<Value> value = app(mkValue, intP | doubleP);
```

Applicative is combinable

```
ParserT<Value> value = app(mkValue, intP | doubleP);

ParserT<BinOp> binOp = app(mkBinOp,
    symbol('+') | symbol('-') | symbol('*') | symbol('/'));
```

Applicative is combinable

```
ParserT<Value> value = app(mkValue, intP | doubleP);

ParserT<BinOp> binOp = app(mkBinOp,
    symbol('+') | symbol('-') | symbol('*') | symbol('/'));

ParserT<FuncArgs> funcArgs = app(mkFuncArgs,
    expr, opt(symbol(',') >> funcArgs));
```

Applicative is combinable

```
ParserT<Value> value = app(mkValue, intP | doubleP);

ParserT<BinOp> binOp = app(mkBinOp,
    symbol('+') | symbol('-') | symbol('*') | symbol('/'));

ParserT<FuncArgs> funcArgs = app(mkFuncArgs,
    expr, opt(symbol(',') >> funcArgs));

ParserT<FuncCall> funcCall = app(mkFuncCall,
    identifier, between('(', ')', funcArgs));
```

Applicative is combinable

```
ParserT<Value> value = app(mkValue, intP | doubleP);

ParserT<BinOp> binOp = app(mkBinOp,
    symbol('+') | symbol('-') | symbol('*') | symbol('/'));

ParserT<FuncArgs> funcArgs = app(mkFuncArgs,
    expr, opt(symbol(',') >> funcArgs));

ParserT<FuncCall> funcCall = app(mkFuncCall,
    identifier, between('(', ')', funcArgs));

ParserT<BinExpr> binExpr = app(mkBinExpr, expr, binOp, expr);
```

Applicative is combinable

```
ParserT<Value> value = app(mkValue, intP | doubleP);

ParserT<BinOp> binOp = app(mkBinOp,
    symbol('+') | symbol('-') | symbol('*') | symbol('/'));

ParserT<FuncArgs> funcArgs = app(mkFuncArgs,
    expr, opt(symbol(',') >> funcArgs));

ParserT<FuncCall> funcCall = app(mkFuncCall,
    identifier, between('(', ')', funcArgs));

ParserT<BinExpr> binExpr = app(mkBinExpr, expr, binOp, expr);

ParserT<Expr> expr = app(mkExpr, binExpr | funcCall | value);
```

Functor

```
fmap :: (A -> B) -> ParserT<A> -> ParserT<B>
```

Applicative

```
app :: (A -> B -> C) -> ParserT<A> -> ParserT<B> -> ParserT<B>
```

Monad

```
bind :: ParserT<A> -> (A -> ParserT<B>)
```

ParserT monad

```
template <typename A>
ParserT<A> pure(const A& a);

template <typename A, typename B>
ParserT<B> bind(
    const ParserT<A>& ma,
    const function<ParserT<B>(A)>& f);
```

Monadic binding

```
ParserT<string> helloUsername =
  bind<string, string>(letters, [=](auto firstName) {
    return bind<string, string>(spaces, [=](auto) {
      return bind<string, string>(letters, [=](auto lastName) {
        return pure<string>(mkHello(firstName, lastName)); }); }); });
```

