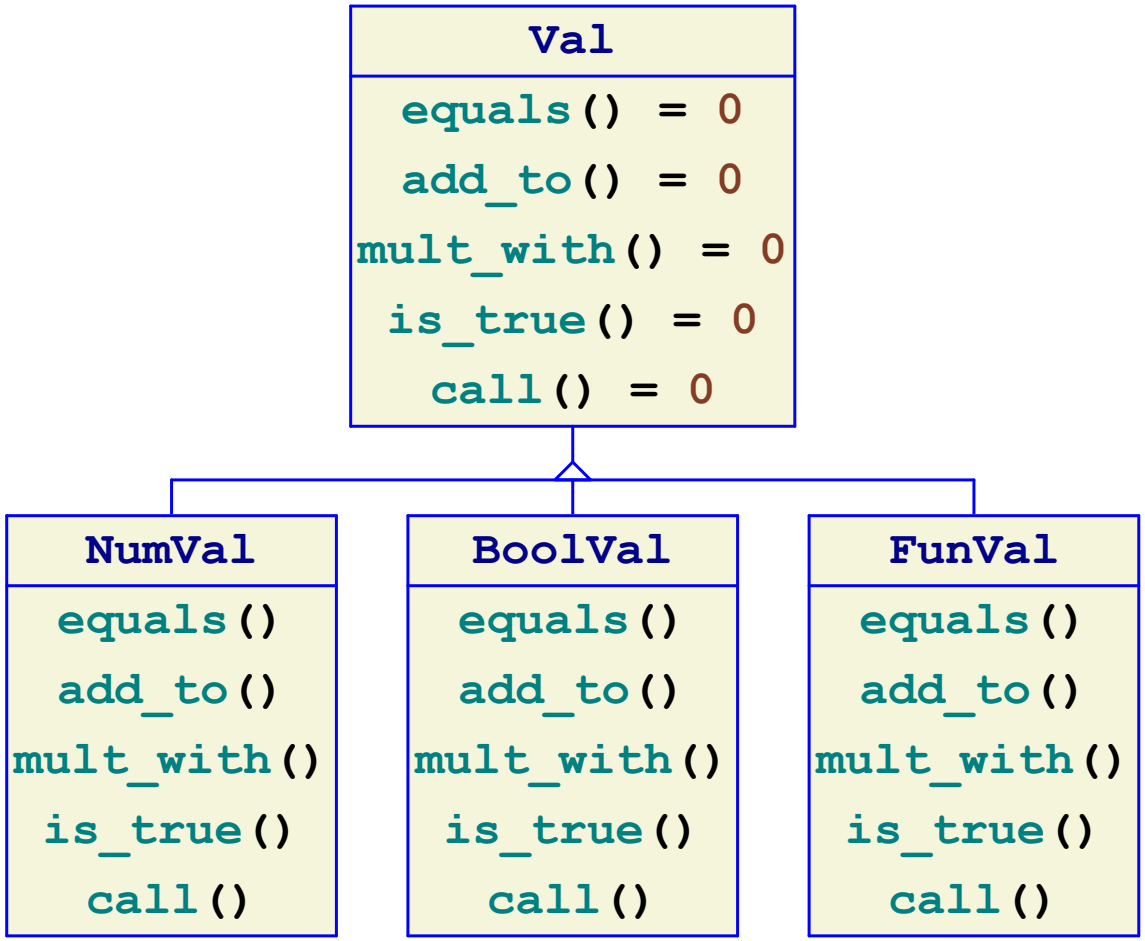
Adding Operations to MSDscript

```
x + y
x - y
x * y
x / y
x % y
x == y
x != y
```

Plan:

- First, reduce repetitive code in current implementation
- Then, add new code for more operators

Value Variants



Value Variants

```
class Val {
  virtual bool equals(PTR(Val) other_val) = 0;
  virtual PTR(Expr) to_expr() = 0;
  virtual std::string to_string() = 0;

  virtual PTR(Val) add_to(PTR(Val) other_val) = 0;
  virtual PTR(Val) mult_with(PTR(Val) other_val) = 0;

  virtual bool is_true() = 0;

  virtual PTR(Val) call(PTR(Val) actual_arg) = 0;
  virtual void call_step(PTR(Val) actual_arg, PTR(Cont) cont) = 0;
};
```

Value Variants

```
class Val {
  virtual bool equals(PTR(Val) other_val) = 0;
  virtual PTR(Expr) to_expr() = 0;
  virtual std::string to_string() = 0;

  virtual PTR(Val) add_to(PTR(Val) other_val) = 0;
  virtual PTR(Val) mult_with(PTR(Val) other_val) = 0;

  virtual bool is_true() = 0;

  virtual PTR(Val) call(PTR(Val) actual_arg) = 0;
  virtual void call_step(PTR(Val) actual_arg, PTR(Cont) cont) = 0;
};
```

Value Variants

```
class Val {
  virtual bool equals(PTR(Val) other_val) = 0;
  virtual PTR(Expr) to_expr() = 0;
  virtual std::string to_s
  virtual PTR(Val) add_to(PTR(Val) other_val) = 0;
  virtual PTR(Val) mult_with(PTR(Val) other_val) = 0;

  virtual bool is_true() = 0;

  virtual PTR(Val) call(PTR(Val) actual_arg) = 0;
  virtual void call_step(PTR(Val) actual_arg, PTR(Cont) cont) = 0;
};
```

BoolVal and **FunVal** have the same implementation

Value Variants

```
class Val {
  virtual bool equals(PTR(Val) other_val) = 0;
  virtual PTR(Expr) to_expr() = 0;
  virtual std::string to_string() = 0;

  virtual PTR(Val) add_to(PTR(Val) other_val) = 0;
  virtual PTR(Val) mult

  virtual bool is_true() = 0;

  virtual PTR(Val) call(PTR(Val) actual_arg) = 0;
  virtual void call_step(PTR(Val) actual_arg, PTR(Cont) cont) = 0;
};
```

