

# MPark.Patterns

Pattern Matching in C++

<https://github.com/mpark/patterns>

**Michael Park**



@mpark

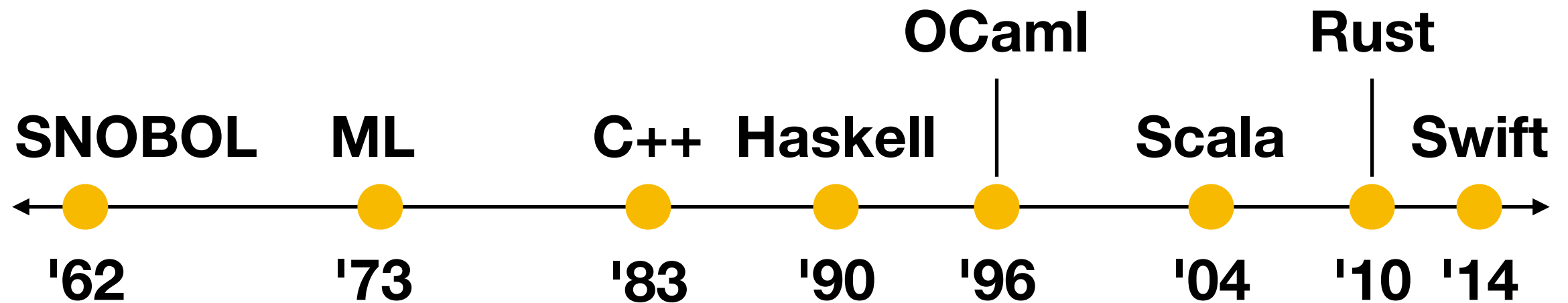


@mcypark



MESOSPHERE

# History



- [Open Pattern Matching for C++ / Mach7](#) (2013)  
Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup
- [Simple, Extensible C++ Pattern Matching Library](#) (2015)  
John Bandela

**Why doesn't C++  
have feature X?**

**No one proposed it**

# Standards Proposal

- [P0095R1: Pattern Matching and Language Variants](#) (2016)  
David Sankel
- Next Revision: Pattern Matching (TBD)  
David Sankel, Michael Park

# Purpose

- Familiarize pattern matching in the C++ community
- Why now? What's changed?
- Can a modern library solution be "good enough"?
  - If not, gain experience to guide the language design

# Overview

- Algebraic Data Types
- What is Pattern Matching?
- Various Forms of Pattern Matching in C++
- MPark.Patterns
- Other Interesting Patterns

# Algebraic Data Types



# Algebraic Data Types

|         | Description                  | Example               | # of Possible States |
|---------|------------------------------|-----------------------|----------------------|
| Product | one of X <b>AND</b> one of Y | <b>tuple</b> <X, Y>   | $ X  \times  Y $     |
| Sum     | one of X <b>OR</b> one of Y  | <b>variant</b> <X, Y> | $ X  +  Y $          |

**Pattern matching is the best tool for decomposing  
Algebraic Data Types**

# What is Pattern Matching?

“In pattern matching, we attempt to **match** *values* against *patterns* and, if so desired, **bind** *variables* to successful matches.”

[https://en.wikibooks.org/wiki/Haskell/Pattern\\_matching](https://en.wikibooks.org/wiki/Haskell/Pattern_matching)

“In pattern matching, we attempt to **match** *values* against *patterns* and, if so desired, **bind** *variables* to successful matches.”

```
struct Point { x: i32, y: i32 }

let p = Point { x: 7, y: 0 };

match p {
    Point { x: 0, y      } => println!("Y axis: {}", y),
    Point { x      , y: 0 } => println!("X axis: {}", x),
    Point { x      , y      } => println!("{}", {}, x, y)
}

// prints: "X axis: 7"
```

[https://en.wikibooks.org/wiki/Haskell/Pattern\\_matching](https://en.wikibooks.org/wiki/Haskell/Pattern_matching)

Pattern matching is a **declarative** approach in lieu of manually **testing** for a *value* with a *sequence of conditionals* and **extracting** the *desired components*.

```
struct Point { int x; int y; };
```

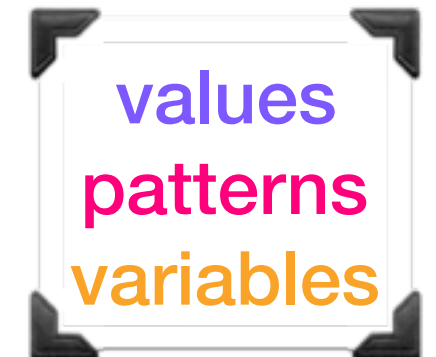
```
auto p = Point { 7, 0 };
```

```
if (p.x == 0)      printf("Y axis: %d\n", p.y);  
else if (p.y == 0) printf("X axis: %d\n", p.x);  
else              printf("%d, %d\n", p.x, p.y);
```

```
// prints: "X axis: 7"
```

# Evaluating Expressions

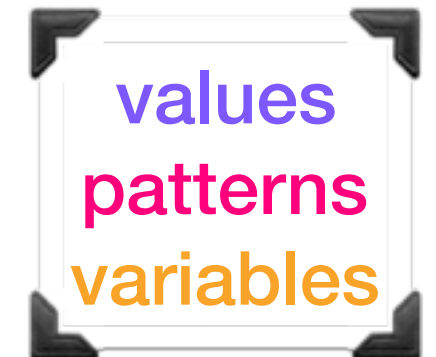
```
enum Expr {  
  Int(i32),  
  Neg(Box<Expr>),  
  Add(Box<Expr>, Box<Expr>),  
  Mul(Box<Expr>, Box<Expr>),  
}
```



```
fn eval(expr: Expr) -> i32 {  
  return match expr {  
    Expr::Int(value) => value,  
    Expr::Neg(expr) => -eval(*expr),  
    Expr::Add(lhs, rhs) => eval(*lhs) + eval(*rhs),  
    Expr::Mul(lhs, rhs) => eval(*lhs) * eval(*rhs),  
  };  
}
```

# Evaluating Expressions

```
enum Expr {  
  Int(i32),  
  Neg(Box<Expr>),  
  Add(Box<Expr>, Box<Expr>),  
  Mul(Box<Expr>, Box<Expr>),  
}
```



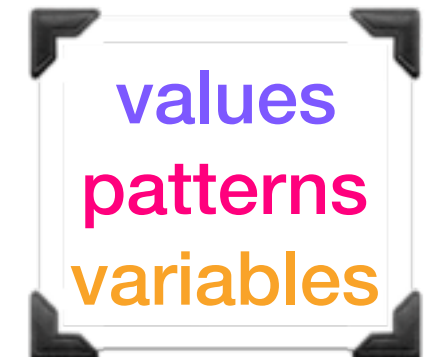
```
fn eval(expr: Expr) -> i32 {  
  return match expr {  
    Expr::Int(value) => value,  
    Expr::Neg(expr)  => -eval(*expr),  
    Expr::Add(lhs, rhs) => eval(*lhs) + eval(*rhs),  
    Expr::Mul(lhs, rhs) => eval(*lhs) * eval(*rhs),  
  };  
}
```

↑  
sum  
↓



# Evaluating Expressions

```
enum Expr {  
  Int(i32),  
  Neg(Box<Expr>),  
  Add(Box<Expr>, Box<Expr>),  
  Mul(Box<Expr>, Box<Expr>),  
}
```



