# MSDscript

```
_let x = 3
_in x
```

➡ `3`

# MSDscript with Conditionals

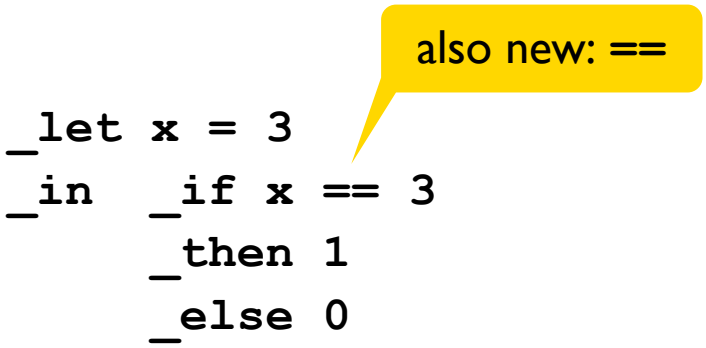
```
_let x = 3
_in  _if x == 3
     _then 1
     _else 0
```

# MSDscript with Conditionals

new: `_if` … `_then` … `_else`

```
_let x = 3
_in  _if x == 3
     _then 1
     _else 0
```

# MSDscript with Conditionals



# MSDscript with Conditionals

```
_let x = 3
_in  _if x == 3
     _then 1
     _else 0
```

➡ `1`

# MSDscript with Conditionals

`_if` and `==` always together?

```
_let x = 3
_in  _if x == 3
     _then 1
     _else 0
```

➡ `1`

# MSDscript with Conditionals

```
_let x = 3
_in  _let same = (x == 3)
     _in _if same
         _then 1
         _else 0
```

# MSDscript with Conditionals

```
_let x = 3
_in  _let same = (x == 3)
     _in _if same
         _then 1
         _else 0
```

== is a new kind of expression

# MSDscript with Conditionals

```
_let x = 3
_in  _let same = (x == 3)
     _in _if same
         _then 1
         _else 0
```

_if ⟨expr⟩ _then ⟨expr⟩ _else

# MSDscript with Conditionals

```
_let x = 3
_in  _let same = (x == 3)
     _in _if same
         _then 1
         _else 0
```

➡ `1`

# MSDscript with Conditionals

```
1 + (_if 3 == 3
      _then 1
      _else 0)
```

➡ `2`

# MSDscript with Conditionals

```
1 + (_if 3 == 3
     _then 1
     _else 0)
```

Like
`( ... ? ... : ... )`
in C-style languages

➡ `2`

# MSDscript Expressions and Values

```
(1 + 2) == 3
```

➡ `_true`

# MSDscript Expressions and Values

```
(1 + 2) == 3
```

➡ `_true`

New kind of ***value***

`(1 + 2) == 3` is an ***expression***

An expression has a ***value*** as determined by `interp`

The value of `(1 + 2) == 3` is `_true`

# MSDscript Expressions and Values

```
(1 + 2) == 3
```

➡ `_true`

New kind of ***value***

`1 + 2` is an ***expression***

An expression has a ***value*** as determined by `interp`

The value of `1 + 2` is `3`

# MSDscript Expressions and Values

```
(1 + 2) == 3
```

➡ `_true`

New kind of ***value***

Some expressions look the same as their values

`3` is an ***expression***

It has the ***value*** `3`

# MSDscript Expressions and Values

```
(1 + 2) == 3
```

➡ `_true`

New kind of ***value***

For now, let's have an expression for each value

`_true` is an ***expression***

It has the ***value*** `_true`

# MSDscript Expressions and Values

```
_if _true
_then 1
_else 0
```

➡ `1`

# MSDscript Expressions and Values

```
(1 + 2) == 0
```

➡ `_false`

# MSDscript with Booleans

⟨expr⟩ = ⟨number⟩
| ⟨boolean⟩ new
| ⟨expr⟩ `==` ⟨expr⟩ new
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩ new

⟨val⟩ = ⟨number⟩
| ⟨boolean⟩ new

# MSDscript with Booleans

⟨expr⟩ = ⟨number⟩
| ⟨boolean⟩ new
| ⟨expr⟩ `==` ⟨expr⟩ new
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩ new

⟨val⟩ = ⟨number⟩
| ⟨boolean⟩ new

⟨boolean⟩ = `_true`
| `_false`

# MSDscript with Booleans

⟨expr⟩ = ⟨number⟩
| ⟨boolean⟩
| ⟨expr⟩ `==` ⟨expr⟩
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩

⟨val⟩ = ⟨number⟩
| ⟨boolean⟩

lower precedence than + or *

# MSDscript with Booleans

⟨expr⟩ = ⟨number⟩ **Num**
| ⟨boolean⟩
| ⟨expr⟩ `==` ⟨expr⟩
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩

⟨val⟩ = ⟨number⟩ **int**
| ⟨boolean⟩

# MSDscript with Booleans

⟨expr⟩ = ⟨number⟩ **NumExpr**
| ⟨boolean⟩
| ⟨expr⟩ `==` ⟨expr⟩
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩

⟨val⟩ = ⟨number⟩ **NumVal**
| ⟨boolean⟩

# MSDscript with Booleans

⟨expr⟩ = ⟨number⟩ **NumExpr**
| ⟨boolean⟩ **BoolExpr**
| ⟨expr⟩ `==` ⟨expr⟩
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩

⟨val⟩ = ⟨number⟩ **NumVal**
| ⟨boolean⟩ **BoolVal**

# MSDscript with Booleans

| | | | | |
|---|---|---|---|---|
| `Expr` | ⟨expr⟩ | = | ⟨number⟩ | `NumExpr` |
| | | \| | ⟨boolean⟩ | `BoolExpr` |
| | | \| | ⟨expr⟩ `==` ⟨expr⟩ | `EqExpr` |
| | | \| | ⟨expr⟩ `+` ⟨expr⟩ | `AddExpr` |
| | | \| | ⟨expr⟩ `*` ⟨expr⟩ | `MultExpr` |
| | | \| | ⟨variable⟩ | `VarExpr` |
| | | \| | `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩ | `LetExpr` |
| | | \| | `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩ | `IfExp` |
| `Val` | ⟨val⟩ | = | ⟨number⟩ | `NumVal` |
| | | \| | ⟨boolean⟩ | `BoolVal` |

# Refactoring MSDscript

**Original**

---

```
int
```

---

```
Expr
Num, Add, Mult
Var, Let

int interp()
Expr* subst(...)
void print(...), …
```

**Refactored**

---

```
Val
NumVal, BoolVal
Expr* to_expr(), …
```

---

```
Expr
NumExpr, AddExpr, MultExpr
VarExpr, LetExpr
BoolExpr, EqExpr, IfExpr
Val* interp()
Expr* subst(...)
void print(...), …
```

# Evaluation

```
1 + 2
```
➡ `3`

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
CHECK( parse_str("1 + 2")->interp()
       ->equals(new NumVal(3)) );
```

```
CHECK( parse_str("1 + 2")->interp()->to_string() == "3" );
```

```
static std::string run(std::string s) {
  return parse_str(s)->interp()->to_string();
}
CHECK( run("1 + 2") == "3" );
```

# Evaluation

```
1 == 2
➡ _false
```

```
CHECK( (new EqExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new BoolVal(false)) );
```

```
CHECK( parse_str("1 == 2")->interp()
       ->equals(new BoolVal(false)) );
```

```
CHECK( parse_str("1 == 2")->interp()->to_string() == "_false" );
```

```
static std::string run(std::string s) {
  return parse_str(s)->interp()->to_string();
}
CHECK( run("1 == 2") == "_false" );
```

# Evaluation

```
1 == 2
➡ _false
```

```
CHECK( (new EqExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new BoolVal(false)) );
```

```
CHECK( parse_str("1 == 2")->interp()
       ->equals(new BoolVal(false)) );
```

```
CHECK( parse_str("1 == 2")->interp()->to_string() == "_false" );
```

```
static std::string run(std::string s) {
  return parse_str(s)->interp()->to_string();
}
CHECK( run("1 == 2") == "_false" );
```

# Parsing with Conditions and Comparisons

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

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

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

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

# Parsing with Conditions and Comparisons

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

parse_comparg    ⟨comparg⟩   = ⟨addend⟩
                             | ⟨addend⟩ + ⟨comparg⟩

parse_addend     ⟨addend⟩    = ⟨multicand⟩
                             | ⟨multicand⟩ * ⟨addend⟩

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

# Parsing with Conditions and Comparisons

⟨expr⟩ = ⟨comparg⟩
| ⟨comparg⟩ **==** ⟨expr⟩

⟨comparg⟩ = ⟨addend⟩
| ⟨addend⟩ **+** ⟨comparg⟩

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

⟨multicand⟩ = ⟨number⟩
| **(** ⟨expr⟩ **)**
| ⟨variable⟩
| **_let** ⟨variable⟩ **=** ⟨expr⟩ **_in** ⟨expr⟩
| **_true**
| **_false**
| **_if** ⟨expr⟩ **_then** ⟨expr⟩ **_else** ⟨expr⟩

```
parse_multicand() {
 if (isdigit(next_ch))
   parse_number()
 else if (next_ch == '(')
   parse_expr()
 else if (isalpha(next_ch))
   parse_variable()
 else if (next_ch == '_') {
   kw = parse_keyword()
   if (kw == "_let") parse_let()
   else if (kw == "_false") ...
   else if (kw == "_true") ...
   else if (kw == "_if") parse_if()
}
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
}
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
  ....    lhs    ....
  ....    rhs    ....
}
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
  ....    lhs->interp() ....
  ....    rhs->interp() ....
}
```

When you have an **Expr**, **interp** it

```
(3 * 4) + 1        1 + (_if _true _then 1 _else 2)
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
  ....    lhs->interp() ....
  ....    rhs->interp() ....
}
```

```
Val *NumExpr::interp() {
  return new NumVal(rep);
}
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::inter
  ....   lhs->inter
  ....   rhs->inter
}
```