NumVal and **FunVal** have the same implementation

Value Variants

```
class Val {
  virtual bool equals(PTR(Val) other_val) = 0;
  virtual PTR(Expr) to_expr() = 0;
  virtual std::string to_string() = 0;

  virtual PTR(Val) add_to(PTR(Val) other_val) = 0;
  virtual PTR(Val) mult_with(PTR(Val) other_val) = 0;

  virtual bool is_true()

  virtual PTR(Val) call(PTR(Val) actual_arg) = 0;
  virtual void call_step(PTR(Val) actual_arg, PTR(Cont) cont) = 0;
};
```

NumVal and **BoolVal** have the same implementation

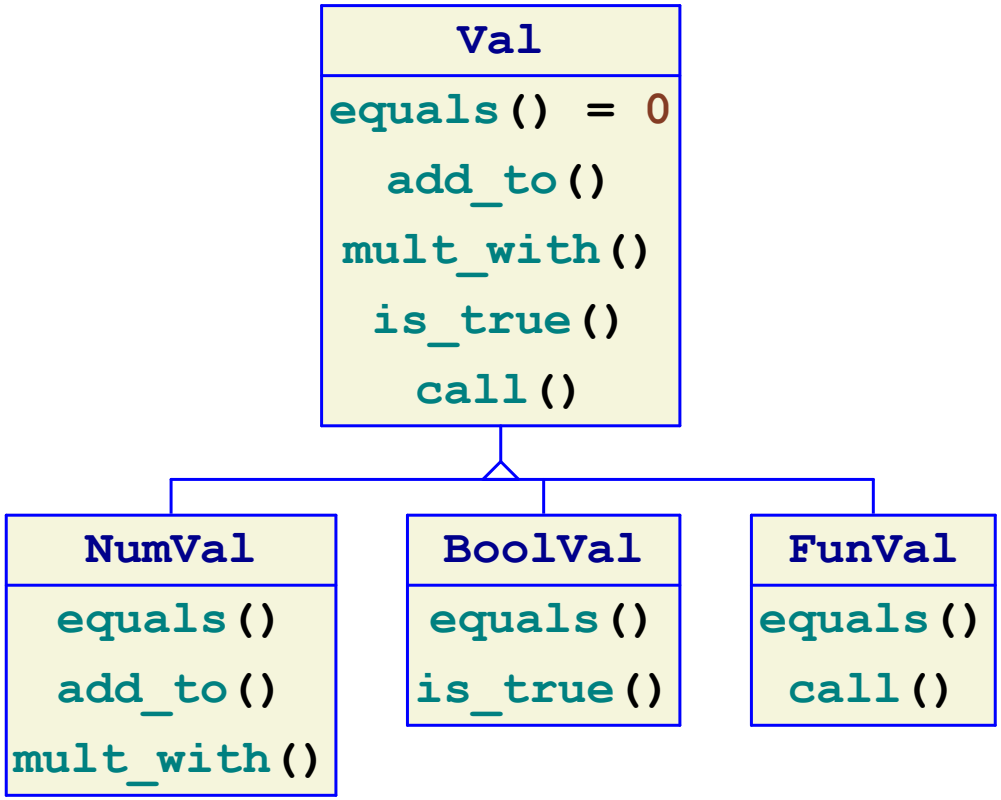
Value Variants

```
class Val {
  virtual bool equals(PTR(Val) other_val) = 0;
  virtual PTR(Expr) to_expr() = 0;
  virtual std::string to_string() = 0;

  virtual PTR(Val) add_to(PTR(Val) other_val);
  virtual PTR(Val) mult_with(PTR(Val) other_val);

  virtual bool is_true();

  virtual PTR(Val) call(PTR(Val) actual_arg);
  virtual void call_step(PTR(Val) actual_arg, PTR(Cont) cont);
};
```

Change `Val` to have a default implementation

⇒ use *inheritance*

Value Variants



Value Default Implementations

```
PTR(Val) Val::add_to(PTR(Val) other_val) {
  throw std::runtime_error("not a number");
}

PTR(Val) Val::mult_with(PTR(Val) other_val) {
  throw std::runtime_error("not a number");
}

bool Val::is_true() {
  throw std::runtime_error("not a boolean");
}

PTR(Val) Val::call(PTR(Val) actual_arg) {
  throw std::runtime_error("not a function");
}

void Val::call_step(PTR(Val) actual_arg, PTR(Cont) cont) {
  throw std::runtime_error("not a function");
}
```

Value Default Implementations

```
PTR(Val) Val::not_a(std::string what) {
  throw std::runtime_error("not a" + what + ": " + this->to_string());
}

PTR(Val) Val::add_to(PTR(Val) other_val) {
  not_a("number");
}

PTR(Val) Val::mult_with(PTR(Val) other_val) {
  not_a("number");
}

bool Val::is_true() {
  not_a("boolean");
}

PTR(Val) Val::call(PTR(Val) actual_arg) {
  not_a("function");
}

void Val::call_step(PTR(Val) actual_arg, PTR(Cont) cont) {
  not_a("function");
}
```

Use a *helper method*

`private` and `[[noreturn]]` implementations

```
PTR(Val) Val::not_a(std::string what) {
  throw std::runtime_error("not a" + what + ": " + this->to_string());
}

PTR(Val) Val::add_to(PTR(Val) other_val) {
  not_a("number");
}

PTR(Val) Val::mult_with(PTR(Val) other_val) {
  not_a("number");
}

bool Val::is_true() {
  not_a("boolean");
}

PTR(Val) Val::call(PTR(Val) actual_arg) {
  not_a("function");
}

void Val::call_step(PTR(Val) actual_arg, PTR(Cont) cont) {
  not_a("function");
}
```