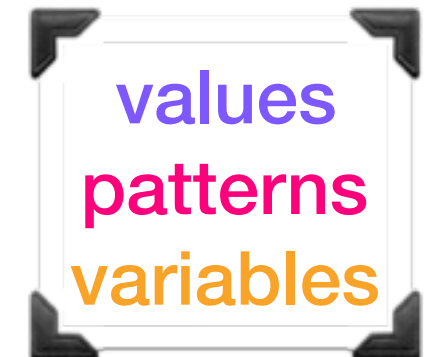
```
fn eval(expr: Expr) -> i32 {  
  return match expr {  
    Expr::Int(value) => value,  
    Expr::Neg(expr)  => -eval(*expr),  
    Expr::Add(lhs, rhs) => eval(*lhs) + eval(*rhs),  
    Expr::Mul(lhs, rhs) => eval(*lhs) * eval(*rhs),  
  };  
}
```

↑  
sum  
↓

← product →

# Evaluating Expressions

```
enum Expr {  
  Int(i32),  
  Neg(Box<Expr>),  
  Add(Box<Expr>, Box<Expr>),  
  Mul(Box<Expr>, Box<Expr>),  
}
```



```
fn eval(expr: Expr) -> i32 {  
  return match expr {  
    Expr::Int(value) => value,  
    Expr::Neg(expr) => -eval(*expr),  
    Expr::Add(lhs, rhs) => eval(*lhs) + eval(*rhs),  
    Expr::Mul(lhs, rhs) => eval(*lhs) * eval(*rhs),  
  };  
}
```

**Composed Patterns!**

# Evaluating Expressions

```
struct Expr { virtual ~Expr() = default; };
struct Int : Expr { int value; };
struct Neg : Expr { shared_ptr<Expr> expr; };
struct Add : Expr { shared_ptr<Expr> lhs, rhs; };
struct Mul : Expr { shared_ptr<Expr> lhs, rhs; };
```

```
int eval(const Expr &expr) {
    if (auto p = dynamic_cast<const Int *>(&expr))
        return p->value;
    if (auto p = dynamic_cast<const Neg *>(&expr))
        return -eval(*p->expr);
    if (auto p = dynamic_cast<const Add *>(&expr))
        return eval(*p->lhs) + eval(*p->rhs);
    if (auto p = dynamic_cast<const Mul *>(&expr))
        return eval(*p->lhs) * eval(*p->rhs);
    throw logic_error("unknown expression");
}
```

# Evaluating Expressions

Manually **testing** for a *value* with a *sequence of conditionals* and **extracting** the *desired components*.

```
int eval(const Expr &expr) {  
    if (auto p = dynamic_cast<const Int *>(&expr))  
        return p->value;  
    if (auto p = dynamic_cast<const Neg *>(&expr))  
        return -eval(*p->expr);  
    if (auto p = dynamic_cast<const Add *>(&expr))  
        return eval(*p->lhs) + eval(*p->rhs);  
    if (auto p = dynamic_cast<const Mul *>(&expr))  
        return eval(*p->lhs) * eval(*p->rhs);  
    throw logic_error("unknown expression");  
}
```

# LLVM

Manually **testing** for a *value* with a *sequence of conditionals* and **extracting** the *desired components*.

```
if (const auto *CE = dyn_cast<ImplicitCastExpr>(E))  
    return // ...  
if (const auto *RE = dyn_cast<DeclRefExpr>(E))  
    return // ...  
if (const auto *ME = dyn_cast<MemberExpr>(E))  
    return // ...  
// ...  
if (const auto *CE = dyn_cast<CallExpr>(E))  
    return // ...  
if (const auto *CE = dyn_cast<CXXConstructExpr>(E))  
    return // ...
```

# Visitor

```
struct Int; struct Neg; struct Add; struct Mul;

struct Expr {
    struct Vis {
        virtual void operator()(const Int &) const = 0;
        virtual void operator()(const Neg &) const = 0;
        virtual void operator()(const Add &) const = 0;
        virtual void operator()(const Mul &) const = 0;
    };

    virtual ~Expr() = default;
    virtual void accept(const Vis&) const = 0;
};

struct Int : Expr {
    void accept(const Vis& vis) const { vis(*this); }
    int value;
};

struct Neg : Expr {
    void accept(const Vis& vis) const { vis(*this); }
    shared_ptr<Expr> expr;
};

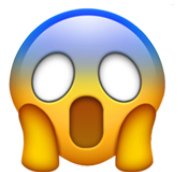
struct Add : Expr {
    void accept(const Vis& vis) const { vis(*this); }
    shared_ptr<Expr> lhs, rhs;
};

struct Mul : Expr {
    void accept(const Vis& vis) const { vis(*this); }
    shared_ptr<Expr> lhs, rhs;
};
```

```
int eval(const Expr &expr) {
    struct Eval : Expr::Vis {
        void operator()(const Int &that) const {
            result = that.value;
        }
        void operator()(const Neg &that) const {
            result = -eval(*that.expr);
        },
        void operator()(const Add &that) const {
            result = eval(*that.lhs) + eval(*that.rhs);
        }
        void operator()(const Mul &that) {
            result = eval(*that.lhs) * eval(*that.rhs);
        }
    };

    int &result;

    int result;
    expr.accept(Eval{result});
    return result;
}
```



# Insight from Swift

**“Pattern matching was probably a foregone conclusion, but I wanted to spell out that having ADTs in the language is what really forces our hand because the alternatives are so bad.”**

<http://apple-swift.readthedocs.io/en/latest/Pattern%20Matching.html>

# Various Forms of Pattern Matching in C++



# Various Forms of Pattern Matching in C++

- Matching Simple Types
- Matching Product Types
- Matching Sum Types

# Matching Simple Types

switch

```
int x = 1;

switch (x) {
    case 1: case 2: printf("one or two\n"); break;
    case 3:         printf("three\n");         break;
    default:        printf("anything\n");
}
```

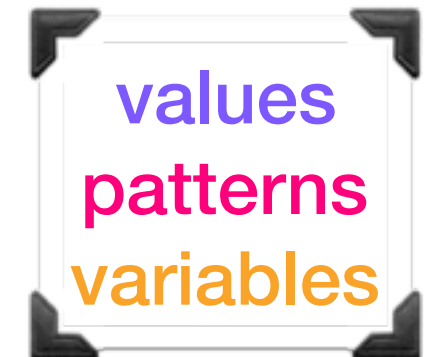
Python  
Style

```
string s = "c";

unordered_map<string, void (*)(>>{
    { "a", [] { printf("A\n"); } },
    { "b", [] { printf("B\n"); } },
    { "c", [] { printf("C\n"); } }
}[s]()
```

# Matching Product Types

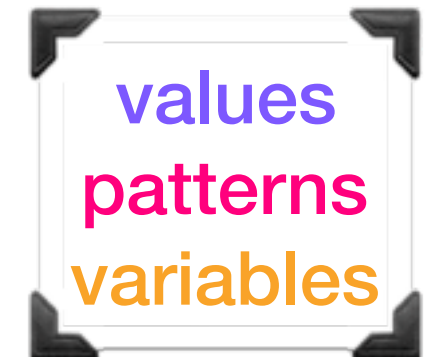
```
pair<X, Y> p;  
tuple<pair<X, Y>, Z> t;
```



|                    | Destructuring   | Nested Destructuring  |
|--------------------|---|---|
| apply              | <pre>apply( [](X x, Y y) {<br/>    // ...<br/>}, p );</pre> | <pre>apply(<br/>    [](pair&lt;X, Y&gt; p, Z z) {<br/>        apply( [z](X x, X y) {<br/>            // ...<br/>        }, p );<br/>    }, t );</pre> |
| Structured Binding | <pre>auto [x, y] = p;</pre>                                 | <pre>auto [p, z] = t;<br/>auto [x, y] = p;<br/><del>auto [[x, y], z] = t;</del></pre>   |

