New C++ features for writing DSLs

CoreHard 2019, Minsk

dr Ivan Čukić

ivan@cukic.co
http://cukic.co

About me

- Independent trainer / consultant
- KDE developer
- Author of the ”Functional Programming in C++” book
- University lecturer

Disclaimer

> Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.
>
> — Philip Wadler

INTRODUCTION

Introduction

```
select name from participants;

<expr> ::= <var> | <lit> | <expr> <op> <expr>

[a-zA-Z][a-zA-Z0-0_]*
```

DSLs and C++

Limited:

- `.something` syntax
- Operators
- Braces and parentheses

Introduction

Introduction

<_/_~~*~~,,,,,___/,,,>

BASICS

Basics first

```
user.name = ”Martha”;
user.surname = ”Jones”; // exception!
user.age = 42;
```

Copy-and-swap

```
type& operator=(const type& value)
{
    auto tmp = value;
    tmp.swap(*this);
    return *this;
}
```

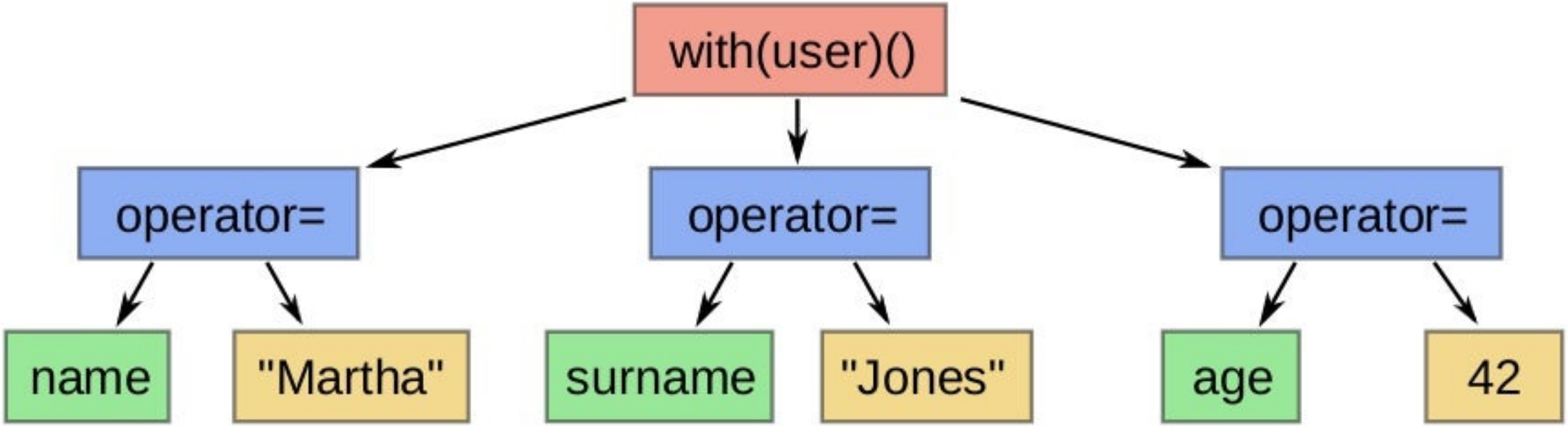
Basics first

```
auto tmp = user;
tmp.name = ”Martha”;
tmp.surname = ”Jones”;
tmp.age = 42;
tmp.swap(user);
```

Basics first

```
with(user) (
    name = ”Martha”,
    surname = ”Jones”,
    age = 42
);
```

Basics first



Basics first

```
class transaction {
public:
    transaction(user_t& user)
        : m_user{user}
    {}

    void operator() (…)
    {
        …
    }

private:
    user_t& m_user;
};
```

Defines the object the transaction will operate on

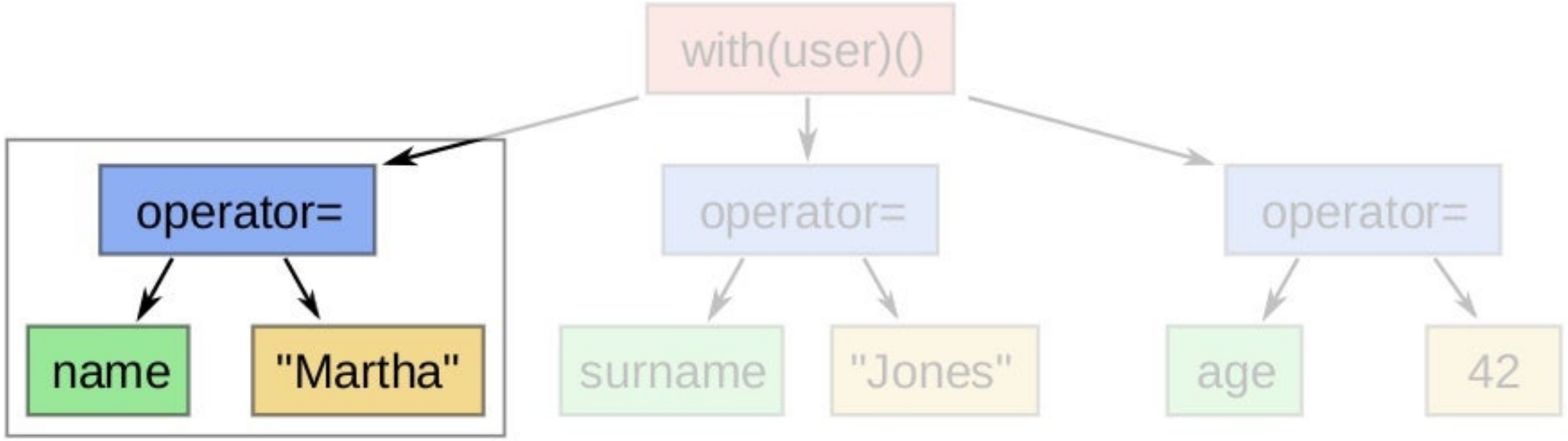
A reference to the object

Basics first

```
class transaction {
public:
    transaction(user_t& user)
        : m_user{user}
    {}

    void operator() (…)
    {
        …
    }

private:
    user_t& m_user;
};
```