Value Default Implementations

```
PTR(Val) Val::not_a(std::string what) {
  throw std::runtime_error("not a" + what + ": " + this->to_string());
}

PTR(Val) Val::add_to(PTR(Val) other_val) {
  not_a("number");
}

PTR(Val) Val::mult_with(PTR(Val) other_val) {
  not_a("number");
}

bool Val::is_true() {
  not_a("boolean");
}

PTR(Val) Val::call(PTR(Val) actual_arg) {
  not_a("function");
}

void Val::call_step(PTR(Val) actual_arg, PTR(Cont) cont) {
  not_a("function");
}
```

How to handle errors is in one place

Value Default Implementations

```
PTR(Val) Val::not_a(std::string what) {
  throw std::runtime_error("not a" + what + ": " + this->to_string());
}

PTR(Val) Val::add_to(PTR(Val) other_val) {
  not_a("number");
}

PTR(Val) Val::mult_with(PTR(Val) other_val) {
  not_a("number");
}

bool Val::is_true() {
  not_a("boolean");
}

PTR(Val) Val::call(PTR(Val) actual_arg) {
  not_a("function");
}

void Val::call_step(PTR(Val) actual_arg, PTR(Cont) cont) {
  not_a("function");
}
```

This code is longer, but better

Value Specialized Implementations

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep + (unsigned)other_num_val->rep);
}

PTR(Val) NumVal::mult_with(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep * (unsigned)other_num_val->rep);
}
```

Value Specialized Implementations

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep + (unsigned)other_num_val->rep);
}

PTR(Val) NumVal::mult_with(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep * (unsigned)other_num_val->rep);
}
```

But no `is_true`, `call`, or `call_step`

Value Specialized Implementations

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep + (unsigned)other_num_val->rep);
}

PTR(Val) NumVal::mult_with(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep * (unsigned)other_num_val->rep);
}
```

Still a lot of repetition in this code

Less Copied Code, Again

```
int NumVal::other_rep(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return other_num_val->rep;
}

PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  return NEW(NumVal)((unsigned)rep + (unsigned)other_rep(other_val));
}

PTR(Val) NumVal::mult_with(PTR(Val) other_val) {
  return NEW(NumVal)((unsigned)rep * (unsigned)other_rep(other_val));
}
```

Use a *helper method*, again

Less Copied Code, Again

private

```
int NumVal::other_rep(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return other_num_val->rep;
}

PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  return NEW(NumVal)((unsigned)rep + (unsigned)other_rep(other_val));
}

PTR(Val) NumVal::mult_with(PTR(Val) other_val) {
  return NEW(NumVal)((unsigned)rep * (unsigned)other_rep(other_val));
}
```

Use a *helper method*, again

Less Copied Code, Again

```
int NumVal::other_rep(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    val_not_a(other_val, "number");
  else
    return other_num_val->rep;
}

PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  return NEW(NumVal)((unsigned)rep + (unsigned)other_rep(other_val));
}

PTR(Val) NumVal::mult_with(PTR(Val) other_val) {
  return NEW(NumVal)((unsigned)rep * (unsigned)other_rep(other_val));
}
```

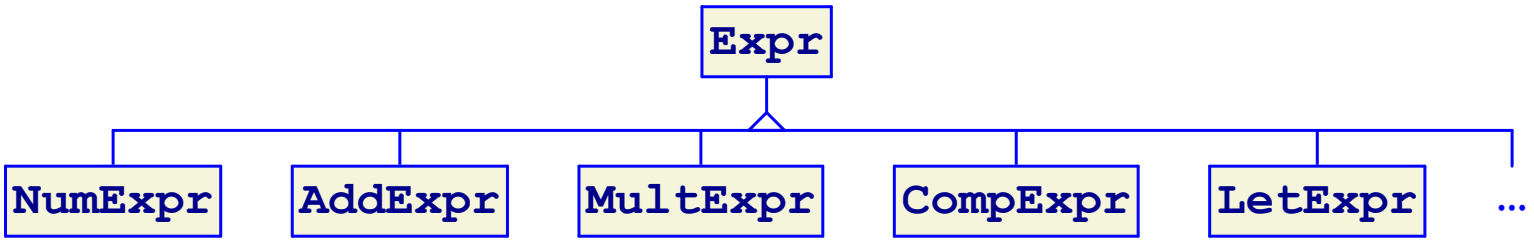
Further consolidate exception throwing

Value Exceptions

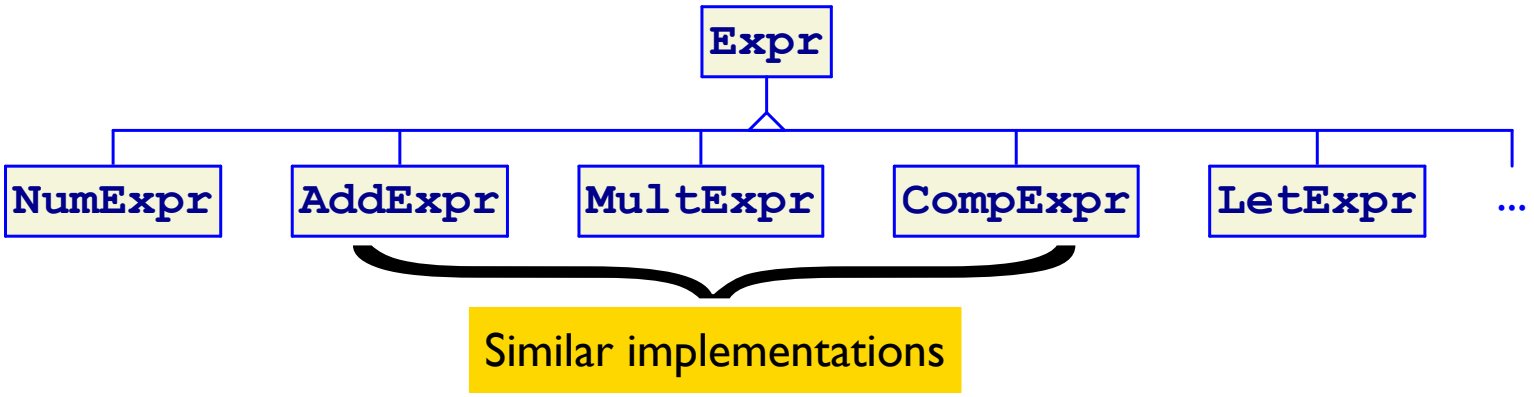
```
class Val {
 ....
private:
  [[noreturn]] PTR(Val) not_a(std::string what);
protected:
  [[noreturn]] PTR(Val) val_not_a(PTR(Val) val, std::string what);
};
```