```
ParserT<string> helloUsername =
  bind<string, string>(letters,       [=](auto firstName) { return
  bind<string, string>(spaces,        [=](auto)           { return
  bind<string, string>(letters,       [=](auto lastName)  { return
  pure<string>(mkHello(firstName, lastName)); }); }); });
```

Monadic binding, reformatted

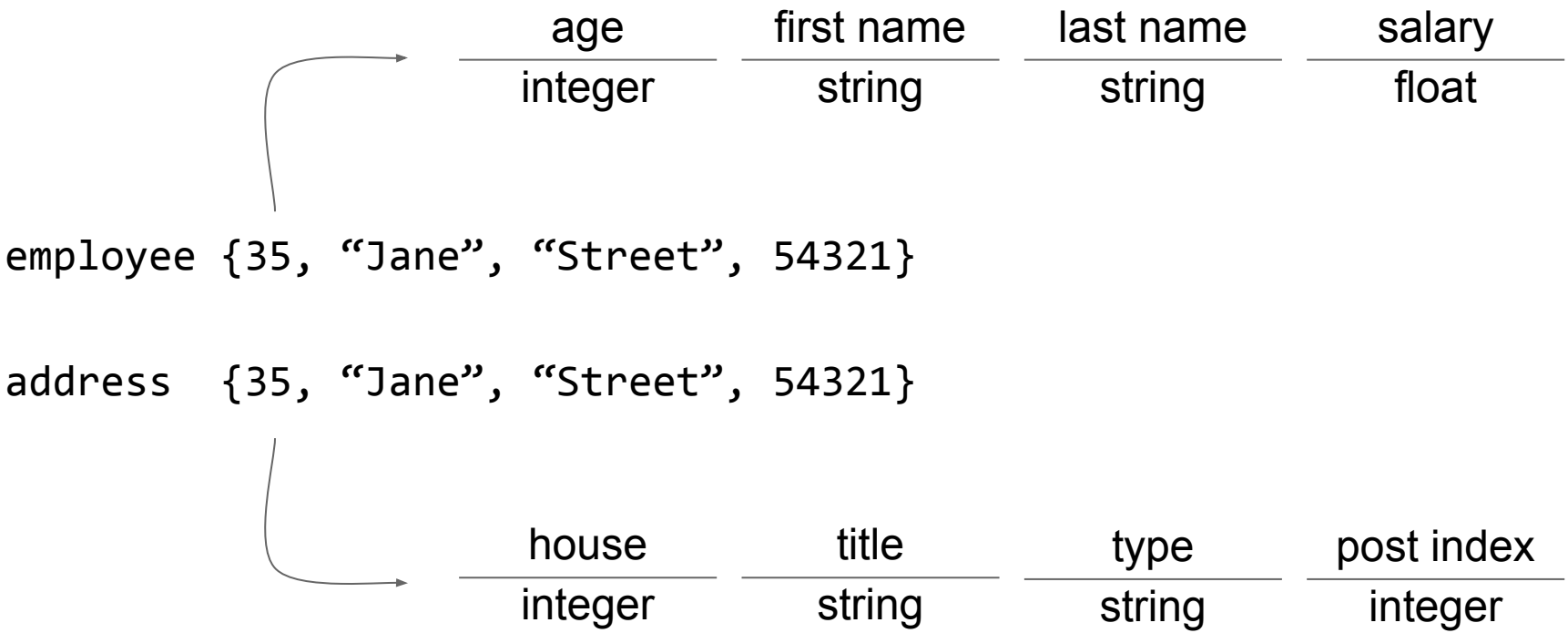
```
ParserT<string> helloUsername =
  bind<string, string>(letters, [=](auto firstName) {
    return bind<string, string>(spaces, [=](auto) {
      return bind<string, string>(letters, [=](auto lastName) {
        return pure<string>(mkHello(firstName, lastName)); }); }); });
```

```
ParserT<string> helloUsername =
  bind<string, string>(letters,       [=](auto firstName) { return
  bind<string, string>(spaces,        [=](auto)           { return
  bind<string, string>(letters,       [=](auto lastName)  { return
  pure<string>(mkHello(firstName, lastName)); }); }); });
```

Dear Santa, I was a good boy last year. Can I have?..

```
ParserT<string> helloUsername = do {
  auto firstName <- letters;
  auto _         <- spaces;
  auto lastName  <- letters;
  return pure<string>(mkHello(firstName, lastName));
};
```

State & context-sensitive grammars


age
integer
first name
string
last name
string
salary
float
employee {35, “Jane”, “Street”, 54321}
address {35, “Jane”, “Street”, 54321}
house
integer
title
string
type
string
post index
integer

Algebraic Data Type

```
struct Employee {
    int age;
    string firstName;
    string lastName;
    double salary;
};
```

```
struct Address {
    int house;
    string title;
    string name;
    int postIndex;
};
```

Algebraic Data Type

```
struct Employee {
    int age;
    string firstName;
    string lastName;
    double salary;
};
```

```
struct Address {
    int house;
    string title;
    string name;
    int postIndex;
};
```

```
using LogEntry = std::variant<Employee, Address>;
```

Algebraic Data Type

```
struct Employee {
   int age;
   string firstName;
   string lastName;
   double salary;
};
```

```
struct Address {
   int house;
   string title;
   string name;
   int postIndex;
};
```

```
using LogEntry = std::variant<Employee, Address>;

function<LogEntry(int a, const string& fn, const string& ln, double s)>
   mkEmployeeEntry;

function<LogEntry(int h, const string& ttl, const string& n, int idx)>
   mkAddressEntry;
```

Putting state

```
struct Config {
   string currentEntry;
};


ParserT<Config, string> logToken = lit("employee") | lit("address");

ParserT<Config, Unit>   setEntry = bind<string, Unit>(
    logToken,
    [](const string& s) {
        return putSt(Config { s });
});
```

Putting state

```
struct Config {
    string currentEntry;
};


ParserT<Config, string> logToken = lit("employee") | lit("address");

ParserT<Config, Unit>   setEntry = bind<string, Unit>(
    logToken,
    [](const string& s) {
        return putSt(Config { s });
});
```

Getting state