Call operator takes a list of actions to perform

Basics first

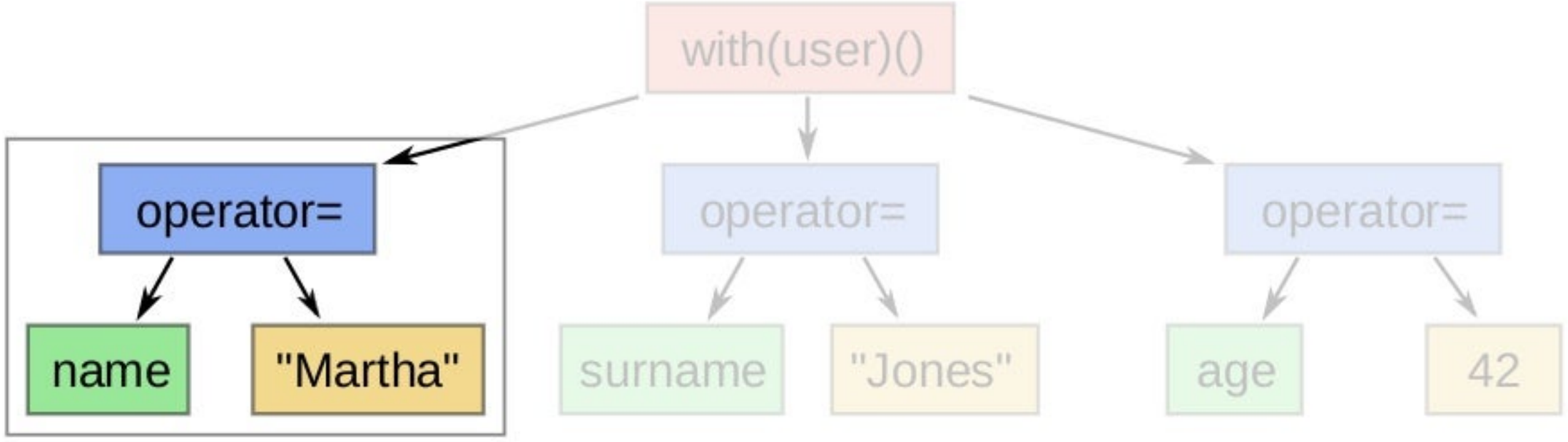


Basics first

```
template <typename Member, typename Value>
struct update {
    update(Member member, Value value)
        : member{member}
        , value{std::move(value)}
    {
    }

    Member member;
    Value value;
};
```

Note that the update structure does not know which object it will be updating.

Basics first



Basics first

```
template <typename Member>
struct field {
    field(Member member)
        : member{member}
    {
    }

    template <typename Value>
    update<Member, Value> operator=(Value&& value) const
    {
        return update{member, FWD(value)};
    }

    Member member;
};
```

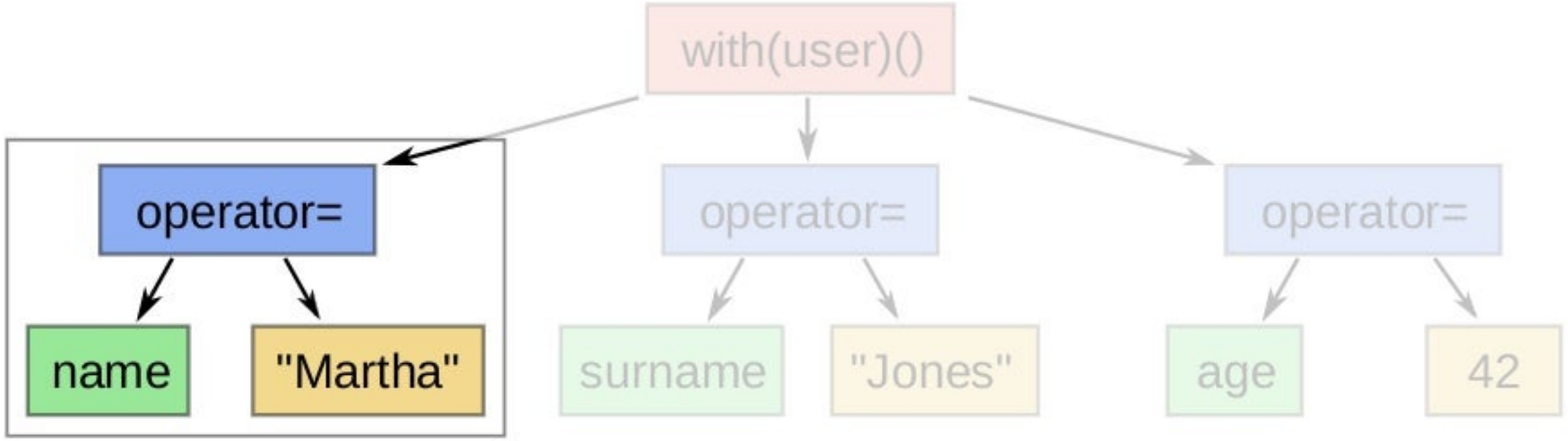
Basics first

```
field name{…};
field surname{…};
field age{…};
```