# Matching Sum Types

```
struct Expr;
struct Neg { shared_ptr<Expr> expr; };
struct Add { shared_ptr<Expr> lhs, rhs; };
struct Mul { shared_ptr<Expr> lhs, rhs; };
struct Expr : variant<int, Neg, Add, Mul> {
    using variant::variant;
};
```



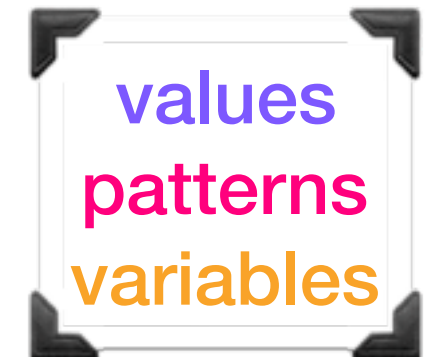
```
int eval(const Expr &expr) {
    return visit(overload(
        [](int value) { return value; },
        [](const Neg &n) { return -eval(*n.expr); },
        [](const Add &a) { return eval(*a.lhs) + eval(*a.rhs); },
        [](const Mul &m) { return eval(*m.lhs) * eval(*m.rhs); })),
    expr);
}
```

# MPark.Patterns

# Main Goals

- Declarative
- Structured
- Cohesive
- Composable

# Basic Structure



```
#include <mpark/patterns.hpp>
```

```
using namespace mpark::patterns; // omitted from here on
match(<expr>...)(
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    // ...
);
```

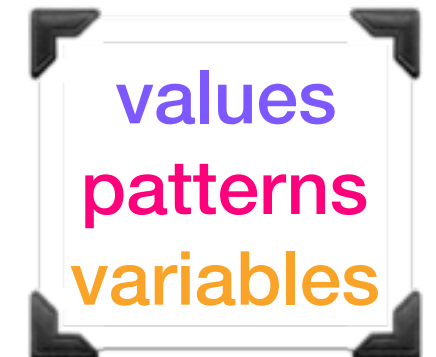
# Back to the Point

```
struct Point { int x; int y; };

auto p = Point { 7, 0 };

match(p)(
  pattern(ds(0, arg)) = [](int y) {
    printf("Y axis: %d\n", y);
  },
  pattern(ds(arg, 0)) = [](int x) {
    printf("X axis: %d\n", x);
  },
  pattern(ds(arg, arg)) = [](int x, int y) {
    printf("%d, %d\n", x, y);
  }
);

// prints: "X axis: 7"
```





# Re: Evaluating Expressions

```
struct Expr;

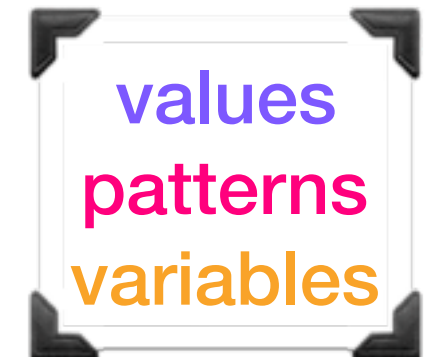
struct Neg { shared_ptr<Expr> expr; };
struct Add { shared_ptr<Expr> lhs, rhs; };
struct Mul { shared_ptr<Expr> lhs, rhs; };

struct Expr : variant<int, Neg, Add, Mul> {
    using variant::variant;
};

namespace std {

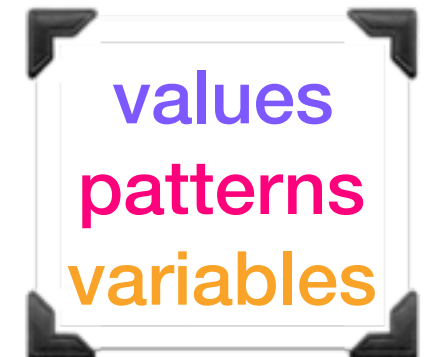
    template <>
    struct variant_size<Expr> // Opt into `VariantLike`
        : integral_constant<size_t, 4> {};

} // namespace std
```

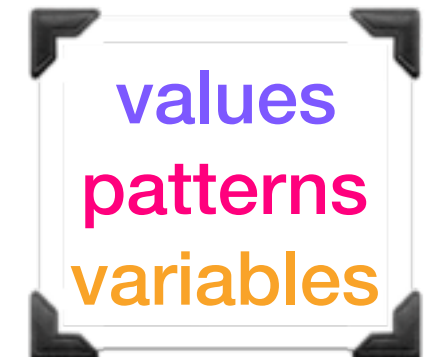


# Re: Evaluating Expressions

```
int eval(const Expr &expr) {  
    return match(expr)(  
        pattern(as<int>(arg)) = [](int value) {  
            return value;  
        },  
        pattern(as<Neg>(ds(arg))) = [](auto &expr) {  
            return -eval(*expr);  
        },  
        pattern(as<Add>(ds(arg, arg))) = [](auto &lhs, auto &rhs) {  
            return eval(*lhs) + eval(*rhs);  
        },  
        pattern(as<Mul>(ds(arg, arg))) = [](auto &lhs, auto &rhs) {  
            return eval(*lhs) * eval(*rhs);  
        }  
    );  
}
```



# Optional Flag



```
optional<string> flag = "-v";
```

```
match(flag)(  
  pattern(some(arg(anyof("-v", "--verbose")))) = [](auto &flag) {  
    // `flag` == "-v" or "--verbose"  
  },  
  pattern(some(arg)) = [](auto &flag) {  
    WHEN(starts_with(flag, "-W")) { // pattern guard!  
      // ...  
    };  
  },  
  pattern(some(_)) = [] { printf("unknown flag!"); }  
  pattern(none) = [] {}  
);
```

# Patterns So Far

| Pattern        | Matches                           | Example                             |
|----------------|-----------------------------------|-------------------------------------|
| Expression     | Any                               | <code>0</code>                      |
| Arg / Wildcard | Any                               | <code>arg / _</code>                |
| Destructure    | Array, Aggregate, TupleLike       | <code>ds(0, _)</code>               |
| As             | Polymorphic, VariantLike, AnyLike | <code>as&lt;Add&gt;(arg)</code>     |
| Optional       | PointerLike                       | <code>some(_), none</code>          |
| Alternation    | One of N Patterns                 | <code>anyof("-f", "--force")</code> |

# Simplifying Expressions

Let's simplify the expression tree we've been evaluating.

Simplification Rules:

- $-(-v) == v$
- $v + 0 == v$
- $v \times 1 == v$
- $v \times 0 == 0$

# Simplifying Expressions

```
struct Expr;

struct Neg { shared_ptr<Expr> expr; };
struct Add { shared_ptr<Expr> lhs, rhs; };
struct Mul { shared_ptr<Expr> lhs, rhs; };

struct Expr : variant<int, Neg, Add, Mul> {
    using variant::variant;
};

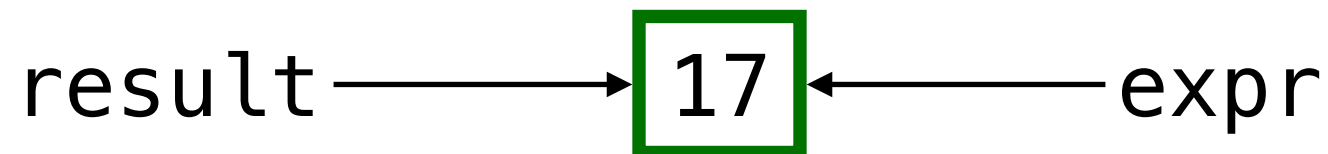
namespace std {

    template <>
    struct variant_size<Expr> // Opt into `VariantLike`
        : integral_constant<size_t, 4> {};

} // namespace std
```