```
PTR(Val) Val::val_not_a(PTR(Val) val, std::string what) {
  throw std::runtime_error("not a" + what + ": " + val->to_string());
}

PTR(Val) Val::not_a(std::string what) {
  val_not_a(this, what);
}
```

Expression Variants



Expression Variants



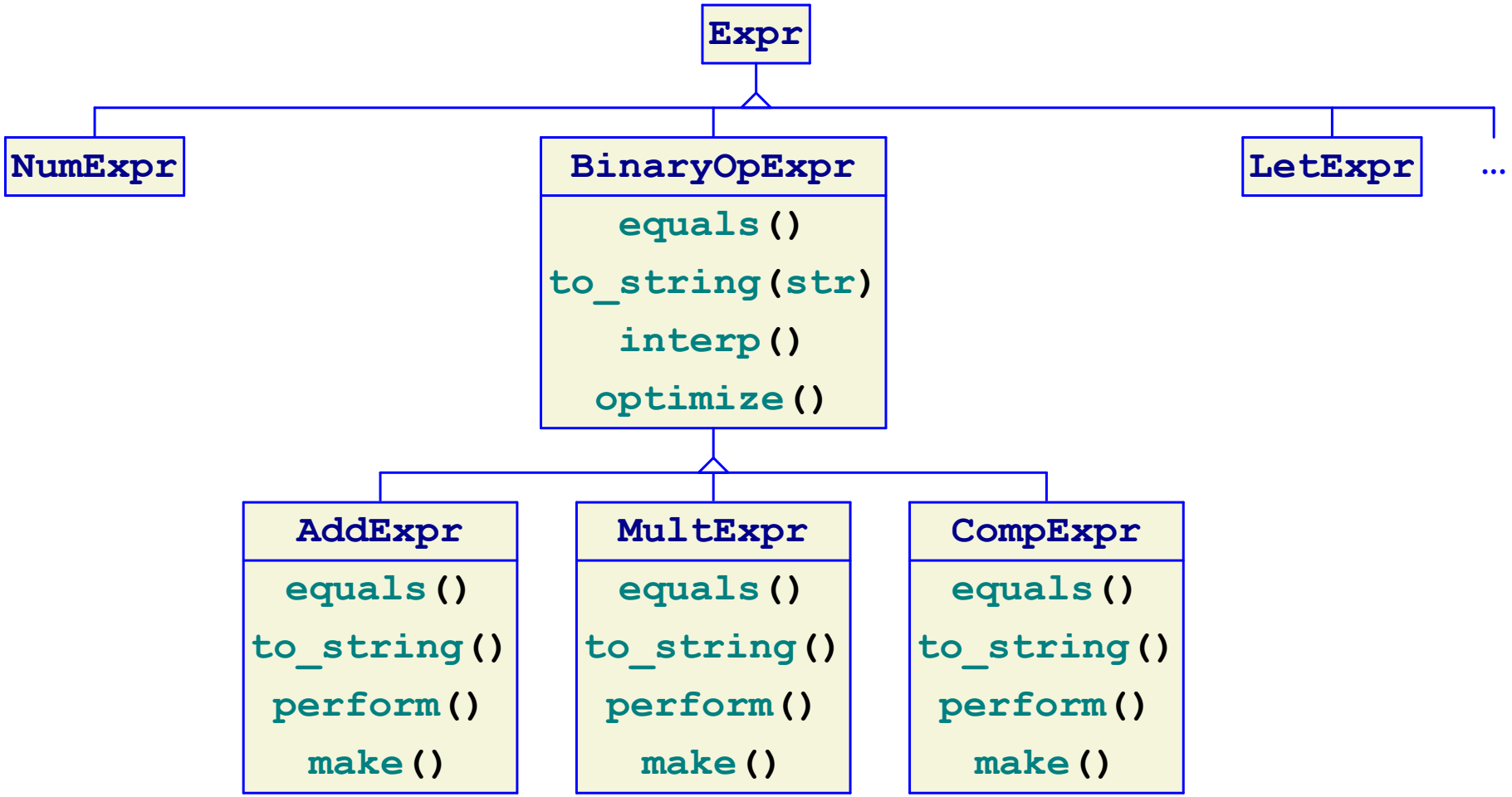
Expression Variants

```
AddExpr::AddExpr(....)
.... lhs->to_string() + "+" ....
.... lhs->interp(env)->add_to    ....
.... NEW(AddExpr) ....
.... NEW(RightThenAddCont) ....
```

```
MultExpr::MultExpr(....)
.... lhs->to_string() + "*" ....
.... lhs->interp(env)->mult_with ....
.... NEW(MultExpr) ....
.... NEW(RightThenMultCont) ....
```

```
CompExpr::CompExpr(....)
.... lhs->to_string() + "==" ....
.... lhs->interp(env)->equals    ....
.... NEW(CompExpr) ....
.... NEW(RightThenCompCont) ....
```

Approach 1: Inheritance plus Overriding



Approach 1: Inheritance plus Overriding

```
class BinaryOpExpr : public Expr {
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  BinaryOpExpr(PTR(Expr) lhs, PTR(Expr) rhs);
  bool equals(PTR(Expr) other_expr);
  .... // all except to_string()

protected:
  std::string to_string(std::string op);
  virtual PTR(Val) perform(PTR(Val) lhs_val, PTR(Val) rhs_val) = 0;
  virtual PTR(BinaryOpExpr) make(PTR(Expr) lhs, PTR(Expr) rhs) = 0;
};
```

Approach 1: Inheritance plus Overriding

New superclass for **`AddExpr`**, etc.