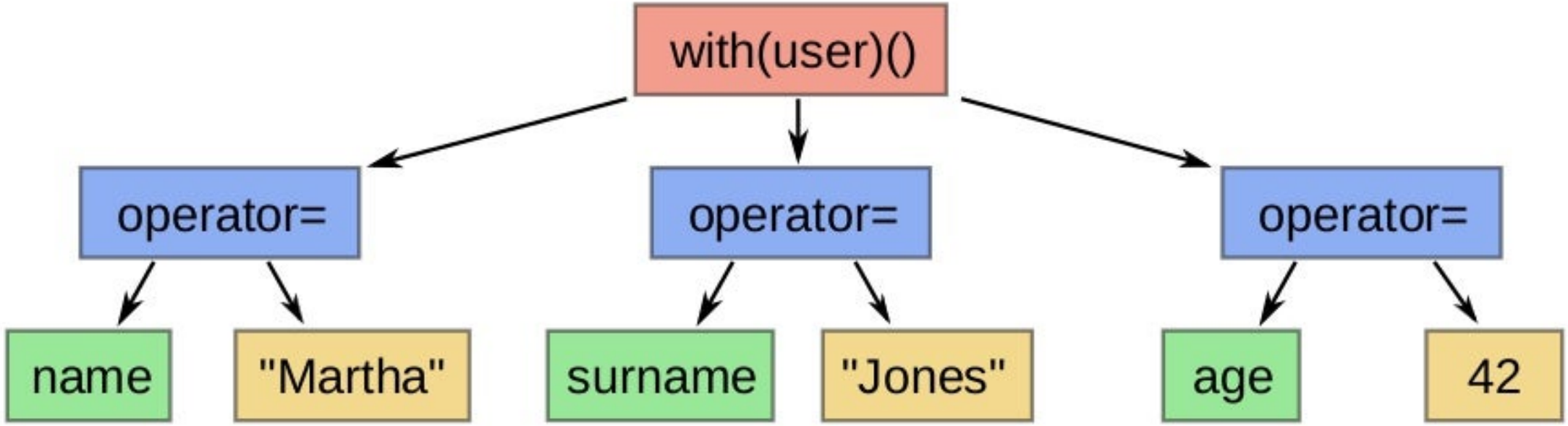
Basics first

```
#define field(name) field_impl([] (auto& object) -> auto& { \
    return object.name; \
})

with(user) (
    field(name) = "Martha",
    field(surname) = "Jones",
    field(age) = 42
);
```

Basics first



Basics first



Transaction

We'll model a simple transaction concept:

- The update action is activated using the call operator:
- Each update action returns a `bool` indicating success or failure.

```
auto update_action{name = "Martha"};

if (update_action(user)) {
    // success
}
```

Transaction

```
class transaction {
public:
    template <typename... Updates>
    bool operator( ) (Updates&&... updates)
    {
        auto temp = m_user;

        }
};
```

> Invoke each update action on the temp object, and swap temp and *this only if they all succeeded

Transaction

```
class transaction {
public:
    template <typename... Updates>
    bool operator( ) (Updates&&... updates)
    {
        auto temp = m_user;

        if ((... && FWD(updates)(temp))) {
            temp.swap(m_user);
            return true;
        }

        return false;
    }
};
```

Basics first

Ideal simple DSL:

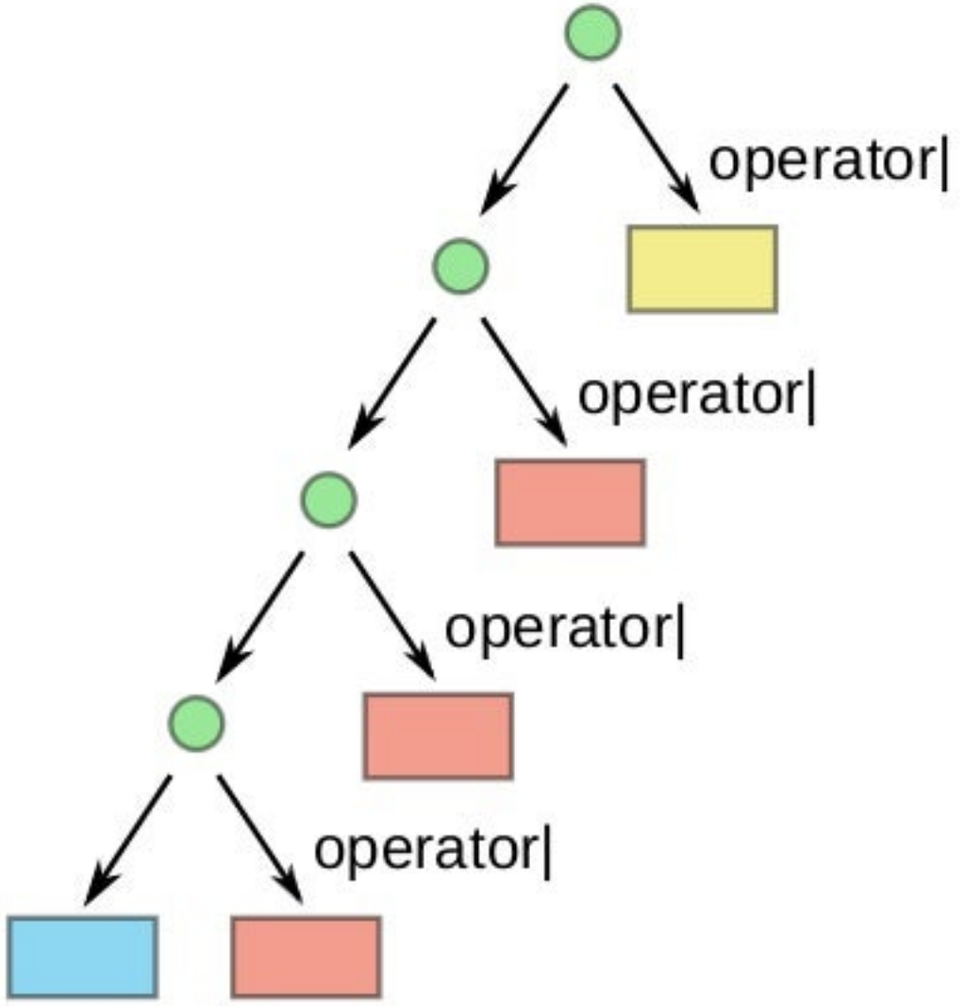
- Context-free
- AST that fits semantics
- Uses only simple constructs

CONTEXT

Syntax

```
service(42042)
        | transform(trim)
        | remove_if(&std::string::empty)
        | filter([] (const std::string& message) {
            return message[0] != '#';
          })
        | sink_to_cerr;
```

Syntax



Syntax

```
template <typename... Nodes>
class expression {
    template <typename Continuation>
    auto operator| (Continuation&& cont) &&
    {
        …
    }
};
```

Syntax

`std::function`: type erasure is cool but slow.

Use a right-associative operator »= to appease Haskell gods?

Syntax

`std::function`: type erasure is cool but slow.

Expression templates to the rescue!

