# Loops

In C/C++:

```
int countdown(int n) {
  int i = n;
  while (i > 0)
    i = i - 1;
  return i;
}
```

In MSDscript?

# Looping by Recursion

```
_let countdown = _fun(countdown)
                  _fun(n)
                   _if n == 0
                   _then 0
                   _else countdown(countdown)(n + -1)
_in countdown(countdown)(1000000)
```

***countdown*(*countdown*)(1000000)**

# Looping by Recursion

```
_let countdown = _fun(countdown)
                  _fun(n)
                   _if n == 0
                   _then 0
                   _else countdown(countdown)(n + -1)
_in countdown(countdown)(1000000)
```

```
countdown(countdown)(1000000)
countdown(countdown)(999999)
```

# Looping by Recursion

```
_let countdown = _fun(countdown)
                  _fun(n)
                   _if n == 0
                   _then 0
                   _else countdown(countdown)(n + -1)
_in countdown(countdown)(1000000)
```

```
countdown(countdown)(1000000)
countdown(countdown)(999999)
countdown(countdown)(999998)
```

# Looping by Recursion

```
_let countdown = _fun(countdown)
                  _fun(n)
                   _if n == 0
                   _then 0
                   _else countdown(countdown)(n + -1)
_in countdown(countdown)(1000000)
```

```
countdown(countdown)(1000000)
countdown(countdown)(999999)
countdown(countdown)(999998)
...
```

# Looping by Recursion

```
_let countdown = _fun(countdown)
                  _fun(n)
                   _if n == 0
                   _then 0
                   _else countdown(countdown)(n + -1)
_in countdown(countdown)(1000000)
```

```
countdown(countdown)(1000000)
countdown(countdown)(999999)
countdown(countdown)(999998)
...
countdown(countdown)(0)
```

# Looping by Recursion

```
_let countdown = _fun(countdown)
                  _fun(n)
                   _if n == 0
                   _then 0
                   _else countdown(countdown)(n + -1)
_in countdown(countdown)(1000000)
```

```
countdown(countdown)(1000000)
countdown(countdown)(999999)
countdown(countdown)(999998)
...
countdown(countdown)(0)
0
```

# Looping by Recursion

```
_let countdown = _fun(countdown)
                  _fun(n)
                   _if n == 0
                   _then 0
                   _else countdown(countdown)(n + -1)
_in countdown(countdown)(1000000)
```

*⇒ crash due to stack overflow*

# Looping by Recursion

```
...
```

```
FunVal::call
  PTR(Env) new_env = NEW(ExtendedEnv)(formal_arg, actual_arg, env);
➡ return body->interp(new_env);
```

```
FunVal::call
  PTR(Env) new_env = NEW(ExtendedEnv)(formal_arg, actual_arg, env);
➡ return body->interp(new_env);
```

```
FunVal::call
  PTR(Env) new_env = NEW(ExtendedEnv)(formal_arg, actual_arg, env);
➡ return body->interp(new_env);
```

```
NumExpr::interp
➡ return NEW(VumVal)(rep);
```

# Deep Recursion

```
_let count = _fun(count)
              _fun(n)
                _if n == 0
                _then 0
                _else 1 + count(count)(n + -1)
_in count(count)(100000)
```

***count*(*count*)(100000)**

# Deep Recursion

```
_let count = _fun(count)
              _fun(n)
                _if n == 0
                _then 0
                _else 1 + count(count)(n + -1)
_in count(count)(100000)
```

```
count(count)(100000)
1 + count(count)(99999)
```

# Deep Recursion

```
_let count = _fun(count)
              _fun(n)
                _if n == 0
                _then 0
                _else 1 + count(count)(n + -1)
_in count(count)(100000)
```

```
count(count)(100000)
1 + count(count)(99999)
1 + 1 + count(count)(99998)
```

# Deep Recursion

```
_let count = _fun(count)
              _fun(n)
                _if n == 0
                _then 0
                _else 1 + count(count)(n + -1)
_in count(count)(100000)
```

```
count(count)(100000)
1 + count(count)(99999)
1 + 1 + count(count)(99998)
...
```

# Deep Recursion

```
_let count = _fun(count)
              _fun(n)
                _if n == 0
                _then 0
                _else 1 + count(count)(n + -1)
_in count(count)(100000)
```

```
count(count)(100000)
1 + count(count)(99999)
1 + 1 + count(count)(99998)
...
1 + 1 + ... count(count)(0)
```

# Deep Recursion

```
_let count = _fun(count)
              _fun(n)
                _if n == 0
                _then 0
                _else 1 + count(count)(n + -1)
_in count(count)(100000)
```

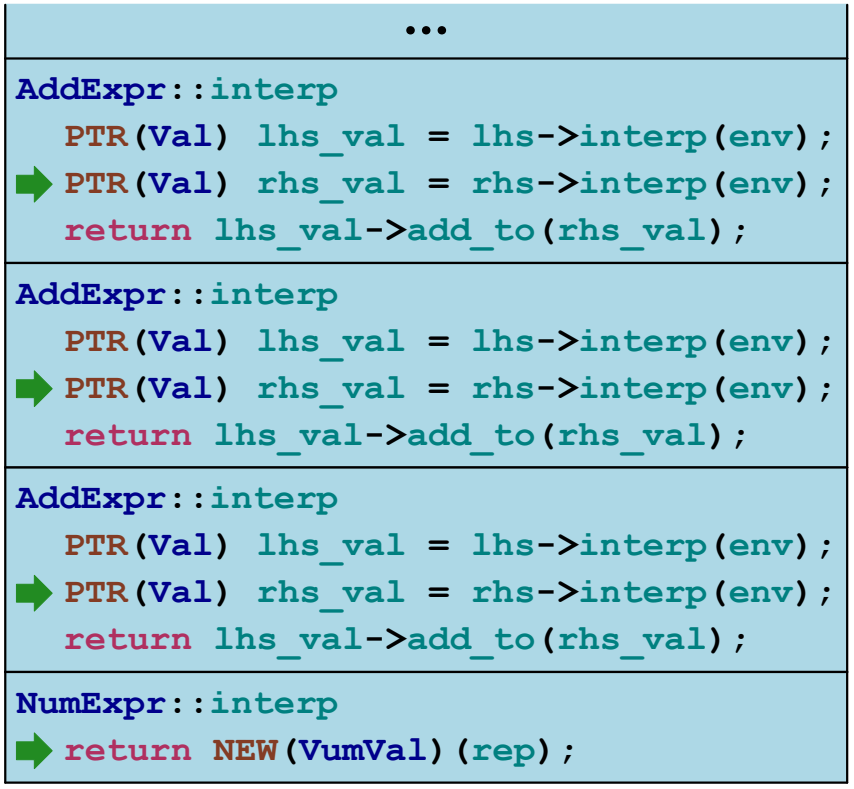
```
count(count)(100000)
1 + count(count)(99999)
1 + 1 + count(count)(99998)
...
1 + 1 + ... count(count)(0)
1 + 1 + ... 0
```

# Deep Recursion

```
_let count = _fun(count)
              _fun(n)
                _if n == 0
                _then 0
                _else 1 + count(count)(n + -1)
_in count(count)(100000)
```

⇒ *crash due to stack overflow*

# Deep Recursion



# Improving the Interpreter

- Make functions calls work for loops, like algebra-style simplification

  So, `countdown(countdown)(`*N*`)` should run in constant space for any *N*

- Use available memory, instead of just available stack space

  So, `count(count)(`*N*`)` should work for *N* > 100,000

⇒ cannot recursively call `interp`

# Interpret Steps

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

# Interpret Steps

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

first

# Interpret Steps

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

first

second

# Interpret Steps

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

first third second

```
PTR(Val) AddExpr::interp(PTR(Env) env) {
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

# Interpret Steps

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

first third second

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

first

# Interpret Steps

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

first third second

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

first second

# Interpret Steps

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

first third second

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

first second third

# Interpretation for the Forgetful

```
PTR(Val) AddExpr::interp(PTR(Env) env) {
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

# Interpretation for the Forgetful

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp lhs env`
Then ...

# Interpretation for the Forgetful

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then ...

Remember `rhs`
`env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then ...

# Interpretation for the Forgetful

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then ...

Remember `rhs` `env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then ...

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

We use `continue` in place of `return` for reasons that will become clear...

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then ...