```
class BinaryOpExpr : public Expr {
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  BinaryOpExpr(PTR(Expr) lhs, PTR(Expr) rhs);
  bool equals(PTR(Expr) other_expr);
  .... // all except to_string()

protected:
  std::string to_string(std::string op);
  virtual PTR(Val) perform(PTR(Val) lhs_val, PTR(Val) rhs_val) = 0;
  virtual PTR(BinaryOpExpr) make(PTR(Expr) lhs, PTR(Expr) rhs) = 0;
};
```

Approach 1: Inheritance plus Overriding

```
class BinaryOpExpr : public Expr {
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  BinaryOpExpr(PTR(Expr) lhs, PTR(Expr) rhs);
  bool equals(PTR(Expr) other_expr);
  .... // all except to_string()

protected:
  std::string to_string(std::string op);
  virtual PTR(Val) perform(PTR(Val) lhs_val, PTR(Val) rhs_val) = 0;
  virtual PTR(BinaryOpExpr) make(PTR(Expr) lhs, PTR(Expr) rhs) = 0;
};
```

Used by each `to_string()`

Approach 1: Inheritance plus Overriding

```
class BinaryOpExpr : public Expr {
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  BinaryOpExpr(PTR(Expr) lhs, PTR(Expr) rhs);
  bool equals(PTR(Expr) other_expr);
  .... // all except to_string()

protected:
  std::string to_string(std::string op);
  virtual PTR(Val) perform(PTR(Val) lhs_val, PTR(Val) rhs_val) = 0;
  virtual PTR(BinaryOpExpr) make(PTR(Expr) lhs, PTR(Expr) rhs) = 0;
};
```

Called by `interp`

Approach 1: Inheritance plus Overriding

```
class BinaryOpExpr : public Expr {
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  BinaryOpExpr(PTR(Expr) lhs, PTR(Expr) rhs);
  bool equals(PTR(Expr) other_expr);
  .... // all except to_string()

protected:
  std::string to_string(std::st
  virtual PTR(Val) perform(PTR(
  virtual PTR(BinaryOpExpr) make(PTR(Expr) lhs, PTR(Expr) rhs) = 0;
};
```

Called by `optimize` — a *factory method*

Approach 1: Inheritance plus Overriding

```
class BinaryOpExpr : public Expr {
  PTR(Expr) lhs;
  PTR(Expr) rhs;
  BinaryOpExpr(PTR(Expr) lhs, PTR(Expr) rhs);
  bool equals(PTR(Expr) other_expr);
  .... // all except to_string()

protected:
  std::string to_string(std::string op);
  virtual PTR(Val) perform(PTR(Val) lhs_val, PTR(Val) rhs_val) = 0;
  virtual PTR(BinaryOpExpr) make(PTR(Expr) lhs, PTR(Expr) rhs) = 0;
};
```

Check `lhs` and `rhs`, but also overridden

Refining Equality

```
bool BinaryOpExpr::equals(PTR(Expr) other_expr) {
  PTR(BinaryOpExpr) other_add = CAST(BinaryOpExpr)(other_expr);
  if (other_add == NULL)
    return false;
  else
    return (lhs->equals(other_add->lhs)
            && rhs->equals(other_add->rhs));
}
```

```
bool AddExpr::equals(PTR(Expr) other_expr) {
  return ((CAST(AddExpr)(other_expr) != nullptr)
          && BinaryOpExpr::equals(other_expr));
}
```

Inherit-and-override style

Completing String Conversion

```
std::string BinaryOpExpr::to_string(std::string op_name) {
  return ("(" + lhs->to_string() + op_name
          + rhs->to_string() + ")");
}
```

```
std::string AddExpr::to_string() {
  return BinaryOpExpr::to_string("+");
}
```

Inherit-helper style

Filling in Operations

```
PTR(Val) BinaryOpExpr::interp(PTR(Env) env) {
  return perform(lhs->interp(env), (rhs->interp(env)));
}
```

```
PTR(Val) AddExpr::perform(PTR(Val) lhs_val, PTR(Val) rhs_val) {
  return lhs_val->add_to(rhs_val);
}
```

Virtual-helper style

Filling in Optimization

```
PTR(Expr) BinaryOpExpr::optimize() {
  PTR(Expr) new_lhs = lhs->optimize();
  PTR(Expr) new_rhs = rhs->optimize();

  PTR(Expr) new_expr = make(new_lhs, new_rhs);

  if (new_lhs->has_value() && new_rhs->has_value()) {
    try {
      return new_expr->interp(Env::empty)->to_expr();
    } catch (std::runtime_error exn) {
      // give up on folding
      return new_expr;
    }
  }
}
```

```
PTR(BinaryOpExpr) AddExpr::make(PTR(Expr) lhs, PTR(Expr) rhs) {
  return NEW(AddExpr)(lhs, rhs);
}
```

Filling in Optimization

```
PTR(Expr) BinaryOpExpr::optimize() {
  PTR(Expr) new_lhs = lhs->optimize();
  PTR(Expr) new_rhs = rhs->optimize();

  PTR(Expr) new_expr = make(new_lhs, new_rhs);

  if (new_lhs->has_value() && new_rhs->has_value()) {
    try {
      return new_expr->interp(Env::empty)->to_expr();
    } catch (std::runtime_error exn) {
      // give up on folding
      return new_expr;
    }
  }
}
```

Virtual-helper as factory method style

```
PTR(BinaryOpExpr) AddExpr::make(PTR(Expr) lhs, PTR(Expr) rhs) {
  return NEW(AddExpr)(lhs, rhs);
}
```

Stepping