```
ParserT<Config, LogEntry> logEntry =
  bind<Config, LogEntry>(getSt, [=](Config cfg) {

    if (cfg.currentEntry == "employee")
        return fmap(mkEmployeeEntry, employeeParser);

    return fmap(mkAddressEntry, addressParser);
});
```

Getting state

```
ParserT<Config, LogEntry> logEntry =
  bind<Config, LogEntry>(getSt, [=](Config cfg) {

    if (cfg.currentEntry == "employee")
        return fmap(mkEmployeeEntry, employeeParser);

    return fmap(mkAddressEntry, addressParser);
});
```

Conclusion

Monadic parser combinators

- Monad: do-notation**, bind, when**, **unless, sequence**, **mapM, guard**
- Applicative: **app**, **pure**
- Alternative: **alt**, **operator|**
- Functor: **fmap**
- Time travelling: **lookForward, lookBackward**
- State handling: **getSt, putSt**
- Error handling: **tryP, safeP, onFail, onSuccess, failWith**

GitHub List: Functional Programming in C++

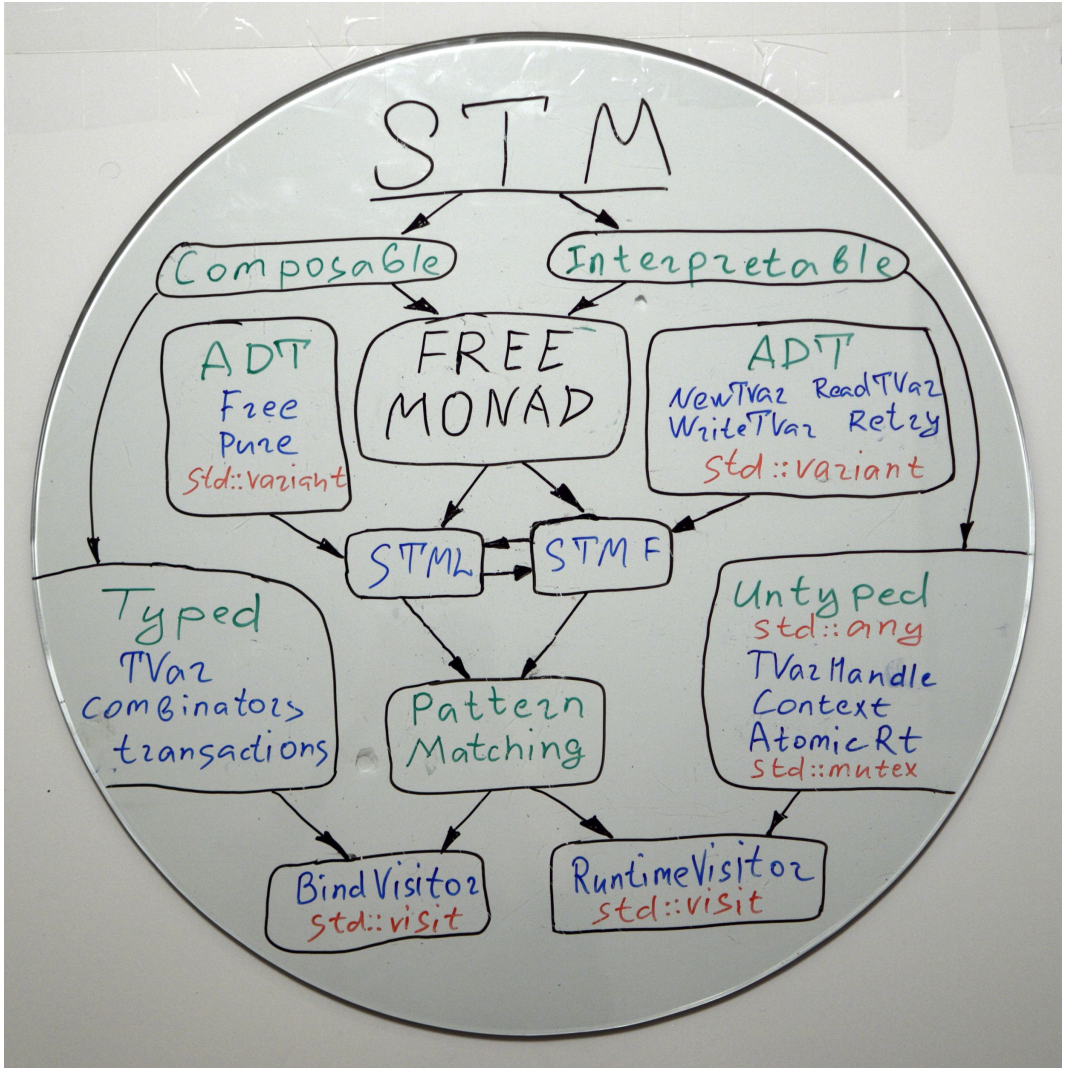
- Books
- Articles
- Papers
- Talks
- QA
- Libraries
- Showcase projects

👉 https://github.com/graninas/cpp_functional_programming

Thanks for watching!

Alexander Granin
graninas@gmail.com
Twitter: @graninas

C++ Russia 2019, Moscow


STM
Composable
Interpretable
ADT
Free
Pure
std::variant
FREE
MONAD
ADT
NewTVar ReadTVar
WriteTVar Retry
std::variant
STML
STMF
Typed
TVar
combinators
transactions
Pattern
Matching
Untyped
std::any
TVarHandle
Context
AtomicRt
std::mutex
BindVisitor
std::visit
RuntimeVisitor
std::visit