```
class NumExpr : public Expr {
  int rep;
  NumExpr(int rep) {
    this->rep = rep;
  }
};
```

```
Val *NumExpr::interp() {
  return new NumVal(rep);
}
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
  return lhs->interp()
   ->add_to(rhs->interp());
}
```

Object-oriented style: let a `Val` decide about operations

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
  return lhs->interp()
   ->add_to(rhs->interp());
}
```

```
Val *NumVal::add_to(Val *other_val) {
  NumVal *other_num = dynamic_cast<NumVal*>(other_val);
  if (other_num == NULL) throw std::runtime_error("add of non-number");
  return new NumVal(rep + other_num->rep);
}
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
  return lhs->interp()
   ->add_to(rhs->interp());
}
```

Allowed because only **NumVal** makes sense

```
Val *NumVal::add_to(Val *other_val) {
  NumVal *other_num = dynamic_cast<NumVal*>(other_val);
  if (other_num == NULL) throw std::runtime_error("add of non-number");
  return new NumVal(rep + other_num->rep);
}
```

# Evaluation

```
1 + 2
➡ 3
```

```
CHECK( (new AddExpr(new NumExpr(1), new NumExpr(2)))->interp()
       ->equals(new NumVal(3)) );
```

```
Val *AddExpr::interp() {
  return lhs->interp()
   ->add_to(rhs->interp());
}
```

```
Val *BoolVal::add_to(Val *other_val) {
  throw std::runtime_error("add of non-number");
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {
   .... lhs ....
   .... rhs ....
   .... body ....
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {
   .... lhs ....
   .... rhs->interp() ....
   .... body->interp() ....
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {
   .... lhs ....
   .... rhs->interp() ....
   .... body->subst(   ...   )->interp() ....
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {

  .... rhs->interp() ....
  .... body->subst(lhs, ...)->interp()....
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {

  Val *rhs_val = rhs->interp();
  .... body->subst(lhs,  ...  rhs_val  ...)->interp()
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
class Val {
  virtual Expr *to_expr() = 0;
}
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {

  Val *rhs_val = rhs->interp();
  .... body->subst(lhs, rhs_val->to_expr())->interp()
}
```

# Evaluation

```
_let x = 2+3
_in x*x
```

```
CHECK( (new LetExpr("x",
                    new AddExpr(new NumExpr(2), new NumExpr(3)),
                    new MultExpr(new VarExpr("x"), new VarExpr("x"))))
      ->interp()
      ->equals(new NumVal(25)) );
```

```
Val *LetExpr::interp() {

  Val *rhs_val = rhs->interp();
  return body->subst(lhs, rhs_val->to_expr())->interp();
}
```

# Conditional Evaluation

```
_if _true
_then 1
_else 2
```

```
CHECK( (new IfExpr(new BoolExpr(true),
                   new NumExpr(1),
                   new NumExpr(2)))->interp()
      ->equals(new NumVal(1)) );
```

```
Val *IfExpr::interp() {
}
```

# Conditional Evaluation

```
_if _true
_then 1
_else 2
```

```
CHECK( (new IfExpr(new BoolExpr(true),
                   new NumExpr(1),
                   new NumExpr(2)))->interp()
      ->equals(new NumVal(1)) );
```

```
Val *IfExpr::interp() {
  .... test_part ....
  .... then_part ....
  .... else_part ....

}
```

# Conditional Evaluation

```
_if _true
_then 1
_else 2
```

```
CHECK( (new IfExpr(new BoolExpr(true),
                   new NumExpr(1),
                   new NumExpr(2)))->interp()
      ->equals(new NumVal(1)) );
```

```
Val *IfExpr::interp() {
  .... test_part->interp() ....
  .... then_part->interp() ....
  .... else_part->interp() ....

}
```

# Conditional Evaluation

```
_if _true
_then 1
_else 2
```

```
CHECK( (new IfExpr(new BoolExpr(true),
                   new NumExpr(1),
                   new NumExpr(2)))->interp()
      ->equals(new NumVal(1)) );
```

```
Val *IfExpr::interp() {
  .... test_part->interp()->is_true() ....
  .... then_part->interp() ....
  .... else_part->interp() ....

}
```

# Conditional Evaluation

```
_if _true
_then 1
_else 2
```

```
class Val {
  virtual bool is_true() = 0;
}
```

```
CHECK( (new IfExpr(new BoolExpr(true),
                   new NumExpr(1),
                   new NumExpr(2)))->interp()
      ->equals(new NumVal(1)) );
```

```
Val *IfExpr::interp() {
  .... test_part->interp()->is_true() ....
  .... then_part->interp() ....
  .... else_part->interp() ....

}
```

# Conditional Evaluation

```
_if _true
_then 1
_else 2
```

```
CHECK( (new IfExpr(new BoolExpr(true),
                   new NumExpr(1),
                   new NumExpr(2)))->interp()
      ->equals(new NumVal(1)) );
```

```
Val *IfExpr::interp() {
  if (test_part->interp()->is_true())
    return then_part->interp();
  else
    return else_part->interp();
}
```