```
void BinaryOpExpr::step_interp() {
  Step::mode = interp_mode;
  Step::expr = lhs;
  Step::env = Step::env;
  Step::cont = NEW(RightThen....Cont)(rhs, Step::env, Step::cont);
}
```

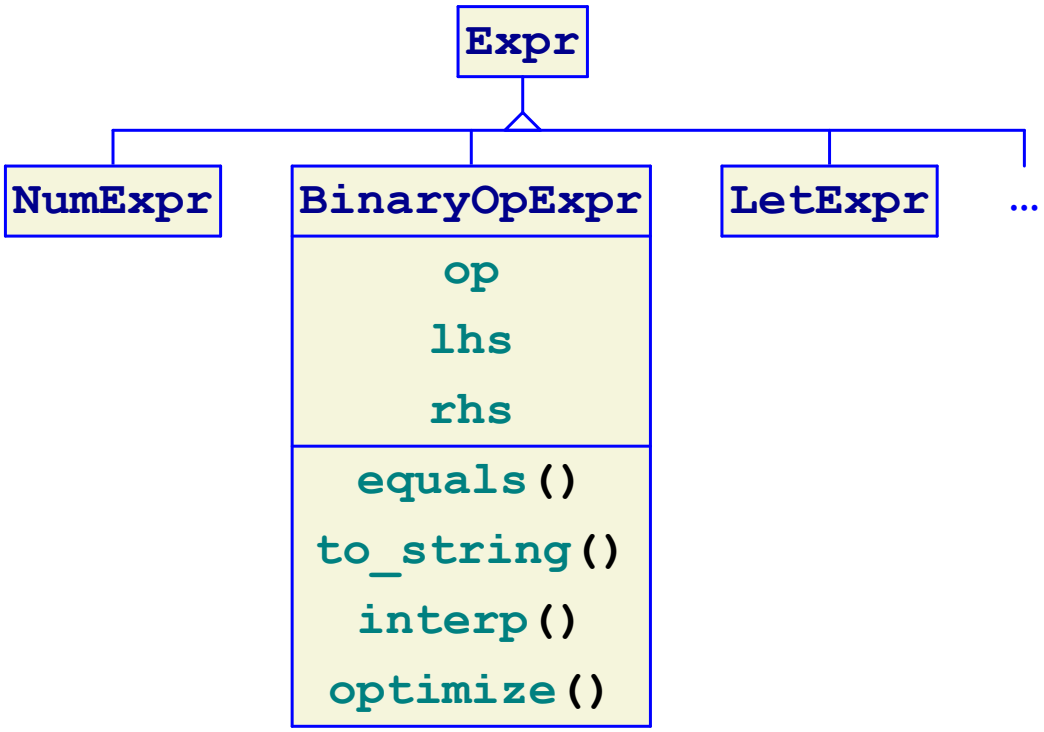
Stepping

```
void BinaryOpExpr::step_interp() {
  Step::mode = interp_mode;
  Step::expr = lhs;
  Step::env = Step::env;
  Step::cont = NEW(RightThen....Cont)(rhs, Step::env, Step::cont);
}
```

Could use another factory method

Maybe we should more directly parameterize over the operation

Approach 2: Delegation



Approach 2: Delegation

```
class BinaryOpExpr : public Expr {
  PTR(BinaryOp) op;
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  .... // all methods
};
```

```
class BinaryOp {
  virtual PTR(Val) perform(PTR(Val), PTR(Val)) = 0;
  virtual std::string to_string() = 0;
  bool equals(PTR(BinaryOp) other_op);
};
```

Approach 2: Delegation

```
class BinaryOpExpr : public Expr {
  PTR(BinaryOp) op;
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  .... // all methods
};
```

Superclass for **AddOp**, **MultOp**, **CompOp**

```
class BinaryOp {
  virtual PTR(Val) perform(PTR(Val), PTR(Val)) = 0;
  virtual std::string to_string() = 0;
  bool equals(PTR(BinaryOp) other_op);
};
```

Approach 2: Delegation

```
class BinaryOpExpr : public Expr {
  PTR(BinaryOp) op;
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  .... // all methods
};
```

```
class BinaryOp {
  virtual PTR(Val) perform(PTR(Val), PTR(Val)) = 0;
  virtual std::string to_string() = 0;
  bool equals(PTR(BinaryOp) other_op);
};
```

```
NEW(BinaryOpExpr)(AddOp::get_instance(), lhs, rhs);
```

Approach 2: Delegation

```
class BinaryOpExpr : public Expr {
  PTR(BinaryOp) op;
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  .... // all methods
};
```

```
class BinaryOp {
  virtual PTR(Val) perform(PTR(Val), PTR(Val)) = 0;
  virtual std::string to_string() = 0;
  bool equals(PTR(BinaryOp) other_op);
};
```

Instead of `NEW(AddExpr)(lhs, rhs`

```
NEW(BinaryOpExpr)(AddOp::get_instance(), lhs, rhs);
```

Approach 2: Delegation

```
class BinaryOpExpr : public Expr {
  PTR(BinaryOp) op;
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  .... // all methods
};
```

```
class BinaryOp {
  virtual PTR(Val) perform(PTR(Val), PTR(Val)) = 0;
  virtual std::string to_string() = 0;
  bool equals(PTR(BinaryOp) other_op);
};
```

Use the *singleton pattern*

```
NEW(BinaryOpExpr)(AddOp::get_instance(), lhs, rhs);
```

Approach 2: Delegation

```
class BinaryOpExpr : public Expr {
  PTR(BinaryOp) op;
  PTR(Expr) lhs;
  PTR(Expr) rhs;

  .... // all methods
};
```

```
class BinaryOp {
  virtual PTR(...al), PTR(Val)) = 0;
  virtual std:...() = 0;
  bool equals(PTR(BinaryOp) other_op);
};
```

Simply pointer equality

Use the *singleton pattern*

```
NEW(BinaryOpExpr)(AddOp::get_instance(), lhs, rhs);
```

Operator Implementation

```
AddOp::AddOp() { } // private

PTR(BinaryOp) AddOp::get_instance() {
  static PTR(AddOp) the_op;
  if (the_op == nullptr)
    the_op = NEW(AddOp)();
  return the_op;
}