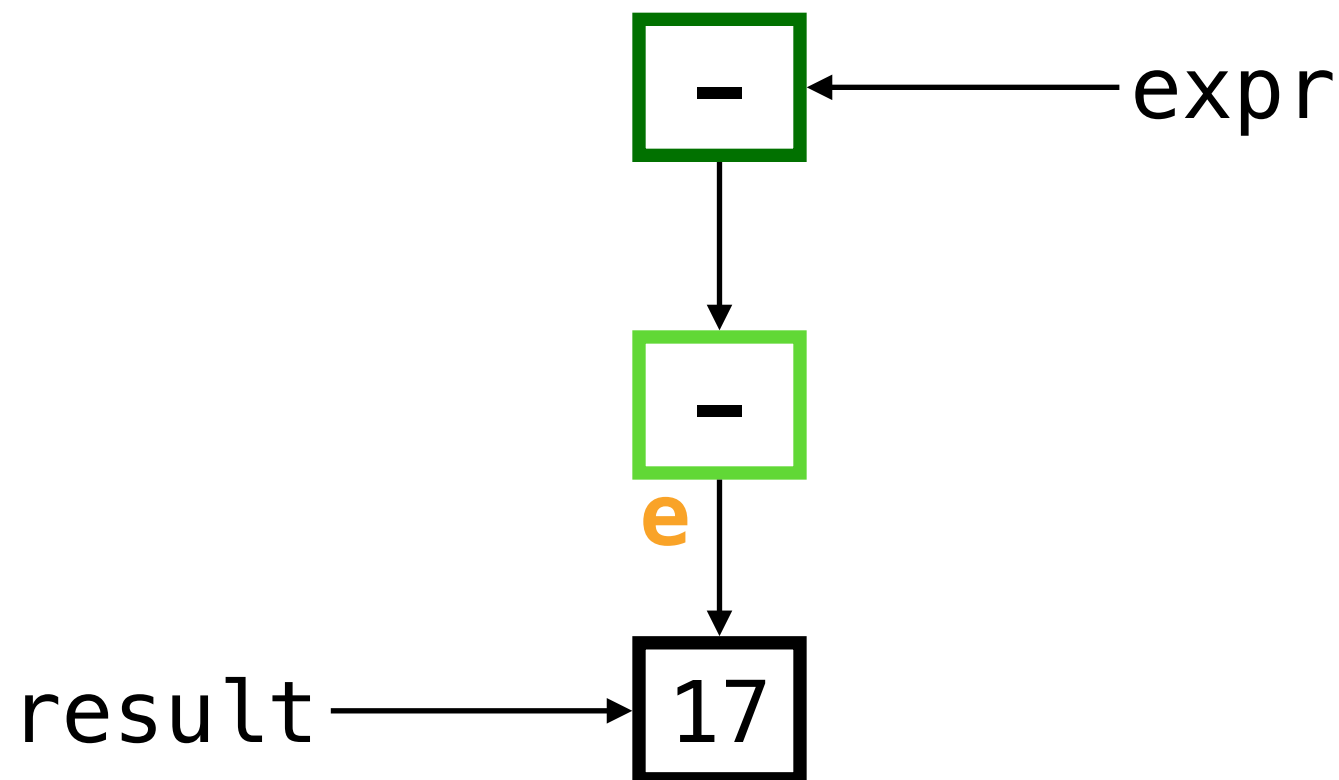
# Simplifying int

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {  
    return match(*expr)(  
        pattern(as<int>(_)) = [&] { return expr; },  
        // ...  
    );  
}
```



# Simplifying $-(-v)$

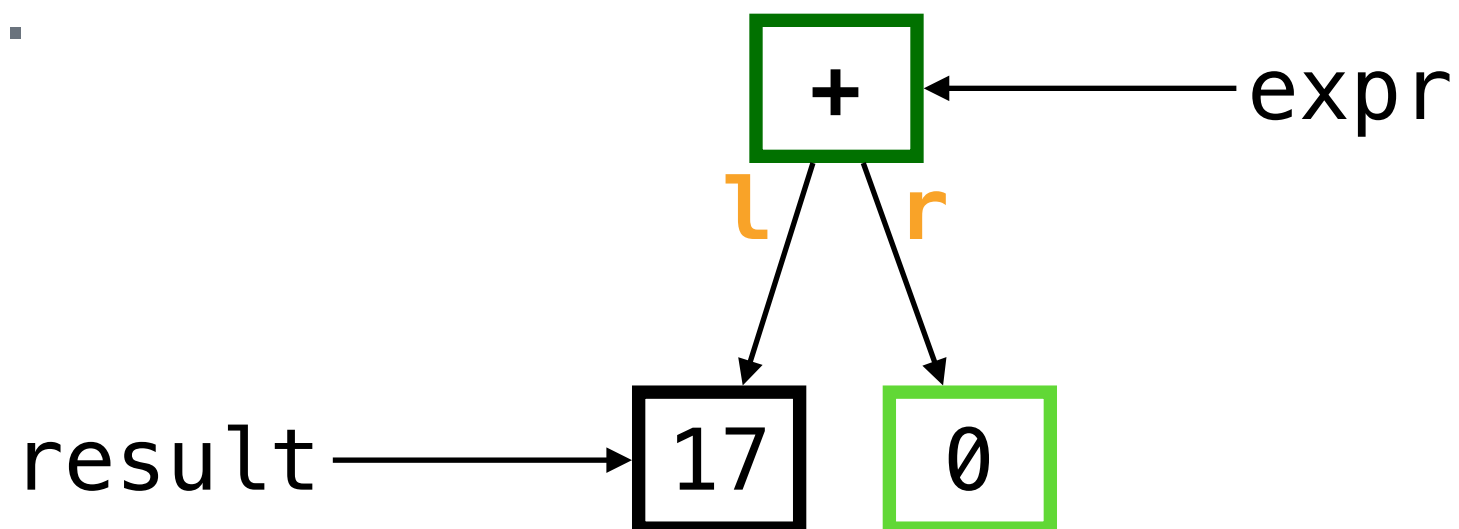
```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {  
    return match(*expr)(  
        // ...  
        pattern(as<Neg>(ds(some(as<Neg>(ds(arg)))))) = [] (auto &e) {  
            return simplify(e);  
        },  
        // ...  
    );  
}
```





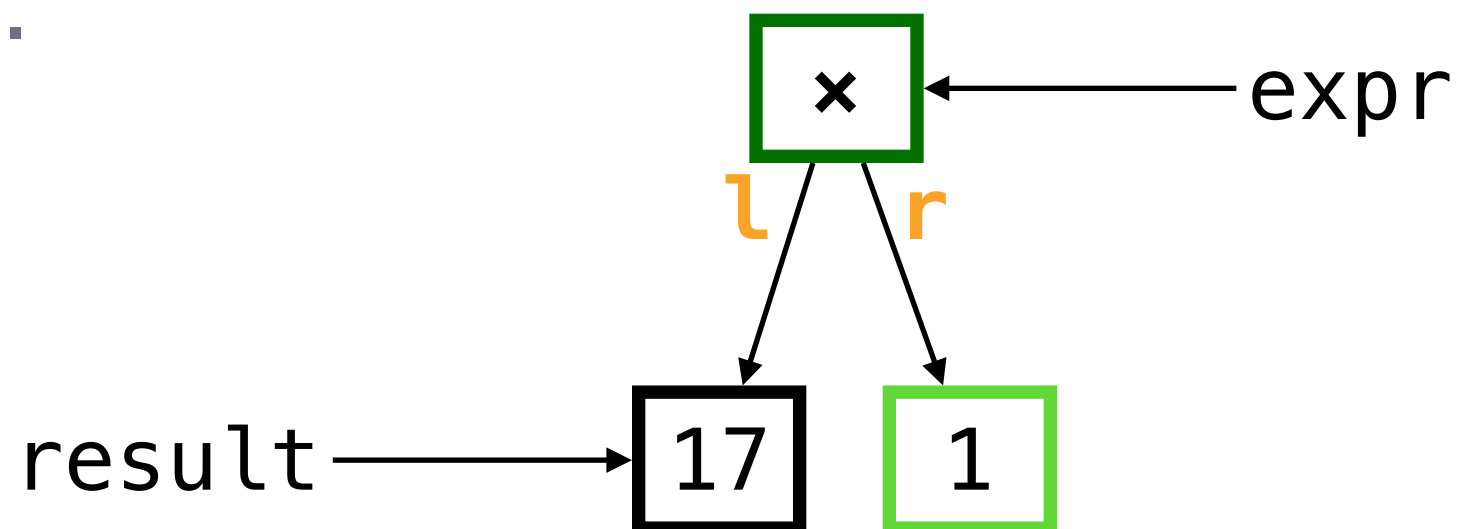
# Simplifying $v + 0$

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {  
    return match(*expr)(  
        // ...  
        pattern(as<Add>(ds(some(as<int>(0)), arg))) = [](auto &r) {  
            return simplify(r);  
        },  
        pattern(as<Add>(ds(arg, some(as<int>(0))))) = [](auto &l) {  
            return simplify(l);  
        },  
        // ...  
    );  
}
```