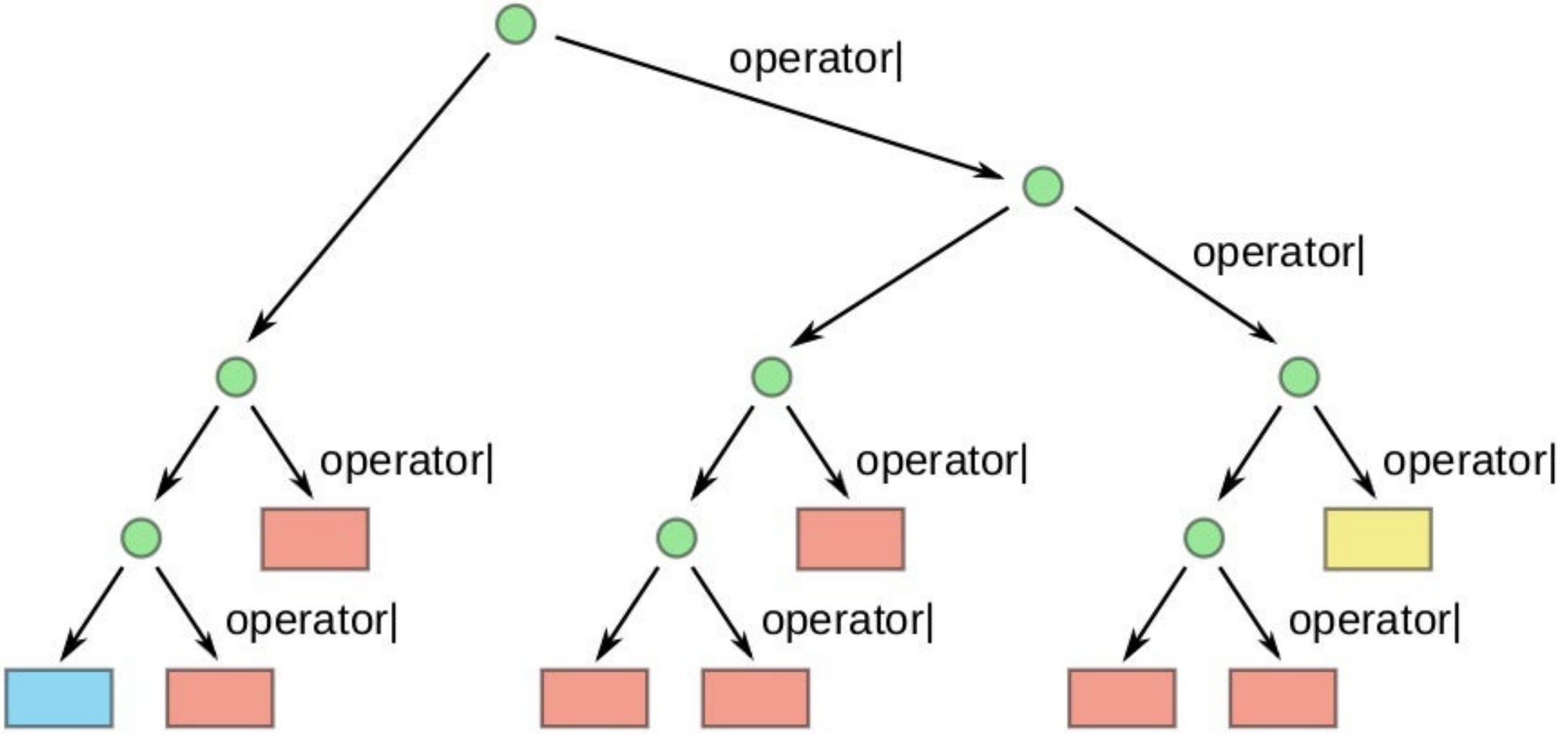
Syntax

```
template <typename... Nodes>
class expression {
    template <typename Continuation>
    auto operator| (Continuation&& cont) &&
    {
        return expression(
            std::tuple_cat(
                std::move(m_nodes),
                std::make_tuple(FWD(cont))));
    }

    std::tuple<Nodes...> m_nodes;
};
```

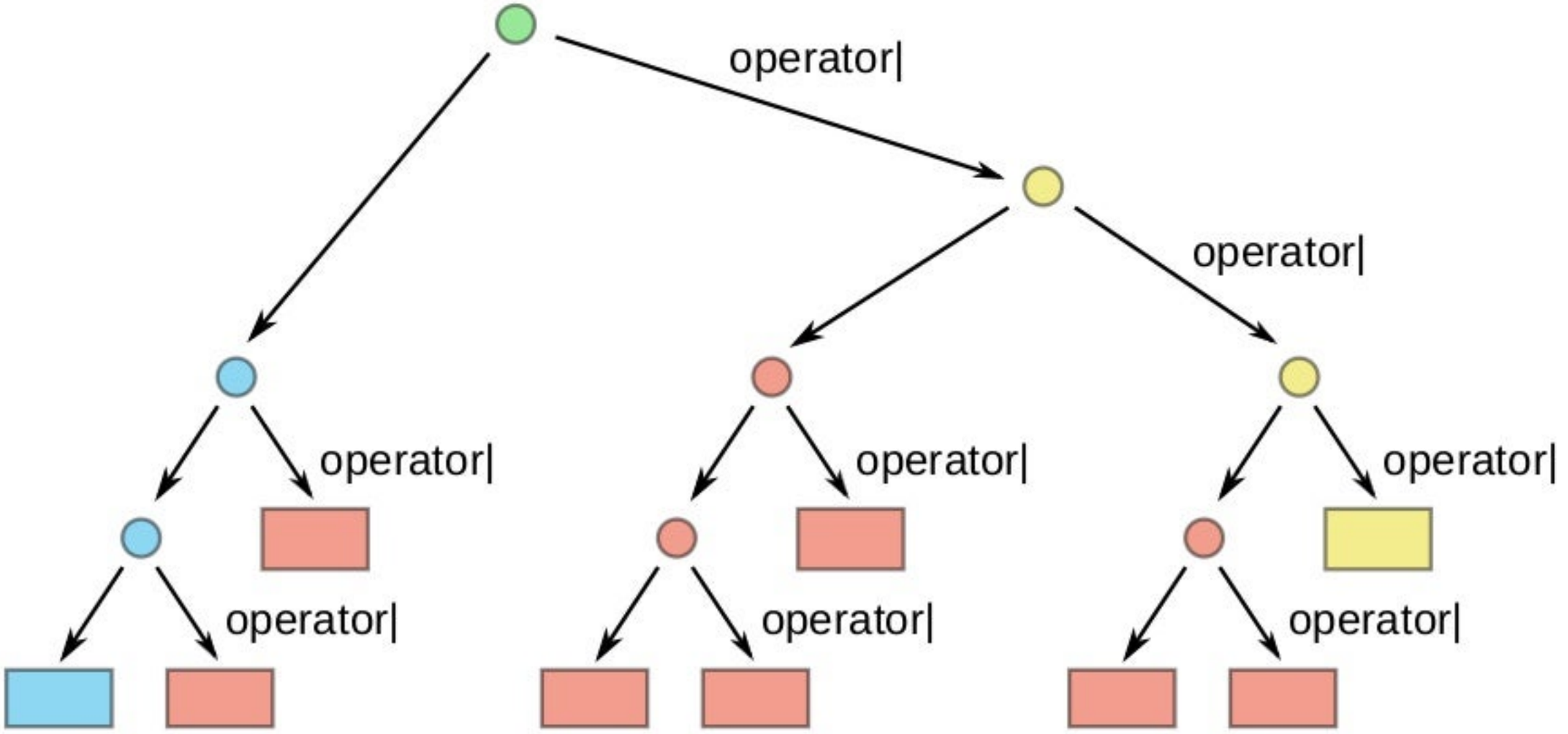
Syntax

```
auto user_names = users | transform(&user_t::name);
auto ignore_empty = transform(trim)
                  | remove_if(&std::string::empty);

user_names | ignore_empty | transform(string_to_upper);
```