Remember `rhs` `env`

---

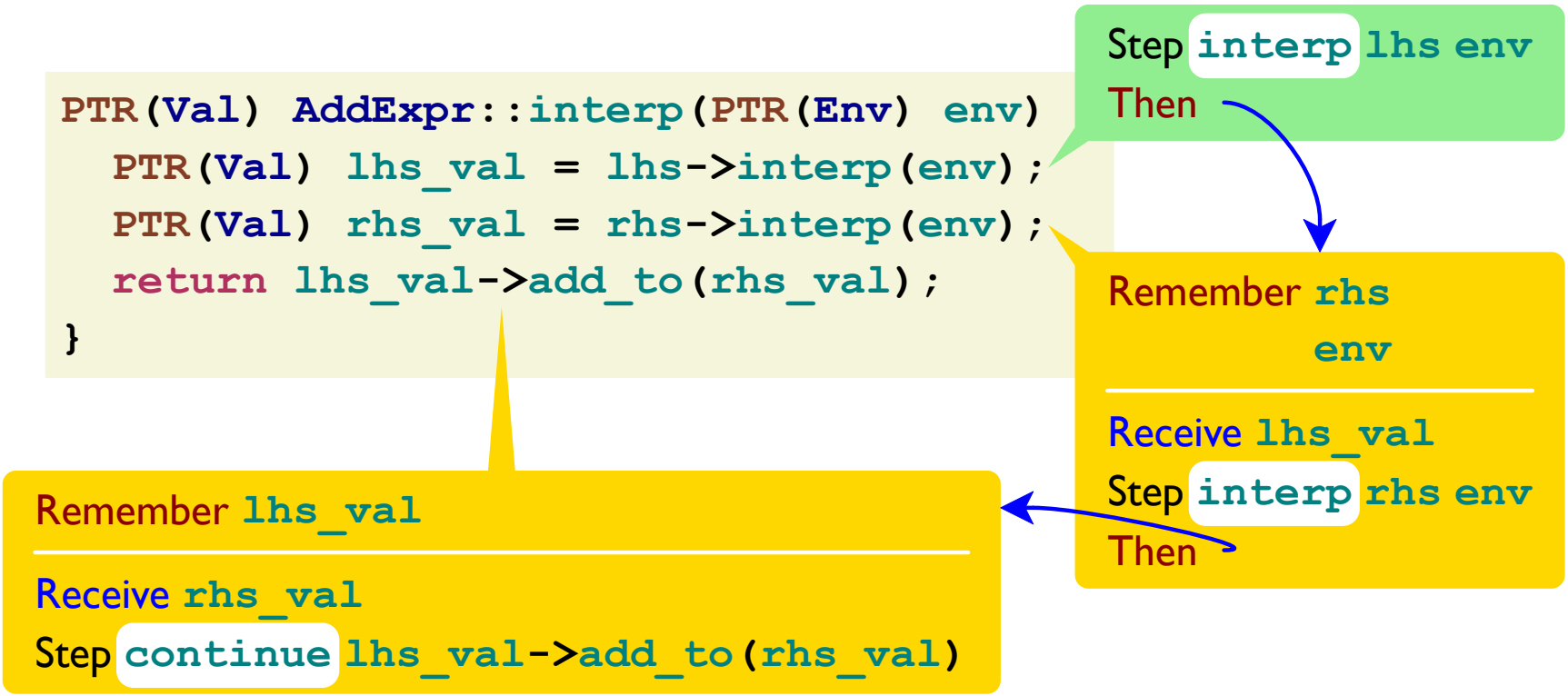
Receive `lhs_val`
Step `interp` `rhs` `env`
Then ...

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful



# Interpretation for the Forgetful

Each balloon has one step: `interp` or `continue`

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs` `env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

A Remember/Receive balloon is a continuation

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs`
`env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

A Remember/Receive balloon is a continuation

... always created by an `interp` step...

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs`
`env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

A Remember/Receive balloon is a continuation

... but not *every* `interp` step

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs` `env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

A step with Then creates a continuation

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs` `env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

A step with Then creates a continuation

Remember parts as fields of the continuation

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs` `env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

Interpret `x + 2` with `x = 3`

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs` `env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

Interpret `x + 2` with `x = 3`

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

`x = 3`

`x`

`2`

Step `interp` `lhs` `env`
Then

Remember `rhs` `env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

Interpret `x + 2` with `x = 3`

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

x = 3

x

2

Step `interp lhs env`
Then

x = 3

Remember `rhs`
`env`

---

Receive `lhs_val`
Step `interp rhs env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

Interpret `x + 2` with `x = 3`

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

x = 3

x

x = 3

Step `interp` `lhs` `env`
Then

2

Remember `rhs` 2
`env` `x = 3`

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

Interpret `x + 2` with `x = 3`

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

x = 3

x

2

x = 3

Step `interp lhs env`
Then

Remember `rhs` 2
`env` x = 3

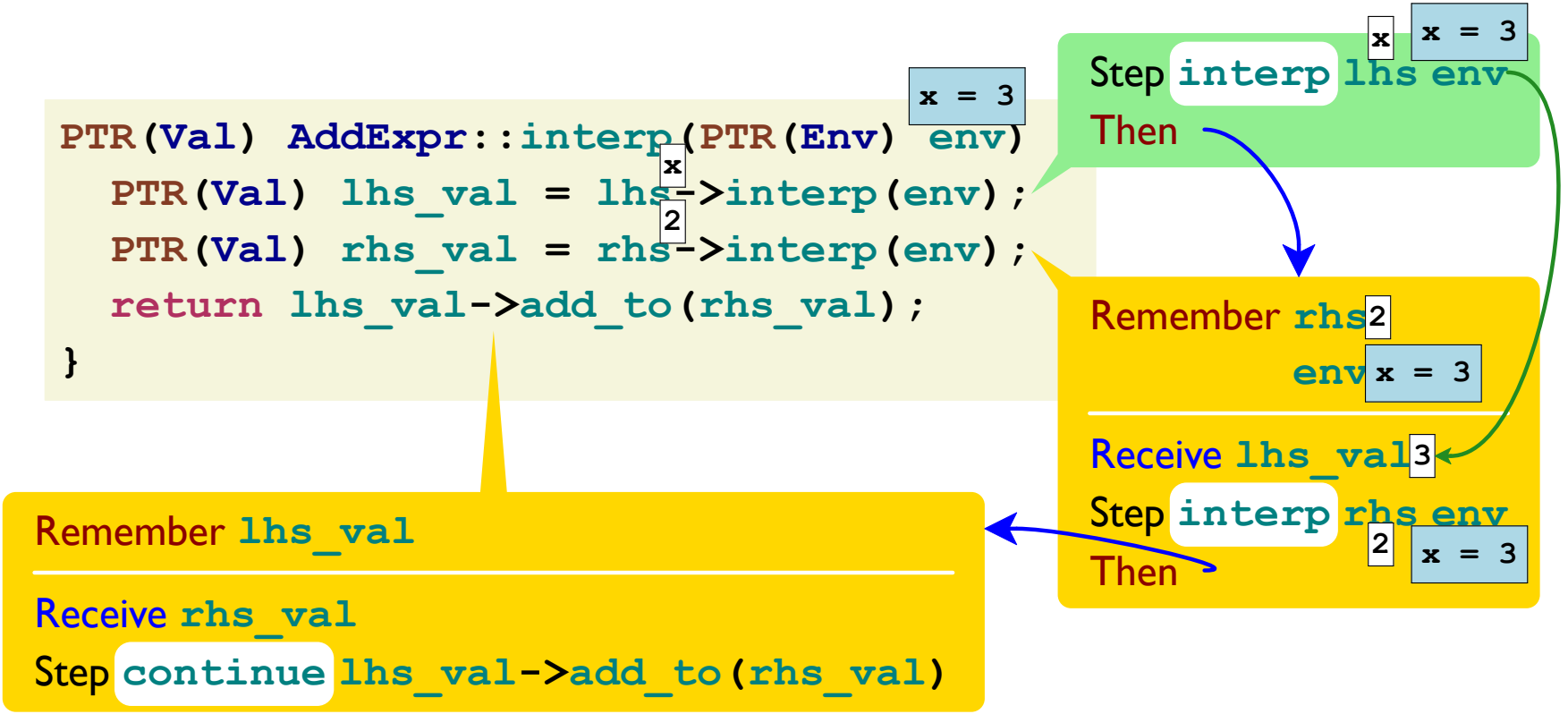
Receive `lhs_val` 3
Step `interp rhs env`
Then