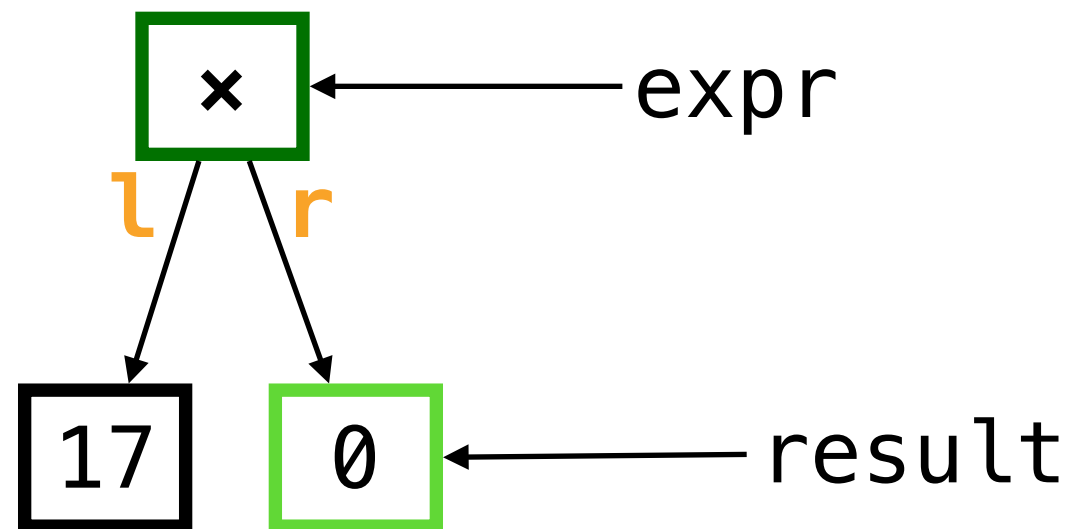
# Simplifying $v \times 1$

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {  
    return match(*expr)(  
        // ...  
        pattern(as<Mul>(ds(some(as<int>(1)), arg))) = [](auto &r) {  
            return simplify(r);  
        },  
        pattern(as<Mul>(ds(arg, some(as<int>(1))))) = [](auto &l) {  
            return simplify(l);  
        },  
        // ...  
    );  
}
```



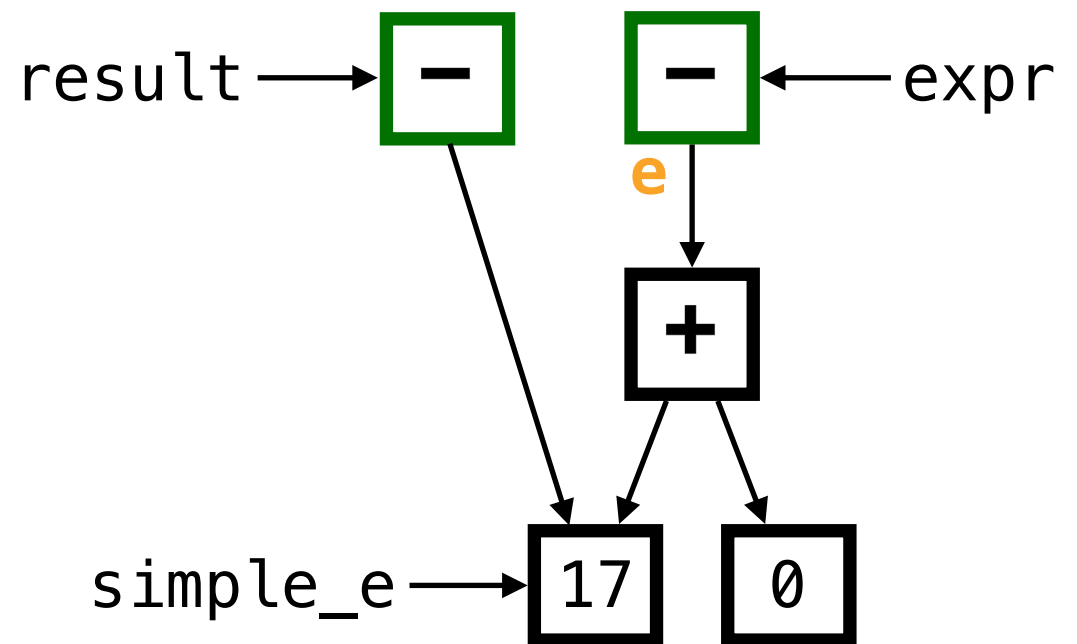
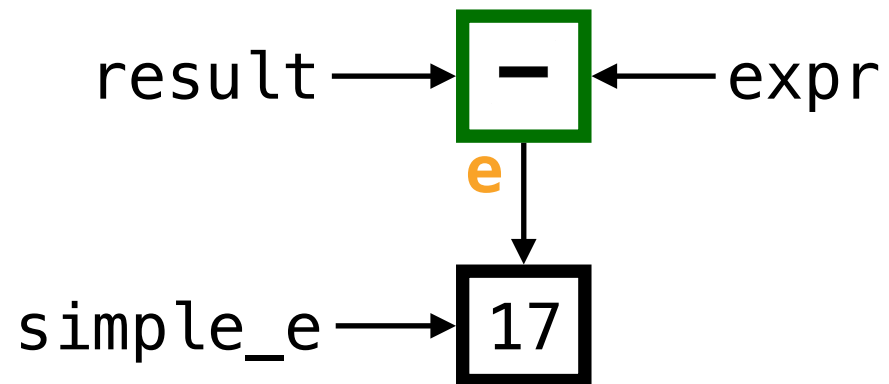
# Simplifying $v \times 0$

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {  
    return match(*expr)({  
        // ...  
        pattern(as<Mul>(ds(arg(some(as<int>(0))), _))) = [](auto &l) {  
            return l;  
        },  
        pattern(as<Mul>(ds(_, arg(some(as<int>(0)))))) = [](auto &r) {  
            return r;  
        },  
        // ...  
    });  
}
```