Syntax



Syntax



Syntax

- Different meanings of `operator|`
- Wildly different types of operands (no inheritance tree)
- Arbitrary complex AST

Universal expression

```
template <typename Left, typename Right>
struct expression {
    Left left;
    Right right;
};
```

```
<node> ::= <producer> | <consumer> | <trafo> | <expression>
<expression> ::= <node> <|> <node>
```

Meta information

Adding meta-information to classes:

```
struct producer_node_tag {};
struct consumer_node_tag {};
struct transformation_node_tag {};

class filter_node {
public:
    using node_type_tag =
        transformation_node_tag;
};
```

Meta information

```
template <typename Node>
using node_category =
    typename remove_cvref_t<Node>::node_type_tag;
```

Universal expression

```
template <typename Tag, typename Left, typename Right>
struct expression {
    using node_type_tag = Tag;

    Left left;
    Right right;
};
```

Meta information

```
template < typename Node
         , typename Category =
                std::detected_t<node_category, Node>
constexpr bool is_node()
{
    if constexpr (!is_detected_v<node_category, Node>) {
        return false;

    } else if constexpr (
            std::is_same_v<complete_pipeline_tag, Category>) {
        return false;

    } else {
        return true;
    }
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{
    …
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{
    if constexpr (!is_producer<Left> && !is_consumer<Right>) {
        return expression<transformation_node_tag, Left, Right>{
            FWD(left), FWD(right)
        };
    }


    …
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{
    … else
    if constexpr (is_producer<Left> && !is_consumer<Right>) {
        return expression<producer_node_tag, Left, Right>{
            FWD(left), FWD(right)
        };
    }

    …
}
```

Restricting the pipe

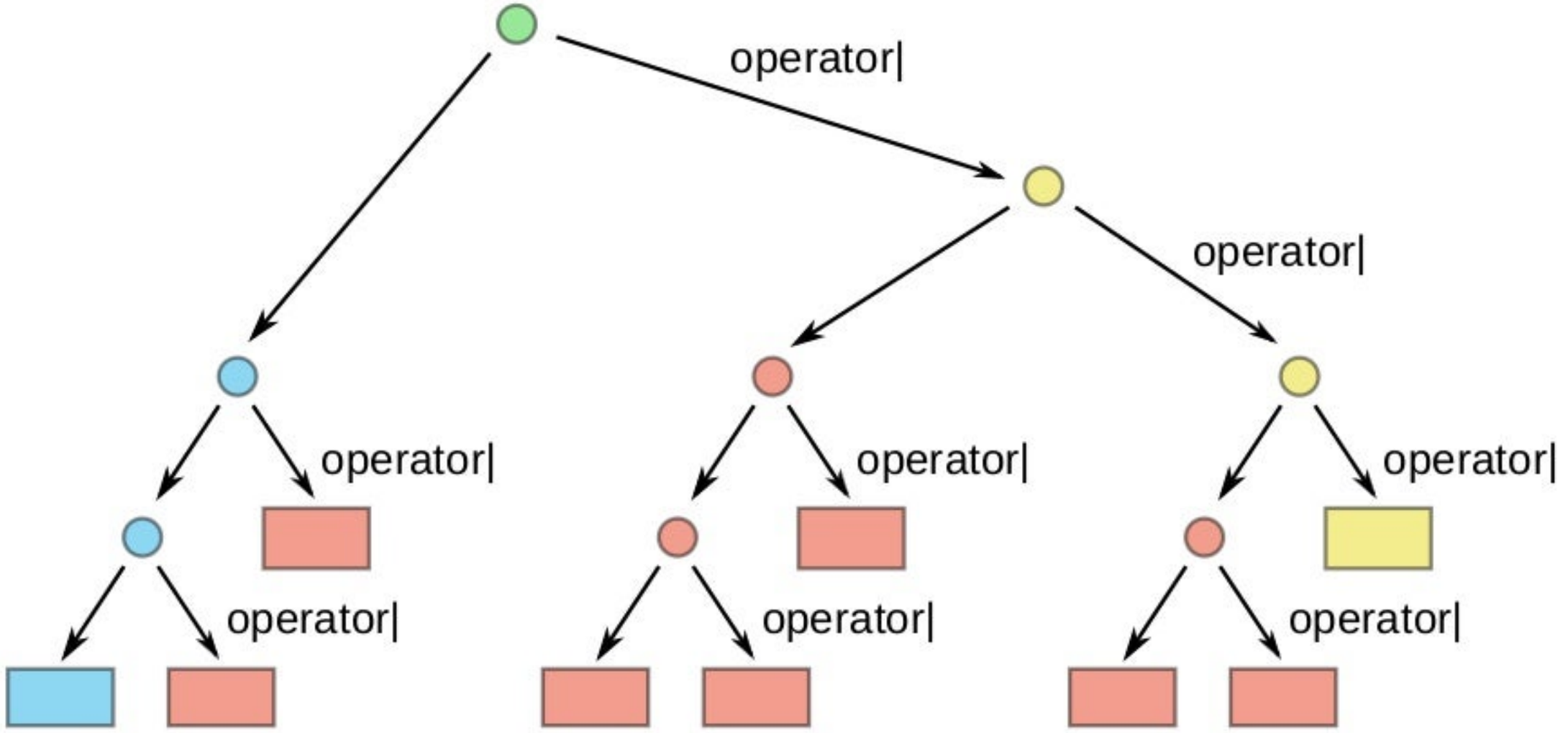
```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{

    … else
    if constexpr (!is_producer<Left> && is_consumer<Right>) {
        return expression<consumer_node_tag, Left, Right>{
            FWD(left), FWD(right)
        };
    }
    …
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{


    … else
    if constexpr (is_producer<Left> && is_consumer<Right>) {
        return expression<complete_pipeline_tag, Left, Right>{
            FWD(left), FWD(right)
        };
    }
}
```