Remember `lhs_val`

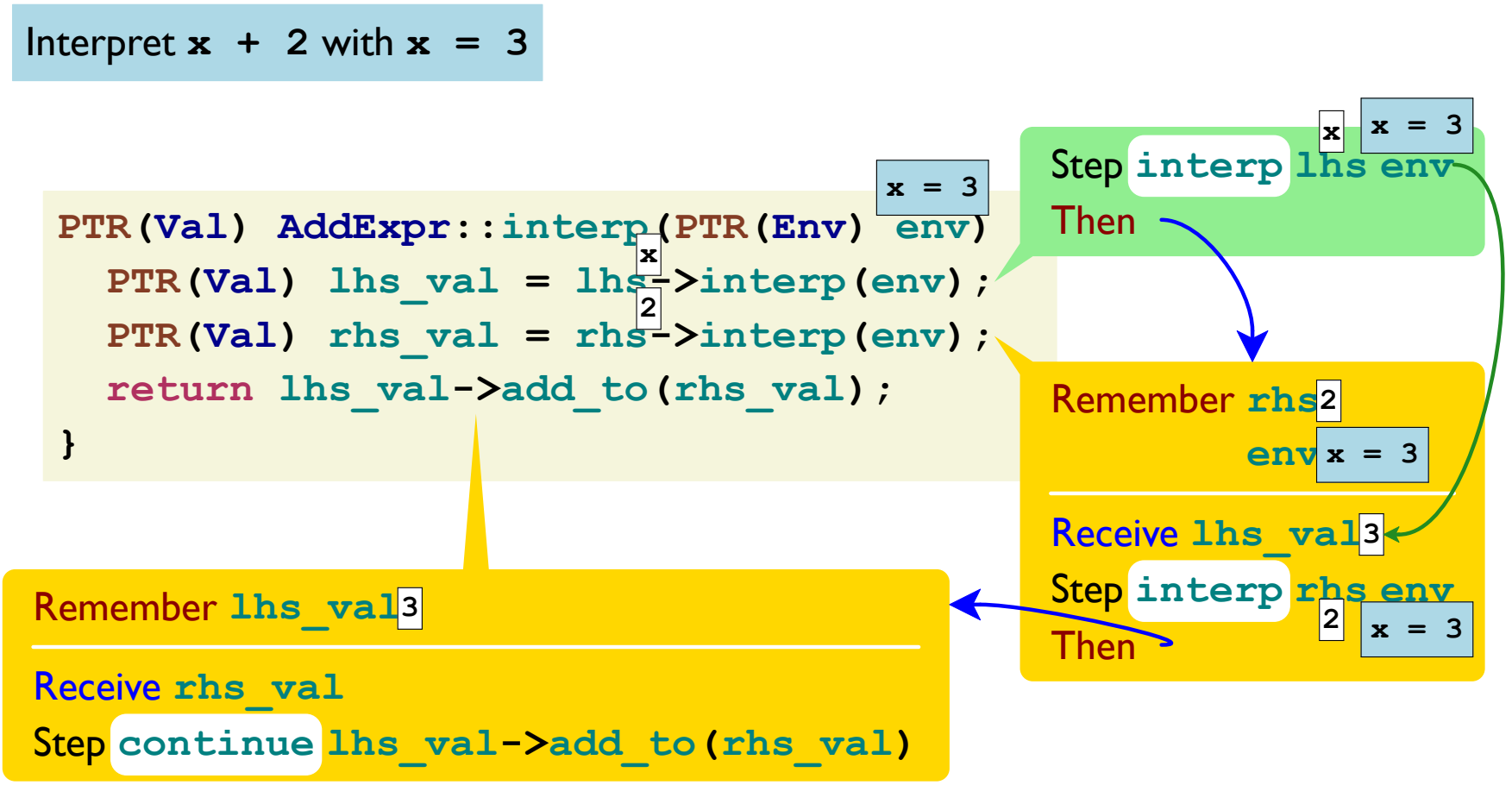
Receive `rhs_val`
Step `continue lhs_val->add_to(rhs_val)`