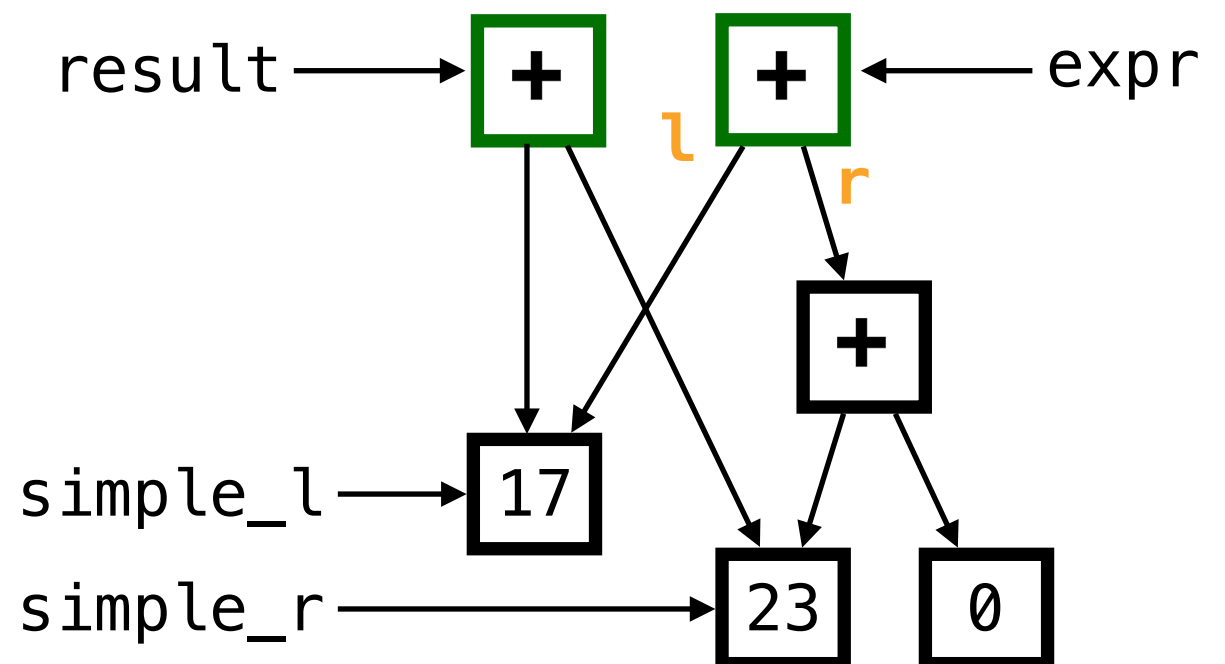
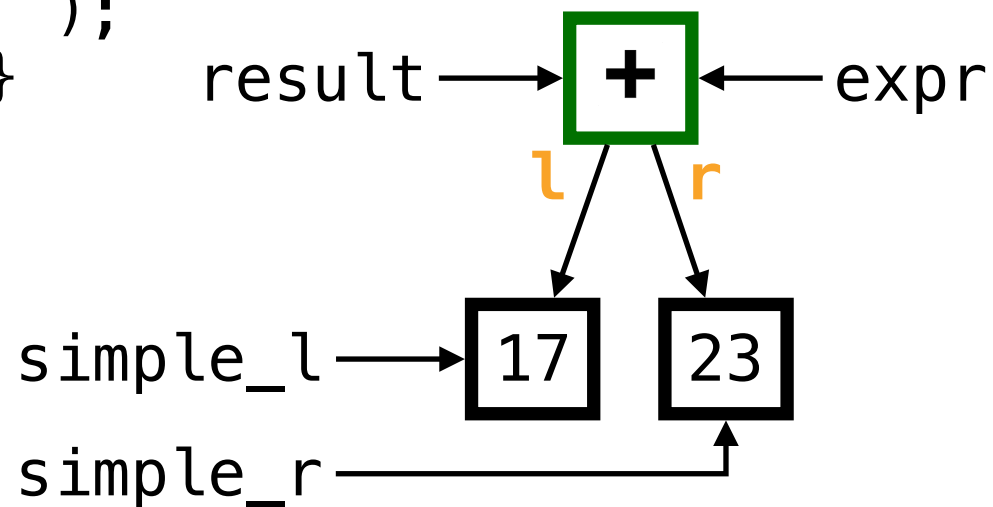
# Simplifying –

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {
    return match(*expr)(
        // ...
        pattern(as<Neg>(ds(arg))) = [&](auto &e) {
            auto simple_e = simplify(e);
            return simple_e == e
                ? expr
                : simplify(make_shared<Expr>(Neg{simple_e}));
        },
        // ...
    );
}
```



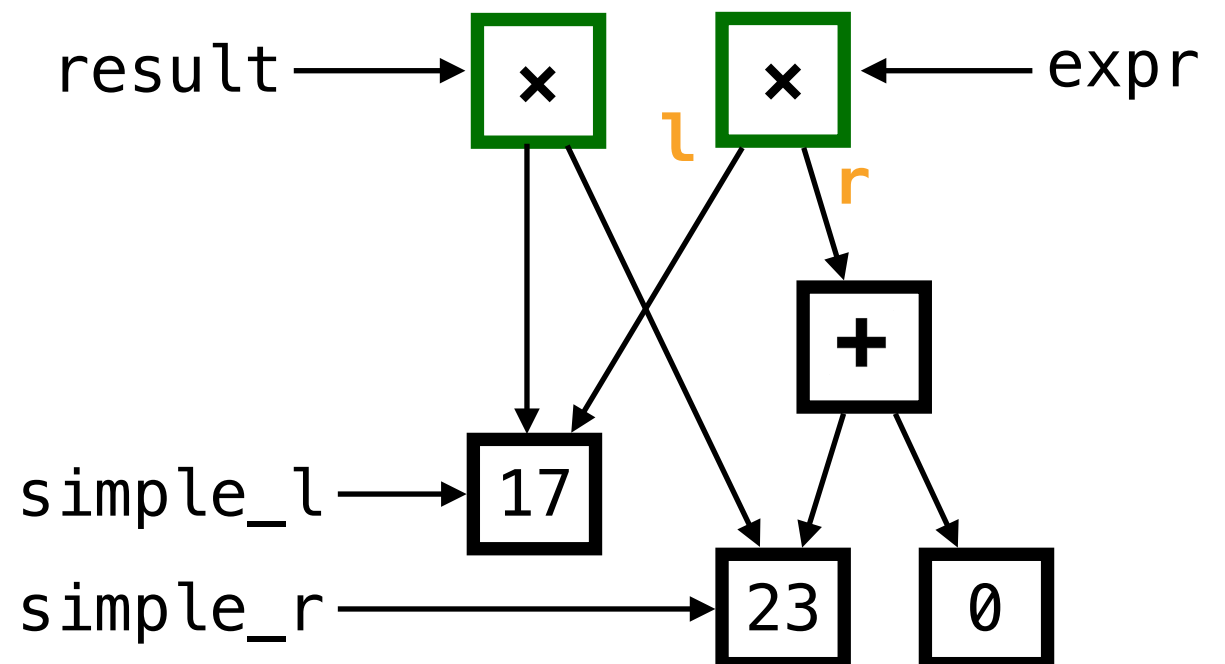
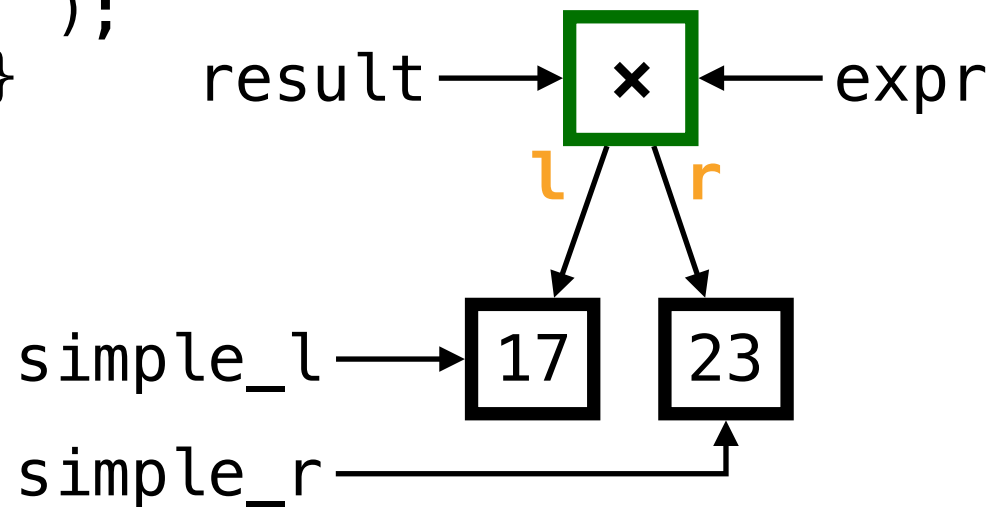
# Simplifying +

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {
    return match(*expr)(
        // ...
        pattern(as<Add>(ds(arg, arg))) = [&](auto &l, auto &r) {
            auto simple_l = simplify(l), simple_r = simplify(r);
            return simple_l == l && simple_r == r
                ? expr
                : simplify(make_shared<Expr>(Add{simple_l,
                                                    simple_r}));
        },
        // ...
    );
}
```