PTR(Val) AddOp::perform(PTR(Val) lhs_val, PTR(Val) rhs_val) {
  return lhs_val->add_to(rhs_val);
}

std::string AddOp::to_string() {
  return "+";
}
```

Equality with Delegation

```
bool BinaryOpExpr::equals(PTR(Expr) other_expr) {
  PTR(BinaryOpExpr) other_op = CAST(BinaryOpExpr)(other_expr);
  if (other_op == NULL)
    return false;
  else
    return (op->equals(other_op->op)
            && lhs->equals(other_op->lhs)
            && rhs->equals(other_op->rhs));
}
```

Substitution and Optimization with Delegation

```
PTR(Expr) BinaryOpExpr::subst(NAME_T var, PTR(Val) new_val) {
  return NEW(BinaryOpExpr)(op,
                           lhs->subst(var, new_val),
                           rhs->subst(var, new_val));
}

PTR(Expr) BinaryOpExpr::optimize() {
  PTR(Expr) new_lhs = lhs->optimize();
  PTR(Expr) new_rhs = rhs->optimize();

  PTR(Expr) new_expr = NEW(BinaryOpExpr)(op, new_lhs, new_rhs);
  ....
}
```

Interpreting and String Conversion with Delegation

```
PTR(Val) BinaryOpExpr::interp(PTR(Env) env) {
  return op->perform(lhs->interp(env), (rhs->interp(env)));
}

std::string BinaryOpExpr::to_string() {
  return ("(" + lhs->to_string() + op->to_string()
          + rhs->to_string() + ")");
}
```

Stepping with Delegation

```
void BinaryOpExpr::step_interp() {
  Step::mode = interp_mode;
  Step::expr = lhs;
  Step::env = Step::env;
  Step::cont = NEW(RightThenOpCont)(op, rhs, Step::env, Step::cont);
}
```

Stepping with Delegation

```
void BinaryOpExpr::step_interp() {
  Step::mode = interp_mode;
  Step::expr = lhs;
  Step::env = Step::env;
  Step::cont = NEW(RightThenOpCont)(op, rhs, Step::env, Step::cont);
}
```

Instead of **RightThenAddCont**, **RightThenMultCont**, and **RightThenCompCont**

Continuations with Delegation

```
class RightThenOpCont : public Cont {
  PTR(BinaryOp) op;
  PTR(Expr) rhs;
  PTR(Env) env;
  PTR(Cont) rest;
  .... // all methods
};
```

```
class OpCont : public Cont {
  PTR(BinaryOp) op;
  PTR(Val) lhs_val;
  PTR(Cont) rest;
  .... // all methods
};
```

Continuations with Delegation

```
class RightThenOpCont : public Cont {
  PTR(BinaryOp) op;
  PTR(Expr) rhs;
  PTR(Env) env;
  PTR(Cont) rest;
  .... // all methods
```

Instead of **AddCont**, **MultCont**, and **CompCont**

```
class OpCont : public Cont {
  PTR(BinaryOp) op;
  PTR(Val) lhs_val;
  PTR(Cont) rest;
  .... // all methods
};
```

Continuations with Delegation

```
void RightThenOpCont::step_continue() {
  Step::mode = interp_mode;
  Step::expr = rhs;
  Step::env = env;
  Step::cont = NEW(OpCont)(op, Step::val, rest);
}
```

```
void OpCont::step_continue() {
  Step::mode = continue_mode;
  Step::val = op->perform(lhs_val, Step::val);
  Step::cont = rest;
}
```

A New Operator

```
SubOp::SubOp() { }

PTR(BinaryOp) SubOp::get_instance() {
  static PTR(SubOp) the_op;
  if (the_op == nullptr)
    the_op = NEW(SubOp)();
  return the_op;
}

PTR(Val) SubOp::perform(PTR(Val) lhs_val, PTR(Val) rhs_val) {
  return lhs_val->sub_with(rhs_val);
}

std::string SubOp::to_string() {
  return "-";
}
```

A New Operator

```
SubOp::SubOp() { }

PTR(BinaryOp) SubOp::get_instance() {
  static PTR(SubOp) the_op;
  if (the_op == nullptr)
    the_op = NEW(SubOp)();
  return the_op;
}

PTR(Val) SubOp::perform(PTR(Val) lhs_val, PTR(Val) rhs_val) {
  return lhs_val->sub_with(rhs_val);
}

std::string SubOp::to_string() {
  return "-";
}
```

New method in **`Val`** and **`NumVal`**

```
NEW(BinaryOpExpr)(SubOp::get_instance(), NEW(VarExpr)("x"), NEW(VarExpr)("y"))
```

Parsing Grammar

⟨expr⟩ = ⟨comparg⟩
| ⟨comparg⟩ `==` ⟨expr⟩

⟨comparg⟩ = ⟨addend⟩
| ⟨addend⟩ `+` ⟨comparg⟩

⟨addend⟩ = ⟨multicand⟩
| ⟨multicand⟩ `*` ⟨addend⟩

⟨multicand⟩ = ⟨inner⟩
| ⟨multicand⟩ `(` ⟨expr⟩ `)`

⟨inner⟩ = ⟨number⟩ | `(` ⟨expr⟩ `)` | ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_true` | `_false`
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| `_fun` `(` ⟨variable⟩ `)` ⟨expr⟩

Generalized Precedence Parsing Grammar

⟨expr⟩ = ⟨oparg$_0$⟩

⟨oparg$_P$⟩ = ⟨oparg$_{P+1}$⟩
| ⟨oparg$_{P+1}$⟩ ⟨op$_P$⟩ ⟨oparg$_P$⟩

⟨oparg$_N$⟩ = ⟨multicand⟩
| ⟨oparg$_N$⟩ `(` ⟨expr⟩ `)`

⟨multicand⟩ = ⟨number⟩ | `(` ⟨expr⟩ `)` | ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| `_fun` `(` ⟨variable⟩ `)` ⟨expr⟩

⟨op$_0$⟩ = `==`
| `!=`

⟨op$_1$⟩ = `+`
| `-`

⟨op$_2$⟩ = `*`
= `/`
| `%`

$N = 3$

Generalized Precedence Parsing Grammar

⟨expr⟩ = ⟨oparg$_0$⟩

⟨oparg$_P$⟩ = ⟨oparg$_{P+1}$⟩
| ⟨oparg$_{P+1}$⟩ ⟨op$_P$⟩ ⟨oparg$_P$⟩

⟨oparg$_N$⟩ = ⟨multicand⟩
| ⟨oparg$_N$⟩ `(` ⟨expr⟩ `)`

⟨multicand⟩ = ⟨number⟩ | `(` ⟨expr⟩ `)` | ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| `_fun` `(` ⟨variable⟩ `)` ⟨expr⟩

⟨op$_0$⟩ = `==`
| `!=`

⟨op$_1$⟩ = `+`
| `-`

⟨op$_2$⟩ = `*`
= `/`
| `%`

$N = 3$