# Interpretation for the Forgetful

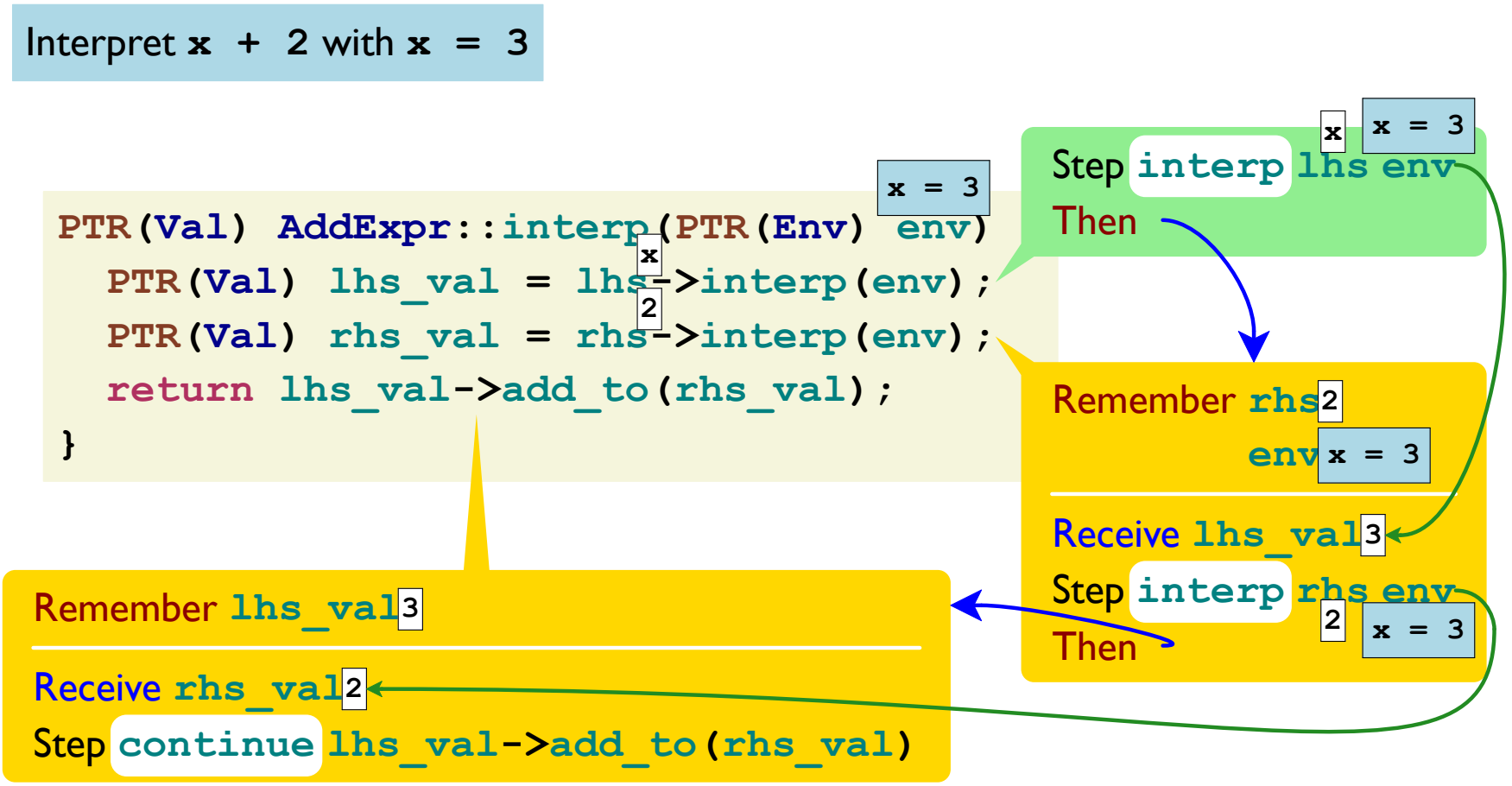
Interpret `x + 2` with `x = 3`



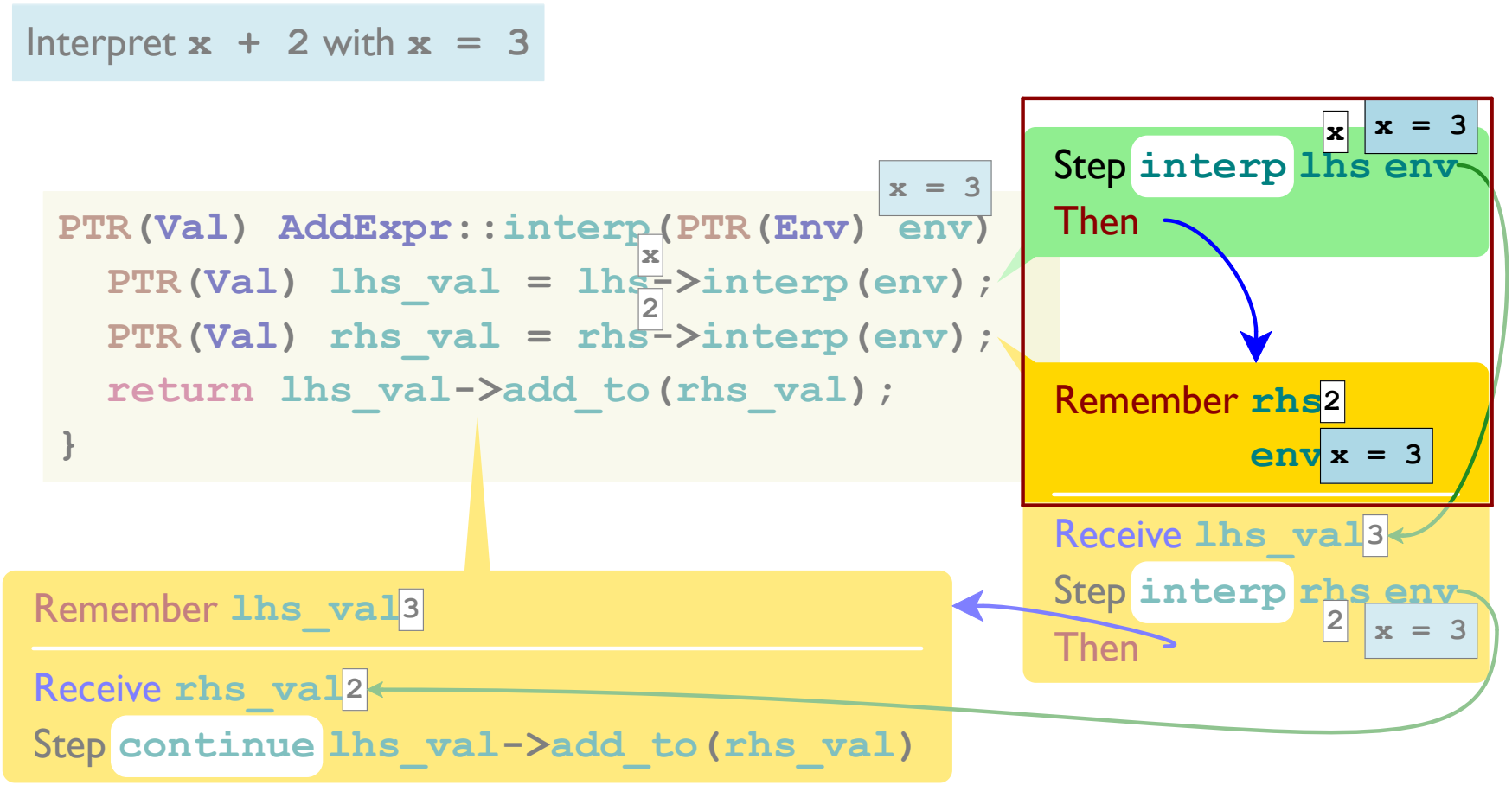
# Interpretation for the Forgetful



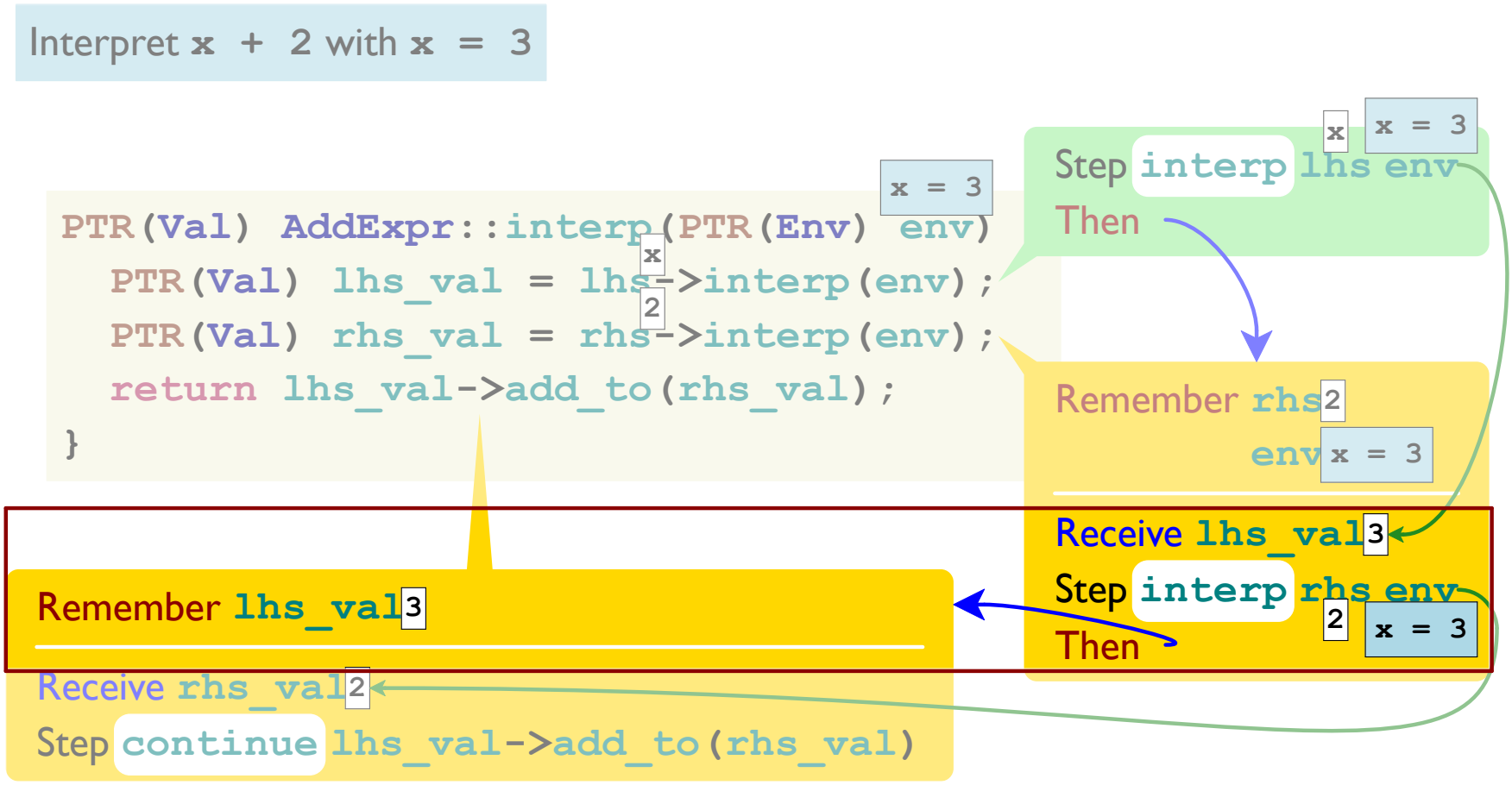
# Interpretation for the Forgetful



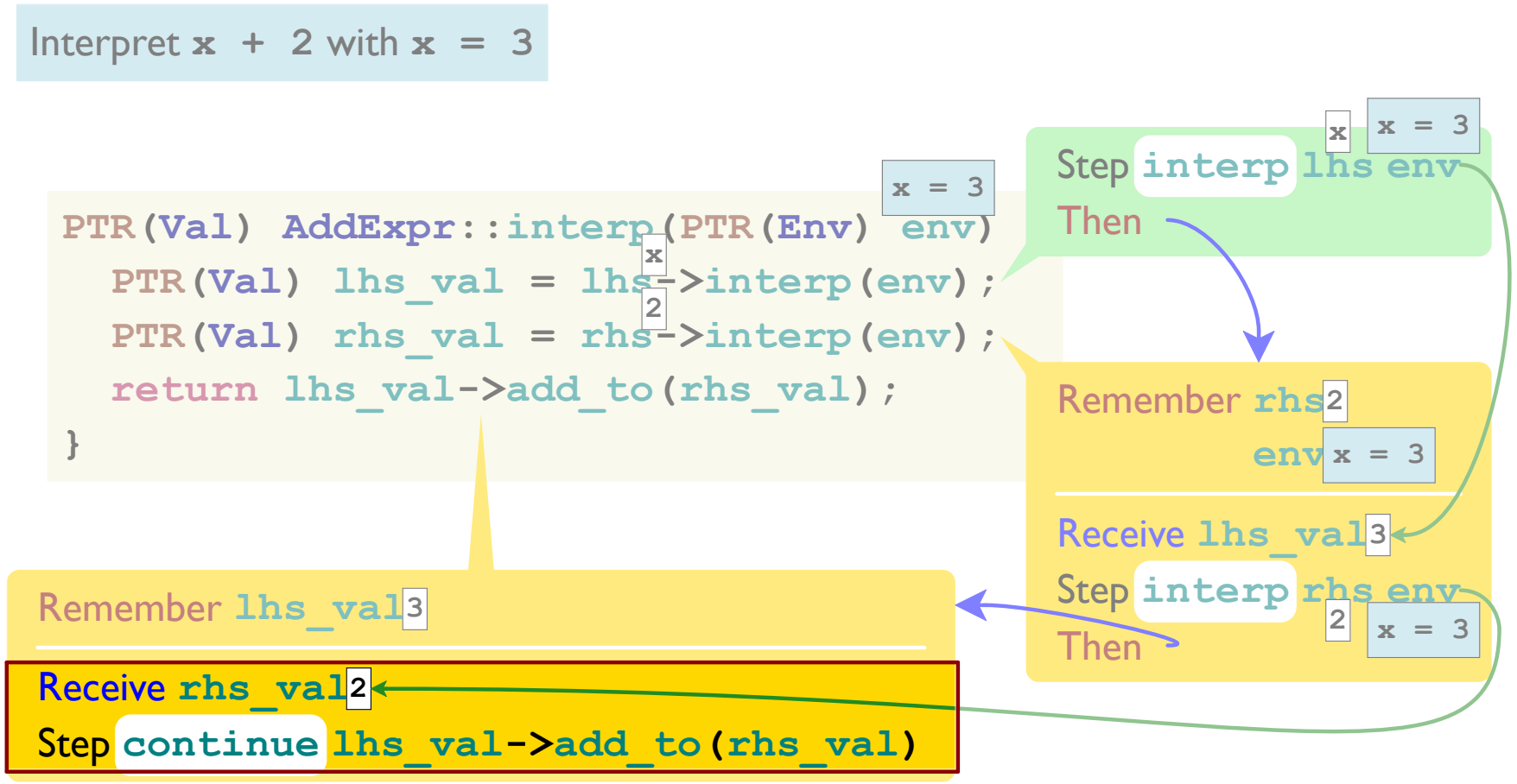
# Interpretation for the Forgetful



# Interpretation for the Forgetful

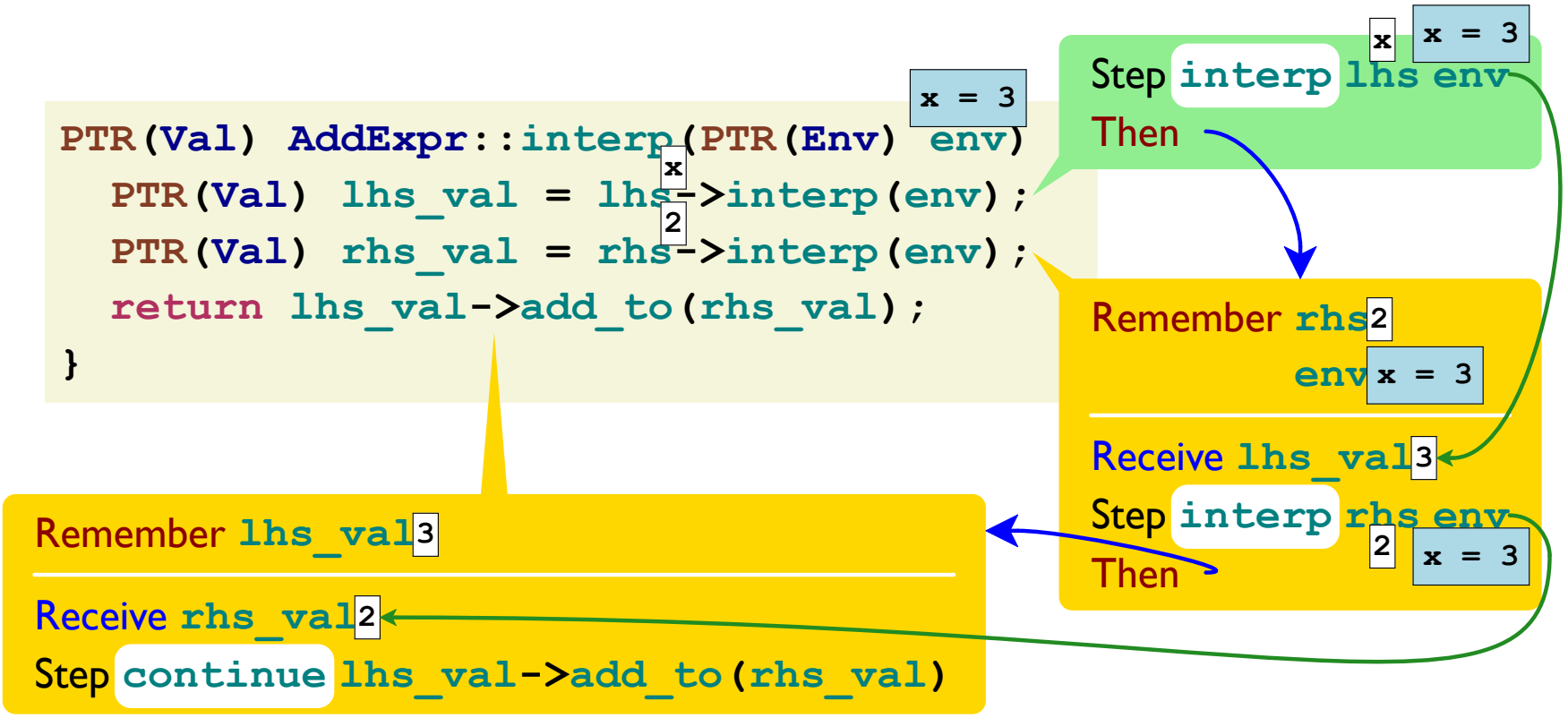