# Simplifying ×

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {
    return match(*expr)(
        // ...
        pattern(as<Mul>(ds(arg, arg))) = [&](auto &l, auto &r) {
            auto simple_l = simplify(l), simple_r = simplify(r);
            return simple_l == l && simple_r == r
                ? expr
                : simplify(make_shared<Expr>(Mul{simple_l,
                                                    simple_r}));
        },
        // ...
    );
}
```



# Putting it all together

```
shared_ptr<Expr> simplify(const shared_ptr<Expr> &expr) {
    return match(*expr)(
        pattern(as<int>(_)) = [&] { return expr; }, // v
        pattern(
            anyof(as<Neg>(ds(some(as<Neg>(ds(arg))))), // -(-v)
                as<Add>(ds(some(as<int>(0)), arg)), // v + 0
                as<Add>(ds(arg, some(as<int>(0)))),
                as<Mul>(ds(some(as<int>(1)), arg)), // v * 1
                as<Mul>(ds(arg, some(as<int>(1)))))) = [&](auto &e) {
                return simplify(e);
            },
            pattern(
                anyof(as<Mul>(ds(arg(some(as<int>(0))), _)), // v * 0
                    as<Mul>(ds(_, arg(some(as<int>(0)))))) = [&](auto &zero) {
                    return zero;
                },
                pattern(as<Neg>(ds(arg))) = [&](auto &e) { /* ... */ },
                pattern(as<Add>(ds(arg, arg))) = [&](auto &l, auto &r) { /* ... */ },
                pattern(as<Mul>(ds(arg, arg))) = [&](auto &l, auto &r) { /* ... */ }
            )
    );
}
```

**The real power of pattern matching is that the patterns are built the same way as the values**



# **A Few More Things...**

# Identifiers

- Typically introduced inside the pattern, but...
- Introduced as parameters of a lambda instead
- Patterns do not have any identifiers

# Pattern Guard

```
int fib(int n) {  
    return match(n)(  
        pattern(arg) = [] (int x) {  
            WHEN(x <= 0) { return 0; };  
        },  
        pattern(1) = [] { return 1; },  
        pattern(arg) = [] (int x) {  
            return fib_v1(x - 1) + fib_v1(x - 2);  
        });  
}
```

- Lives inside the handler... why?
- Reusing the identifier introduced in the lambda

# Identifier Pattern

```
struct Point { int x; int y; };  
auto p = Point { 7, 0 };
```

```
IDENTIFIERS(x, y);
```

```
match(p)(  
    pattern(ds(0 , y )) = [](int y) {  
        printf("Y axis: %d\n", y);  
    },  
    pattern(ds(x , 0 )) = [](int x) {  
        printf("X axis: %d\n", x);  
    },  
    pattern(ds(x , y )) = [](int x, int y) {  
        printf("%d, %d\n", x, y);  
    }  
);
```

```
// prints: "X axis: 7"
```

# Identifier Pattern

Repeated identifiers mean the values have to be equal!

```
tuple<int, int, int> t = { 101, 202, 101 };
```

```
IDENTIFIERS(x, y);  
match(t) {  
    pattern(ds(x, x, x)) = [] { printf("all the same!"); },  
    pattern(ds(x, y, x)) = [] { printf("bordered!"); },  
    pattern(_) = [] { printf("No recognized pattern"); }  
};  
  
// prints: "bordered!"
```

# Back to the Pattern Guard

```
int fib(int n) {  
  IDENTIFIERS(x);  
  return match(n)(  
    pattern(x).when(x <= 0) = [](int) { return 0; },  
    pattern(1) = [] { return 1; },  
    pattern(x) = [](int x) {  
      return fib_v1(x - 1) + fib_v1(x - 2);  
    });  
}
```

# Variadic Pattern

Syntax: `variadic(<pattern>)`

- Exactly once anywhere within a Destructure Pattern
- The inner pattern is repeatedly expanded as necessary

```
tuple<int, int, int> t = { 101, 202, 101 };
```

```
match(t)(  
  pattern(ds(arg, arg, arg)) = [](auto, auto, auto) {});
```

```
match(t)(  
  pattern(ds(variadic(arg))) = [](auto, auto, auto) {});
```

# Inside of a `template`

```
template <typename Tuple>
void print(Tuple &&tuple) {
    match(forward<Tuple>(tuple))(
        pattern(ds(variadic(arg))) = [] (auto &&... xs) {
            int dummy[] = { (cout << xs << ' ', 0)... };
            (void)dummy;
        }
    );
}

print(tuple(101, "hello", 1.1));

// prints: "101 hello 1.1 "
```



# This is C++17 apply!

```
template <typename F, typename Tuple>
decltype(auto) apply(F &&f, Tuple &&t) {
    return match(forward<T>(t))(
        pattern(ds(variadic(arg))) = forward<F>(f));
}
```

# *almost* tuple\_cat

```
tuple<int, int, int> t = { 101, 202, 101 };  
  
match(t)(  
    pattern(ds(variadic(arg))) = [](auto, auto, auto) {});
```

# *almost* tuple\_cat

```
template <typename... Tuples>
auto almost_tuple_cat(Tuples &&... tuples) {
    return match(forward<Tuples>(tuples)...)(
        pattern(variadic(ds(variadic(arg)))) = [](auto &&... xs) {
            return make_tuple(forward<decltype(xs)>(xs)...);
        });
}
```

# Performance

# MPark.Patterns

```
void fizzbuzz() {  
    using namespace mpark::patterns;  
    for (int i = 1; i <= 100; ++i) {  
        match(i % 3, i % 5)(  
            pattern(0, 0) = [] { printf("fizzbuzz\n"); },  
            pattern(0, _) = [] { printf("fizz\n"); },  
            pattern(_, 0) = [] { printf("buzz\n"); },  
            pattern(_, _) = [i] { printf("%d\n", i); });  
    }  
}
```

# switch