```
PTR(Expr) parse_oparg(int precedence, std::istream &in)
```

Parser with Operator Precedence

```
static PTR(Expr) parse_expr(std::istream &in) {
  return parse_oparg(0, in);
}
```

Parser with Operator Precedence

```
static PTR(Expr) parse_oparg(int precedence, std::istream &in) {
  if (precedence >= op_tables.size()) {
    return parse_oparg_n(in);
  } else {
    PTR(Expr) expr = parse_oparg(precedence + 1, in);
    PTR(BinaryOp) op = parse_binary_op(precedence, in);

    if (op != nullptr) {
      PTR(Expr) rhs = parse_oparg(precedence, in);
      return NEW(BinaryOpExpr)(op, expr, rhs);
    } else
      return expr;
  }
}
```

Parser with Operator Precedence

```
static PTR(Expr) parse_oparg(int precedence, std::istream &in) {
  if (precedence >= op_tables.size()) {
    return parse_oparg_n(in);
  } else {
    PTR(Expr) expr = parse_oparg(precedence + 1, in);
    PTR(BinaryOp) op = parse_binary_op(precedence, in);

    if (op != nullptr) {
      PTR(Expr) rhs = parse_oparg(precedence, in);
      return NEW(BinaryOpExpr)(op, expr, rhs);
    } else
      return expr;
  }
}
```

Table that we will define...

Parser with Operator Precedence

```
static PTR(Expr) parse_oparg(int precedence, std::istream &in) {
  if (precedence >= op_tables.size()) {
    return parse_oparg_n(in);
  } else {
    PTR(Expr) expr = parse_oparg(precedence + 1, in);
    PTR(BinaryOp) op = parse_binary_op(precedence, in);

    if (op != nullptr) {
      PTR(Expr) rhs = parse_oparg(precedence, in);
      return NEW(BinaryOpExpr)(op, expr, rhs);
    } else
      return expr;
  }
}
```

Formerly known as `parse_multicand`

Parser with Operator Precedence

```
static PTR(Expr) parse_oparg(int precedence, std::istream &in) {
  if (precedence >= op_tables.size()) {
    return parse_oparg_n(in);
  } else {
    PTR(Expr) expr = parse_oparg(precedence + 1, in);
    PTR(BinaryOp) op = parse_binary_op(precedence, in);

    if (op != nullptr) {
      PTR(Expr) rhs = parse_oparg(precedence, in);
      return NEW(BinaryOpExpr)(op, expr, rhs);
    } else
      return expr;
  }
}
```

Parser with Operator Precedence

```
static PTR(Expr) parse_oparg(int precedence, std::istream &in) {
  if (precedence >= op_tables.size()) {
    return parse_oparg_n(in);
  } else {
    PTR(Expr) expr = parse_oparg(precedence + 1, in);
    PTR(BinaryOp) op = parse_binary_op(precedence, in);

    if (op != nullptr) {
      PTR(Expr) rhs = parse_oparg(precedence, in);
      return NEW(BinaryOpExpr)(op, expr, rhs);
    } else
      return expr;
  }
}
```

Uses `op_tables[precedence]`

Parser with Operator Precedence

```
PTR(BinaryOp) parse_binary_op(int precedence, std::istream &in) {
  for (PTR(BinaryOp) op : op_tables[precedence]) {
    std::string op_name = op->to_string();
    int i = 0;
    char next_ch = peek_after_spaces(in);

    while (next_ch == op_name[i]) {
      consume(in, next_ch);
      if (++i == op_name.length())
        return op;
      next_ch = in.peek();
    }

    // didn't match, put consumed character back
    while (i-- > 0)
      in.putback(op_name[i]);
  }
  return nullptr;
}
```

Parser with Operator Precedence

```
PTR(BinaryOp) parse_binary_op(int precedence, std::istream &in) {
  for (PTR(BinaryOp) op : op_tables[precedence]) {
    std::string op_name = op->to_string();
    int i = 0;
    char next_ch = peek_after_spaces(in);

    while (next_ch == op_name[i]) {
      consume(in, next_ch);
      if (++i == op_name.length())
        return op;
      next_ch = in.peek();
    }

    // didn't match, put consumed character back
    while (i-- > 0)
      in.putback(op_name[i]);
  }
  return nullptr;
}
```

Try each `op` at `precedence`

Operator Table

```
static std::vector<PTR(BinaryOp)> equality_table {
  CompOp::get_instance(), CompNotOp::get_instance()
};

static std::vector<PTR(BinaryOp)> additive_table {
  AddOp::get_instance(), SubOp::get_instance()
};

static std::vector<PTR(BinaryOp)> multiplicative_table {
  MultOp::get_instance(), DivOp::get_instance(), ModOp::get_instance()
};


std::vector<std::vector<PTR(BinaryOp)>> op_tables {
  equality_table, additive_table, multiplicative_table
};
```

Performance

Seconds for `fib(fib)(28)` with GC:

	8k heap	8M heap
`AddExpr`, etc.	`0.79`	`0.54`
`BinaryOpExpr`	`0.85`	`0.56`

Good abstractions sometimes come with a performance penalty

Often, that penalty is worthwhile