# Interpretation for the Forgetful

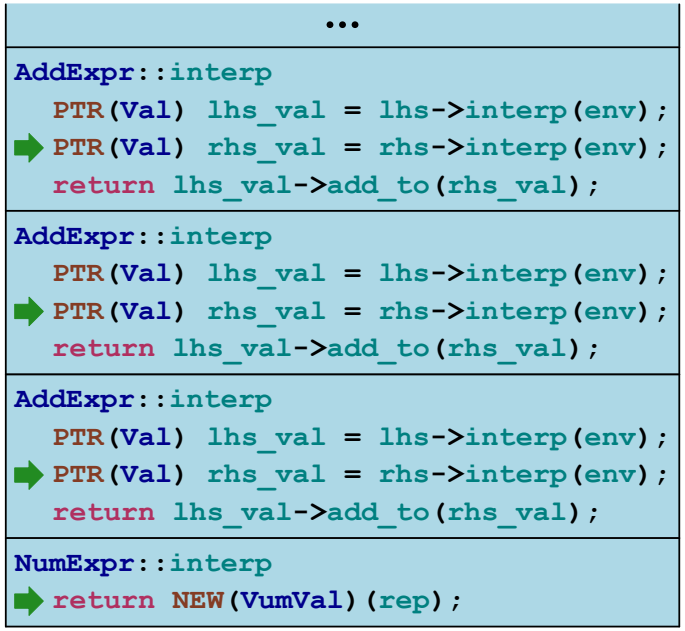


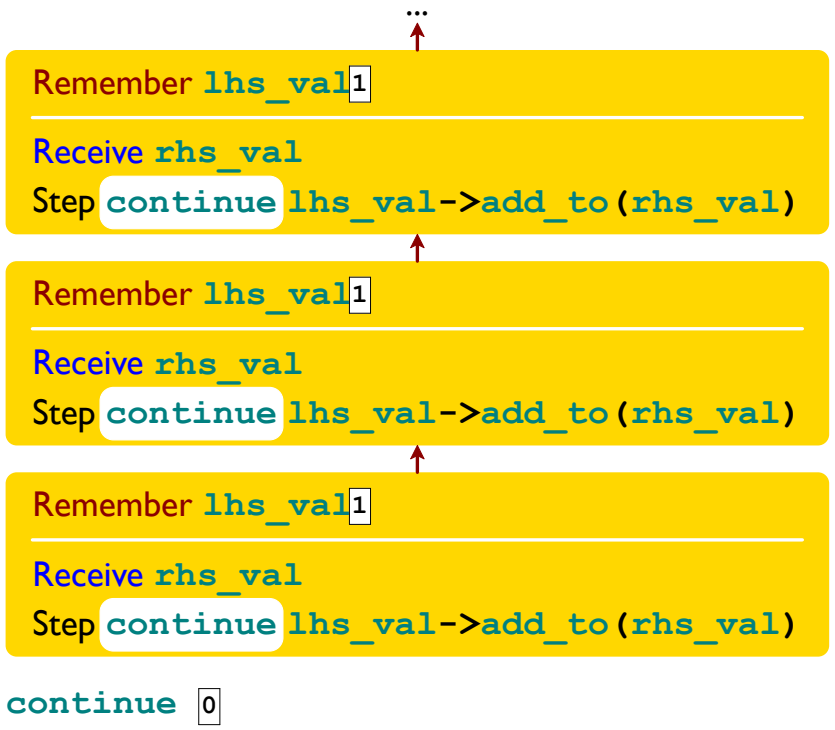
# Interpretation for the Forgetful

A continuation bubble plays the same role as a stack frame

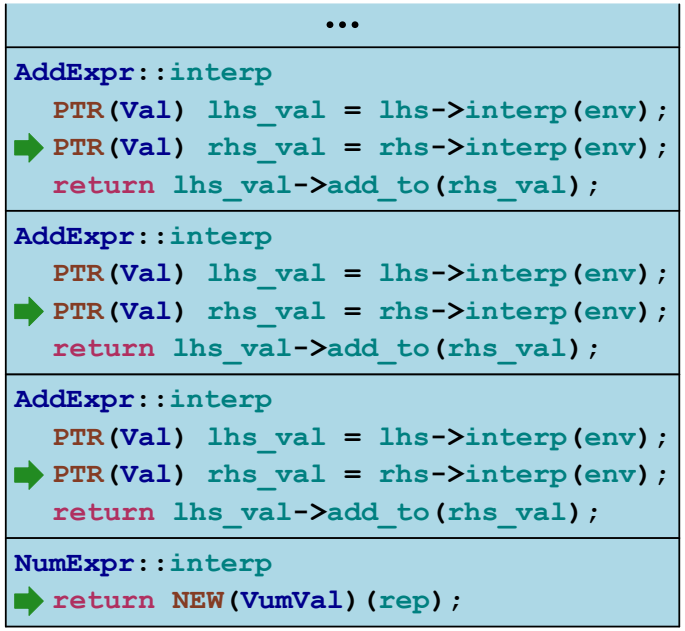


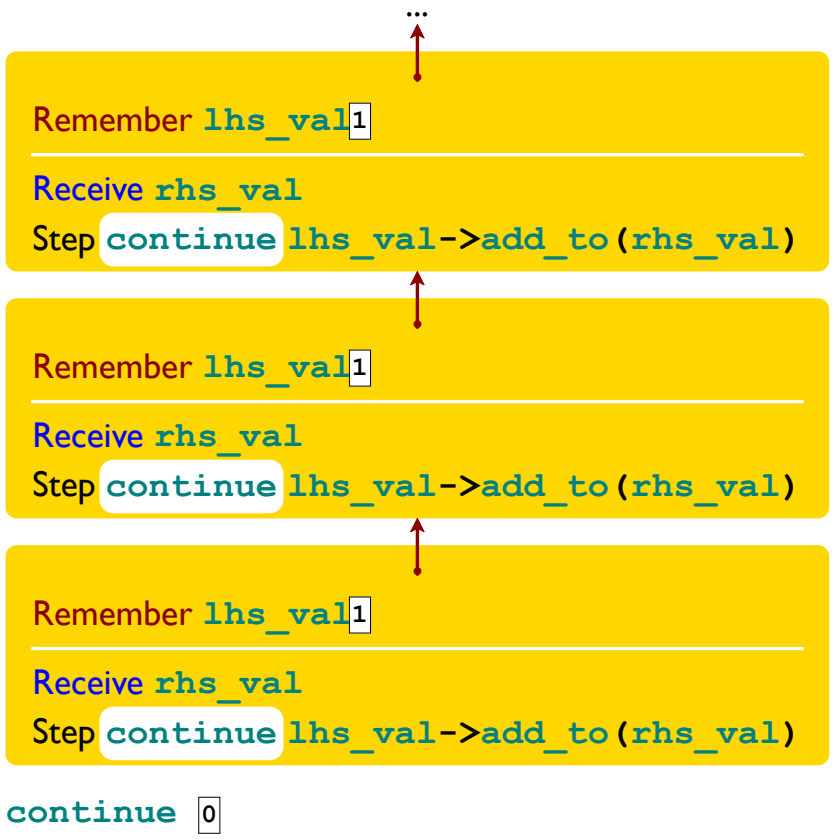
# From Stack to Continuation



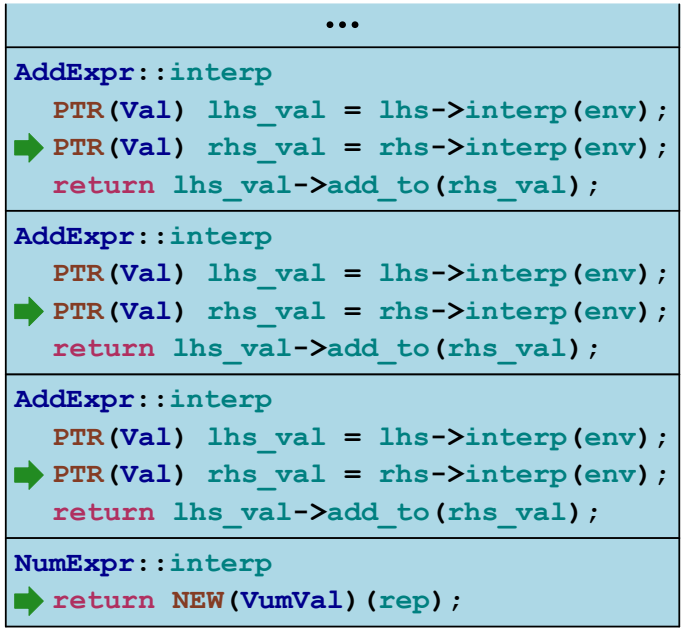



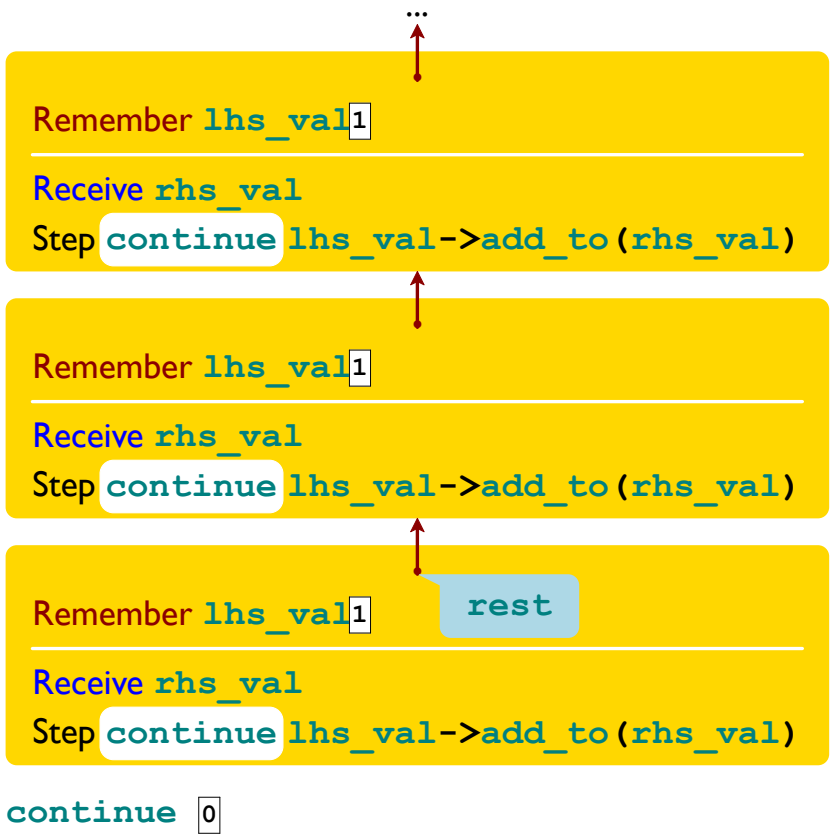