```
void fizzbuzz() {  
    for (int i = 1; i <= 100; ++i) {  
        switch (i % 3) {  
            case 0:  
                switch (i % 5) {  
                    case 0: printf("fizzbuzz\n"); break;  
                    default: printf("fizz\n"); break;  
                }  
                break;  
            default:  
                switch (i % 5) {  
                    case 0: printf("buzz\n"); break;  
                    default: printf("%d\n", i); break;  
                }  
                break;  
        }  
    }  
}
```

|                         |                         |                      |                      |
|-------------------------|-------------------------|----------------------|----------------------|
| fizzbuzz(): #           | call printf             | imul rax, rcx,       | jmp .LBB1_8          |
| @fizzbuzz()             | jmp .LBB0_9             | 1431655766           | .LBB1_6: # in Loop:  |
| push rbx                | .LBB0_4: # in Loop:     | mov rdx, rax         | Header=BB1_1 Depth=1 |
| mov ebx, 1              | Header=BB0_1 Depth=1    | shr rdx, 63          | mov edi, .Lstr       |
| .LBB0_1: # =>This Inner | test eax, eax           | shr rax, 32          | jmp .LBB1_8          |
| Loop Header: Depth=1    | je .LBB0_7              | add eax, edx         | .LBB1_7: # in Loop:  |
| movsxd rcx, ebx         | mov edi, .Lstr.4        | lea edx, [rax +      | Header=BB1_1 Depth=1 |
| imul rax, rcx,          | jmp .LBB0_8             | 2*rax]               | mov edi, .Lstr.5     |
| 1431655766              | .LBB0_6: # in Loop:     | imul rax, rcx,       | .LBB1_8: # in Loop:  |
| mov rdx, rax            | Header=BB0_1 Depth=1    | 1717986919           | Header=BB1_1 Depth=1 |
| shr rdx, 63             | mov edi, .Lstr          | mov rsi, rax         | call puts            |
| shr rax, 32             | jmp .LBB0_8             | shr rsi, 63          | .LBB1_9: # in Loop:  |
| add eax, edx            | .LBB0_7: # in Loop:     | sar rax, 33          | Header=BB1_1 Depth=1 |
| lea edx, [rax +         | Header=BB0_1 Depth=1    | add eax, esi         | inc ebx              |
| 2*rax]                  | mov edi, .Lstr.5        | lea esi, [rax +      | cmp ebx, 101         |
| imul rax, rcx,          | .LBB0_8: # in Loop:     | 4*rax]               | jne .LBB1_1          |
| 1717986919              | Header=BB0_1 Depth=1    | mov eax, ecx         | xor eax, eax         |
| mov rsi, rax            | call puts               | sub eax, esi         | pop rbx              |
| shr rsi, 63             | .LBB0_9: # in Loop:     | cmp ecx, edx         | ret                  |
| sar rax, 33             | Header=BB0_1 Depth=1    | je .LBB1_4           | .L.str.3:            |
| add eax, esi            | inc ebx                 | test eax, eax        | .asciz "%d\n"        |
| lea esi, [rax +         | cmp ebx, 101            | je .LBB1_6           | .Lstr:               |
| 4*rax]                  | jne .LBB0_1             | mov edi, .L.str.3    | .asciz "buzz"        |
| mov eax, ecx            | pop rbx                 | xor eax, eax         |                      |
| sub eax, esi            | ret                     | mov esi, ebx         |                      |
| cmp ecx, edx            | main: # @main           | call printf          | .Lstr.4:             |
| je .LBB0_4              | push rbx                | jmp .LBB1_9          | .asciz "fizz"        |
| test eax, eax           | mov ebx, 1              | .LBB1_4: # in Loop:  |                      |
| je .LBB0_6              | .LBB1_1: # =>This Inner | Header=BB1_1 Depth=1 | .Lstr.5:             |
| mov edi, .L.str.3       | Loop Header: Depth=1    | test eax, eax        | .asciz "fizzbuzz"    |
| xor eax, eax            | movsxd rcx, ebx         | je .LBB1_7           |                      |
| mov esi, ebx            |                         | mov edi, .Lstr.4     |                      |

Exactly the same  
generated code!

# Future Work



# Future Work

- Determine an API for Ranges
- Experiment further with identifiers
- Exhaustiveness checking

# MPark.Patterns

Pattern Matching in C++

<https://github.com/mpark/patterns>

**Michael Park**



@mpark



@mcypark



MESOSPHERE

# Implementation Peek

# Structure

```
using namespace mpark::patterns;
IDENTIFIERS(<identifier>...); // optional
match(<expr>...)(
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    // ...
);
```

# Structure

```
using namespace mpark::patterns;
IDENTIFIERS(<identifier>...); // optional
match(<expr>...)(
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    // ...
);

template <typename... Patterns>
struct Pattern {
    template <typename F>
    Case<Pattern, F> operator=(F &&f) && noexcept;

    Ds<Patterns &&...> patterns;
};
```

# Structure

```
using namespace mpark::patterns;
IDENTIFIERS(<identifier>...); // optional
match(<expr>...)(
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    // ...
);

template <typename Pattern, typename F>
struct Case {
    Pattern pattern;
    F f;
};
```

# Structure

```
using namespace mpark::patterns;
IDENTIFIERS(<identifier>...); // optional
match(<expr>...)(
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    pattern(<pattern>...) = [](<binding>...) { /* ... */ },
    // ...
);

template <typename... Values>
struct Match {
    template <typename Pattern, typename F, typename... Cases>
    decltype(auto) operator()(Case<Pattern, F> &&case_,
                             Cases&&... cases) && {
        // ...
    }

    tuple<Values &&...> values;
}
```

# Match::operator()

```
template <typename Pattern, typename F, typename... Cases>
decltype(auto) operator()(Case<Pattern, F> &&case_,
                          Cases&&... cases) && {
    auto result = try_match(move(case_).pattern.patterns,
                            move(values),
                            move(case_).f);

    if (result) {
        return move(result).get();
    }

    if constexpr (sizeof...(Cases) == 0) {
        throw match_error{};
    } else {
        return move(*this)(forward<Cases>(cases)...);
    }
}
```



# Match::operator()

```
template <typename Pattern, typename F, typename... Cases>
decltype(auto) operator()(Case<Pattern, F> &&case_,
                          Cases&&... cases) && {
    auto result = try_match(move(case_).pattern.patterns,
                           move(values),
                           move(case_).f);

    if (result) {
        return move(result).get();
    }

    if constexpr (sizeof...(Cases) == 0) {
        throw match_error{};
    } else {
        return move(*this)(forward<Cases>(cases)...);
    }
}
```

# match\_result<T>

```
template <typename T>
struct match_result : optional<forwarder<T>> {
    using type = T;

    using super = optional<forwarder<T>>;
    using super::super;

    match_result(no_match_t) noexcept {}
    match_result(nullopt_t) = delete;

    decltype(auto) get() && {
        return (*static_cast<super &&>(*this)).forward();
    }
};
```

# match\_result-aware invoke

```
// `invoke`-like utility for `try_match` functions.
template <typename F, typename... Args>
auto match_invoke(F &&f, Args &&... args) {
    using R = invoke_result_t<F, Args...>;
    if constexpr (is_void_v<R>) {
        invoke(forward<F>(f), forward<Args>(args)...);
        return match_result<void>(void_{});
    } else if constexpr (is_match_result_v<R>) {
        return invoke(forward<F>(f), forward<Args>(args)...);
    } else {
        return match_result<R>(
            invoke(forward<F>(f), forward<Args>(args)...));
    }
}
```

# Expression Pattern

```
template <typename ExprPattern, typename Value, typename F>
auto try_match(const ExprPattern &expr_pattern,
               Value &&value,
               F &&f) {
    return expr_pattern == forward<Value>(value)
        ? match_invoke(forward<F>(f))
        : no_match;
}
```

# Arg Pattern

```
template <typename Pattern, typename Value, typename F>
auto try_match(const Arg<Pattern> &arg,
               Value &&value,
               F &&f) {
    if constexpr (is_void_v<Pattern>) {
        return match_invoke(forward<F>(f),
                             forward<Value>(value));
    } else {
        return try_match(arg.pattern,
                         forward<Value>(value),
                         forward<F>(f));
    }
}
```