EVALUATION

Evaluation



AST transformation

1. Collect nodes from the left sub-tree
2. Collect nodes from the right sub-tree
3. Merge the results

AST transformation

```
template <typename Expr>
auto collect_nodes(Expr&& expr)
{
    auto collect_sub_nodes = [] (auto&& sub) {
        if constexpr (is_expression<decltype(sub)>) {
            return collect_nodes(std::move(sub));
        } else {
            return std::make_tuple(std::move(sub));
        }
    };

    return std::tuple_cat(
        collect_sub_nodes(std::move(expr.left)),
        collect_sub_nodes(std::move(expr.right)));
}
```

Evaluation

Two choices:

- Connect left-to-right
- Connect right-to-left

LTR

Pros:

- Easier
- Easy to pass `value_type` around

Cons:

- Type erasure

RTL

Pros:

- No need for type erasure

Cons:

- No way to pass `value_type`:

```
service(42042) | debounce<std::string>(200ms) | ...
```

Both!

Feed forward and backward connect.

Context propagation

```
struct transform_t {

    template <typename In>
    using value_type_for_input_t = …

};
```

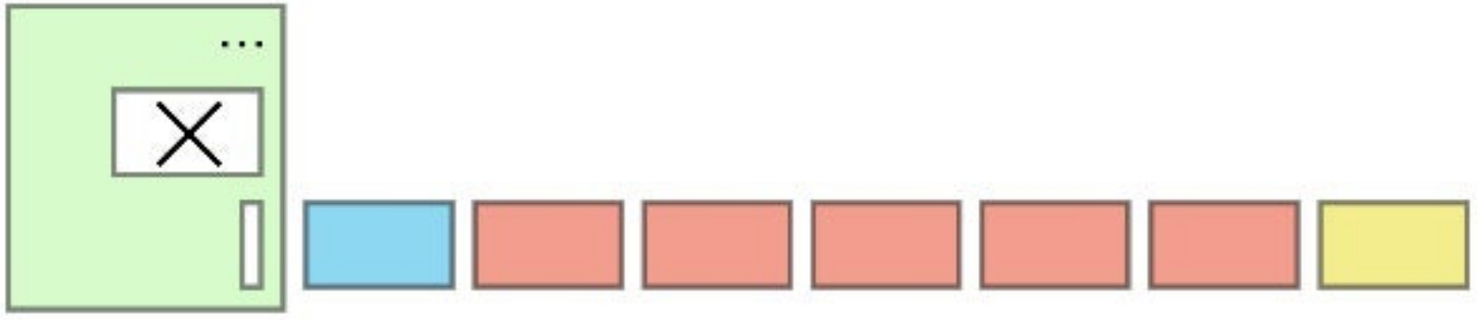
Context propagation

```
using new_value_type =
    typename Data::template value_type_for_input_t<ValueType>;
```

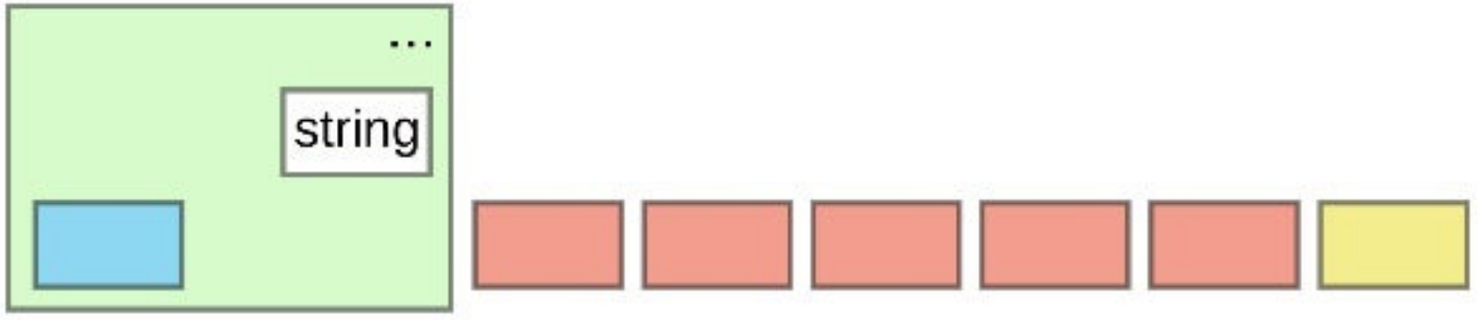
Context propagation

```
template <typename LastValueType,
          typename... ProcessedNodes>
struct folding_context {
    …
};
```