# From Stack to Continuation



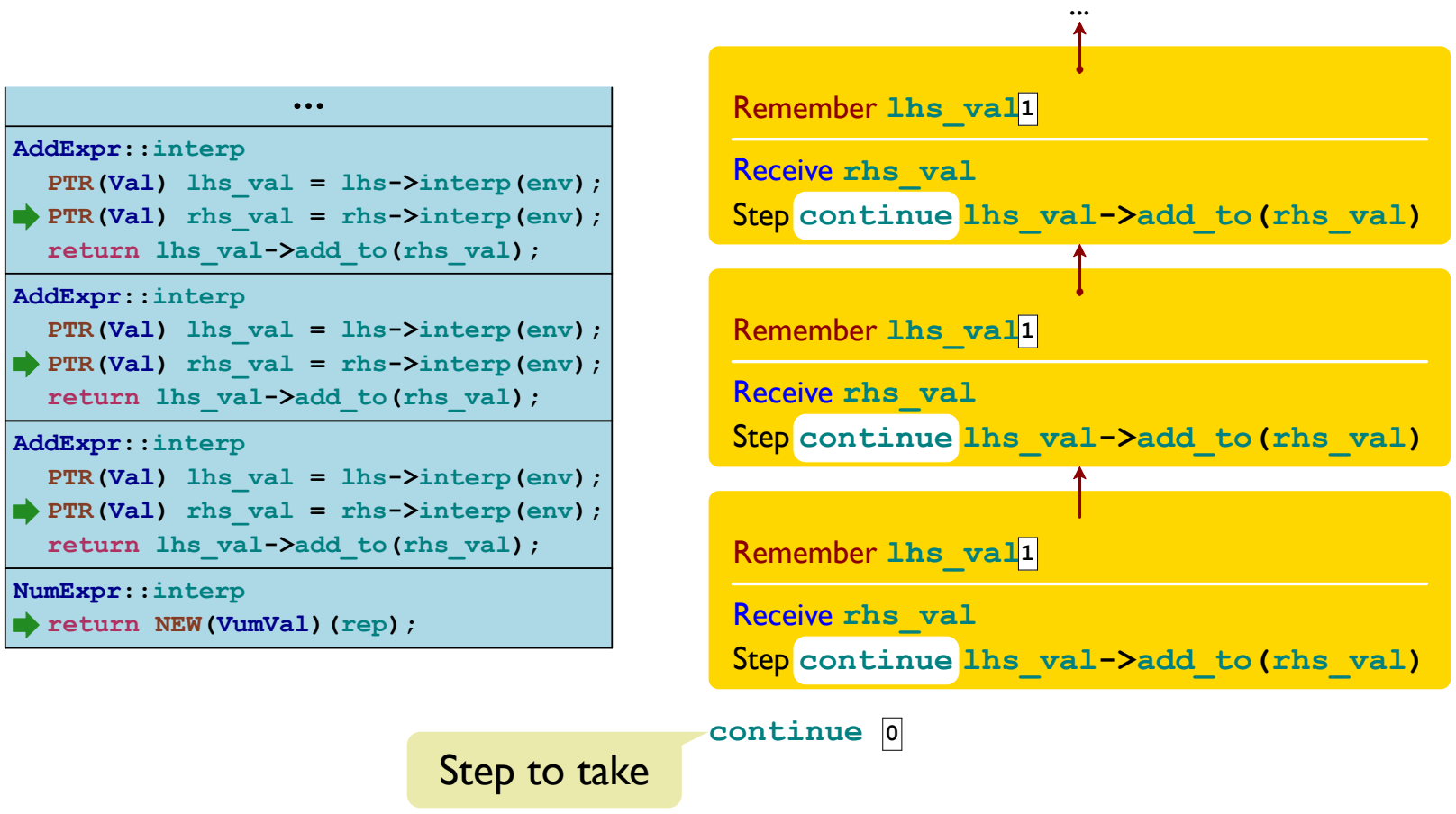



# From Stack to Continuation

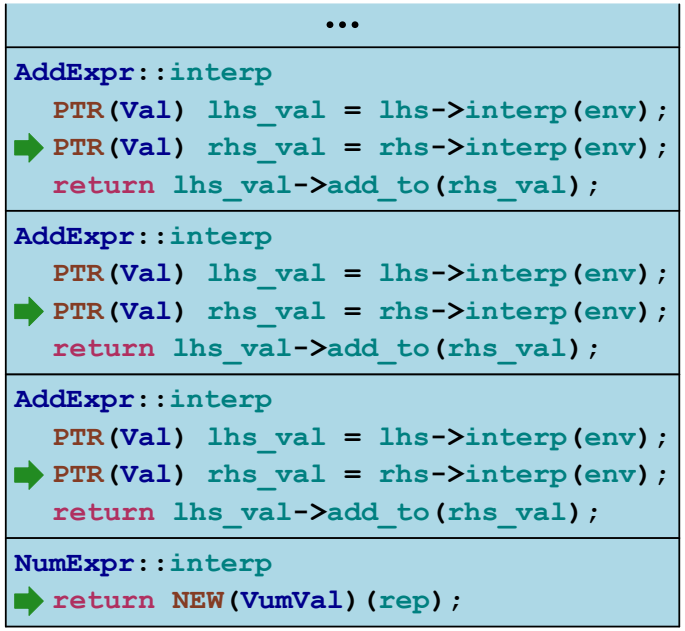


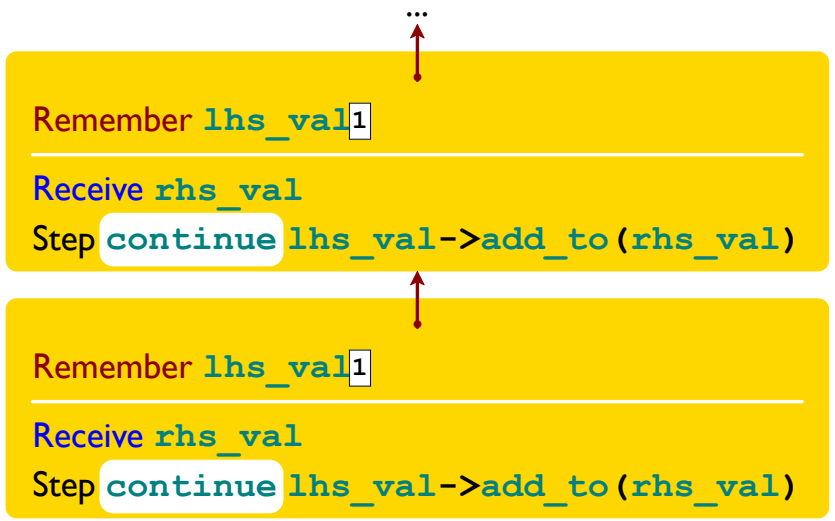



# From Stack to Continuation



# From Stack to Continuation



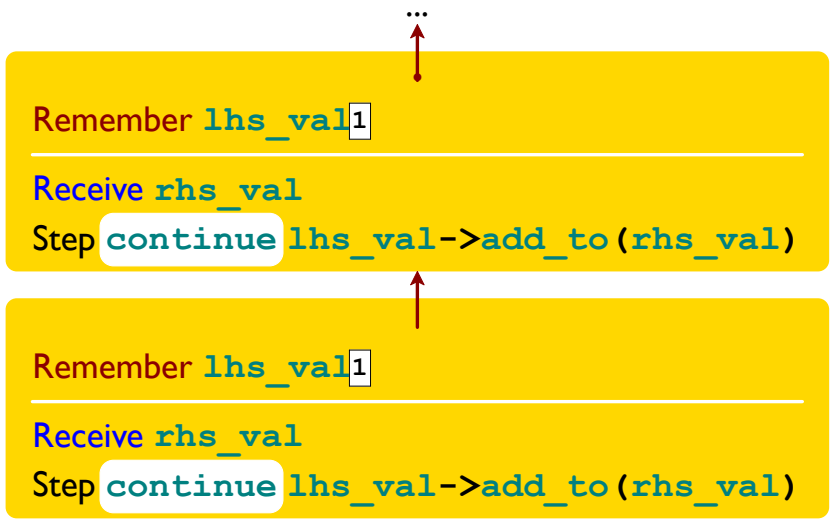

Remember `lhs_val` 1

Receive `rhs_val` 0

Step `continue lhs_val->add_to(rhs_val)`

# From Stack to Continuation

```
...
AddExpr::interp
  PTR(Val) lhs_val = lhs->interp(env);
➡ PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
AddExpr::interp
  PTR(Val) lhs_val = lhs->interp(env);
➡ PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
AddExpr::interp
  PTR(Val) lhs_val = lhs->interp(env);
➡ PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
NumExpr::interp
➡ return NEW(VumVal)(rep);
```