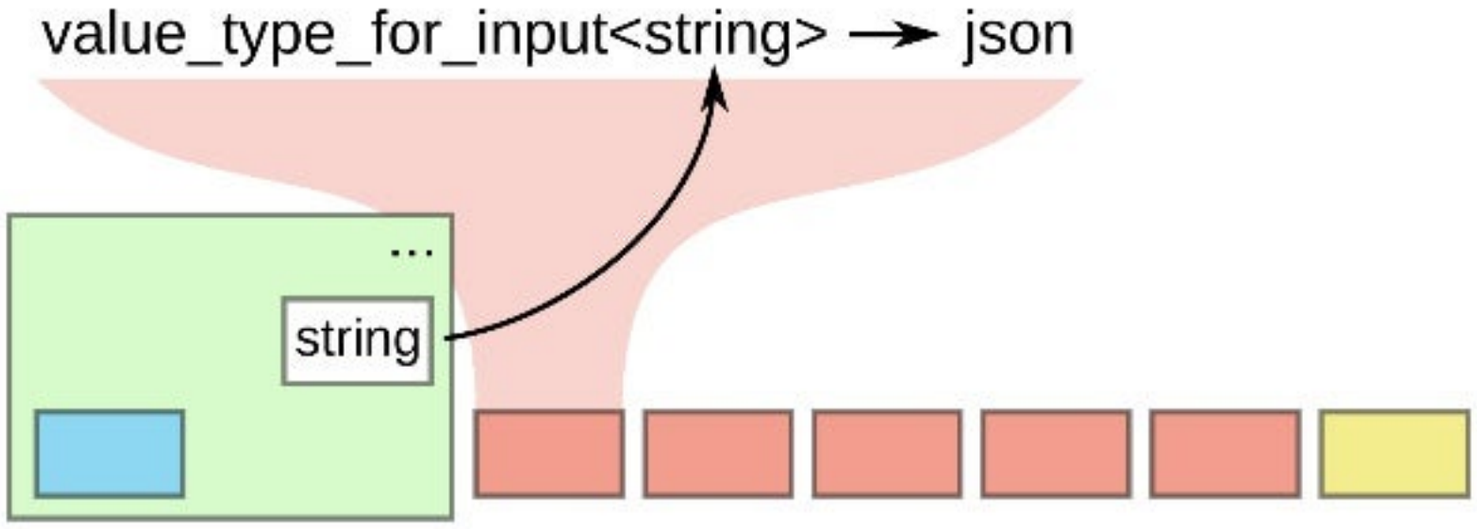
Context propagation



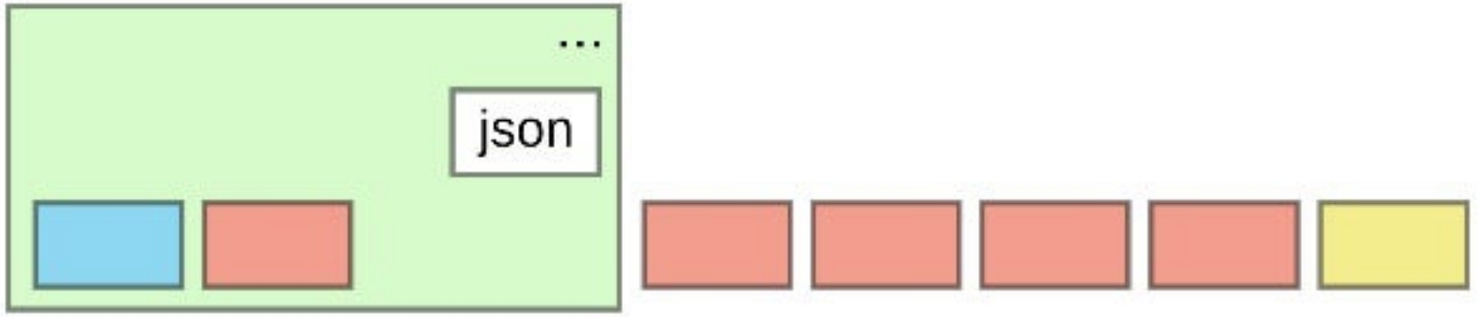
Context propagation



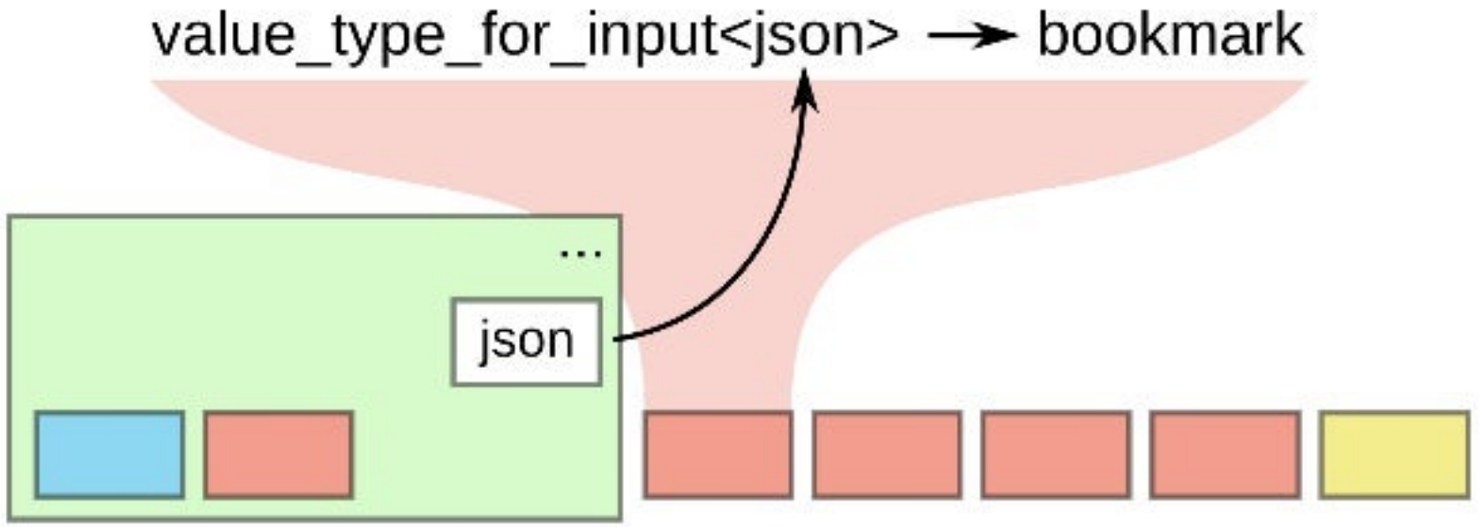
Context propagation



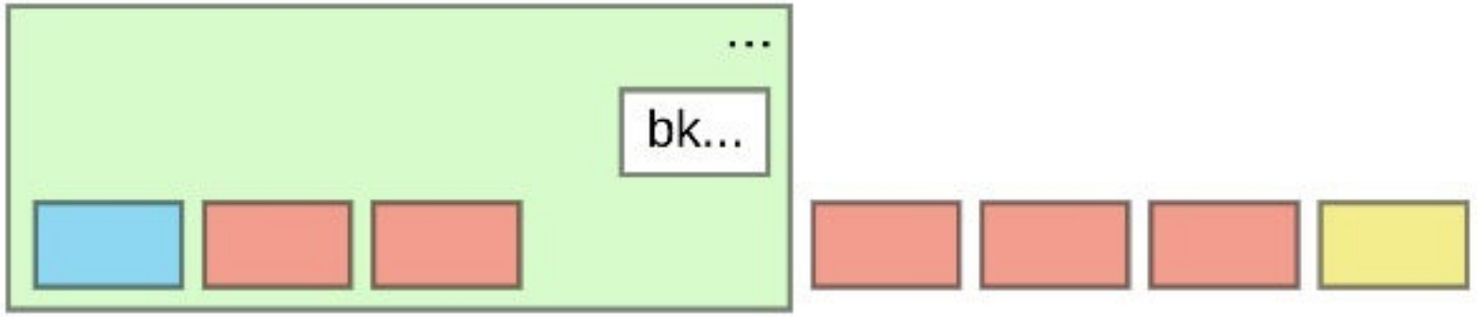
Context propagation



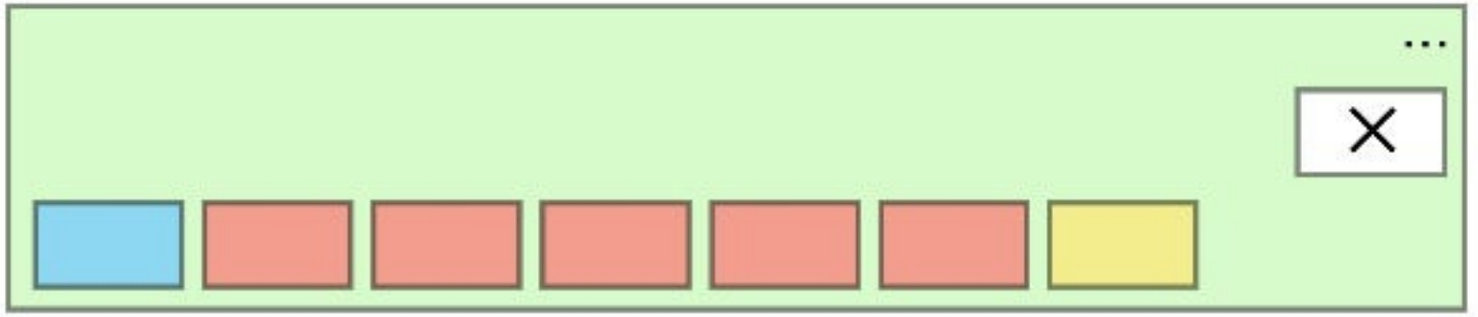
Context propagation



Context propagation



Context propagation



Context propagation

Context propagation

```
template <typename... Nodes>
auto propagate_value_type(Nodes&& ...nodes)
{
    return (init_context() % ... % FWD(nodes)).nodes;
}
```

Context propagation

```
template <typename NodesTuple>
auto propagate_value_type(NodesTuple&& nodes)
{
    return (init_context() % ... % ???).nodes;
}
```

Context propagation

```
template <typename NodesTuple,
          size_t... Idx>
auto propagate_value_type(NodesTuple&& nodes, std::index_sequence<Idx...>)
{
    return (init_context() % ... %
        std::get<Idx>(FWD(nodes))).nodes;
}
```

Context propagation

```
propagate_value_type(
        nodes,
        std::make_index_sequence<std::tuple_size_v<Tuple>>()
    );
```

Connection

Now we have a list of enriched nodes,
we can connect them right-to-left.

Evaluation

```
template <typename... Nodes>
auto evaluate_nodes(Nodes&&... nodes)
{
    return (... % nodes);
}
```

Connection

```
template <typename Node, typename Connected>
auto operator% (Node&& new_node, Connected&& connected)
{
    return FWD(new_node).with_continuation(FWD(connected));
}
```

BEST PRACTICES

Asserts

```
#define assert_value_type(T)                                    \
    static_assert(                                              \
        std::is_same_v<T, std::remove_cvref_t<T>>,              \
        "This is not a value type")
```

Printf debugging

```
template <typename... Types>
class print_types;

print_types<std::vector<bool>::reference>{};

error: incomplete type 'class print_types<std::_Bit_reference>'
```

Printf debugging

```
template <typename... Types>
[[deprecated]] class print_types;
```

Printf debugging

For complex expression template types, create a sanitization script:

- `basic_string...` → `string`
- `transformation_node_tag` → `TRAFO`

Change all < and > into (and) and pass the output through `clang-format`.

Printf debugging

```
expression(
    expression(
        void,
        expression(PRODUCER,
                   expression(PRODUCER, ping_process,
                              transform("(λ tests_multiprocess.cpp:91:26)")),
                   transform("(λ tests_multiprocess.cpp:82:38)"))),
    expression(
        TRAFO,
        expression(TRAFO,
                   expression(TRAFO,
                              expression(TRAFO, identity_fn,
                                         transform("(λ tests_multiprocess.cpp:99:

        …
```

Answers? Questions! Questions? Answers!

Kudos (in chronological order):

Friends at **KDE**
Saša Malkov and **Zoltan Porkolab**
Сергей Платонов

cukic.co/to/fp-in-cpp
Functional Programming in C++