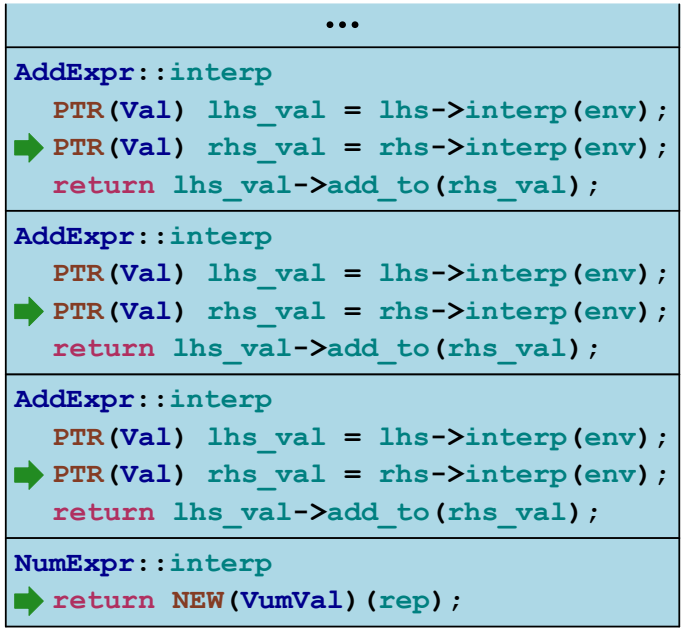


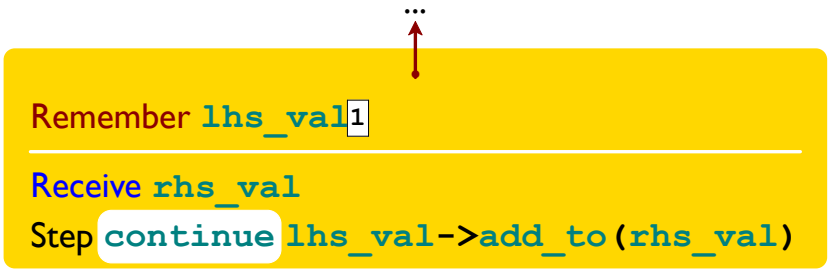

Remember `lhs_val` 1

Receive `rhs_val` 0

Step `continue` 1

# From Stack to Continuation



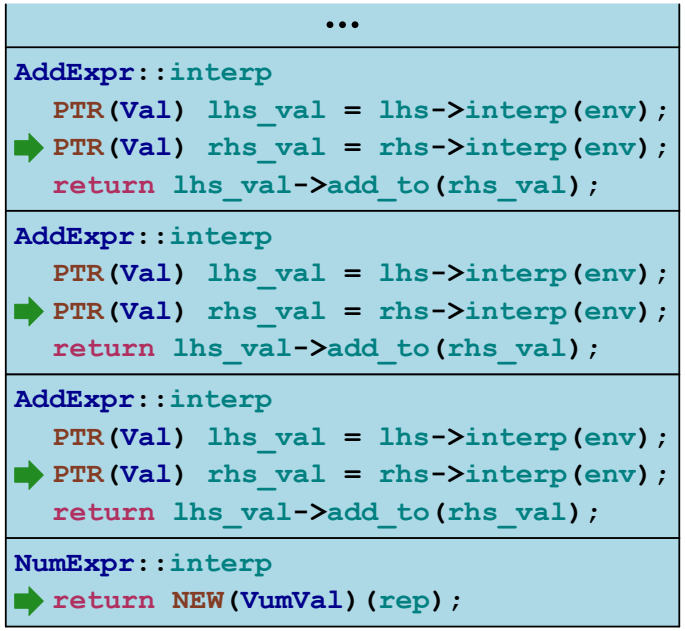



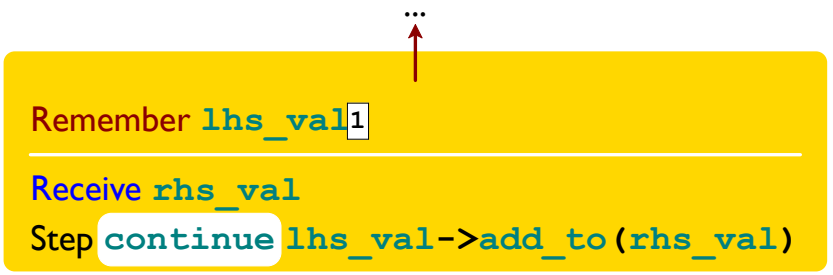

Remember `lhs_val`1

Receive `rhs_val`1

Step `continue lhs_val->add_to(rhs_val)`

# From Stack to Continuation



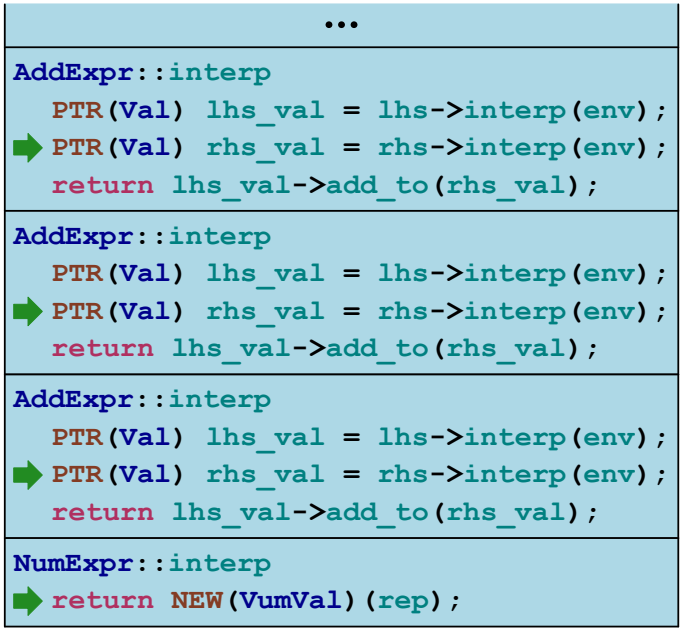

Remember `lhs_val`[1]

Receive `rhs_val`[1]

Step `continue`[2]

# From Stack to Continuation

...

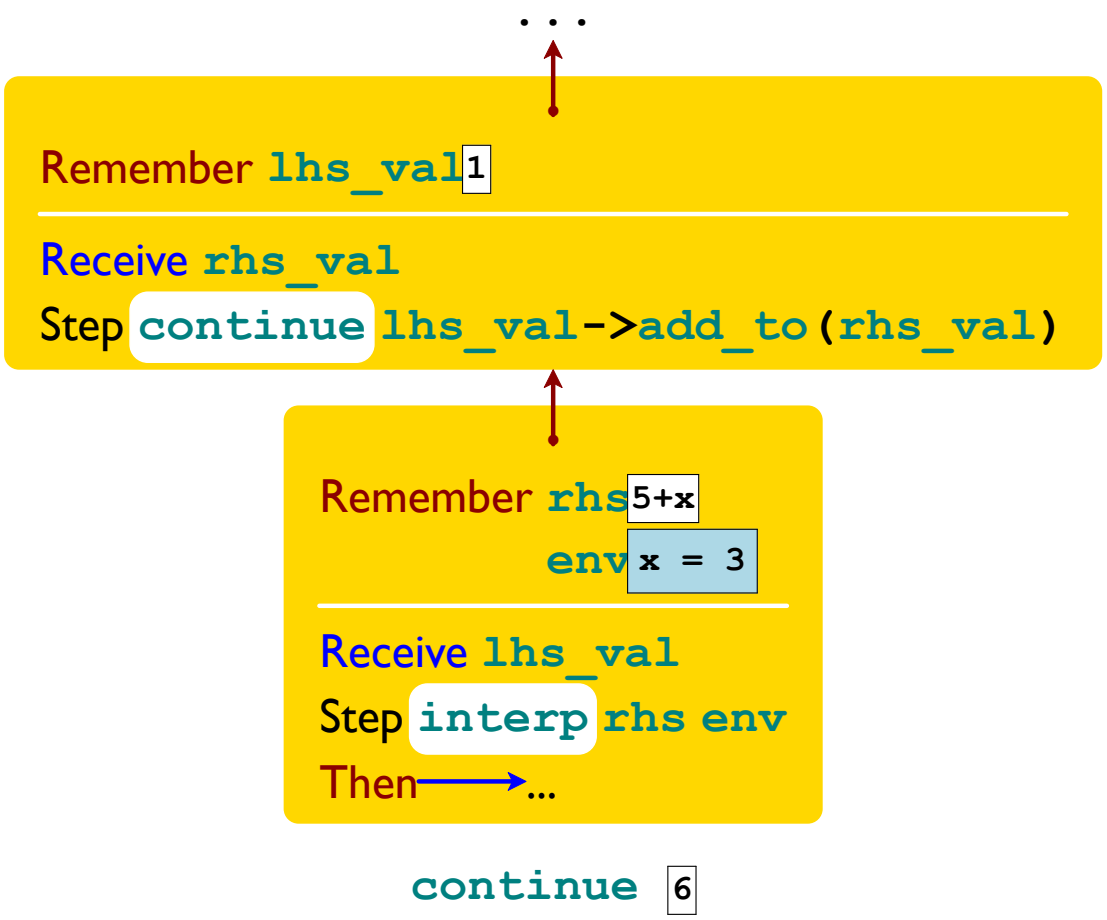



Remember `lhs_val`[1]

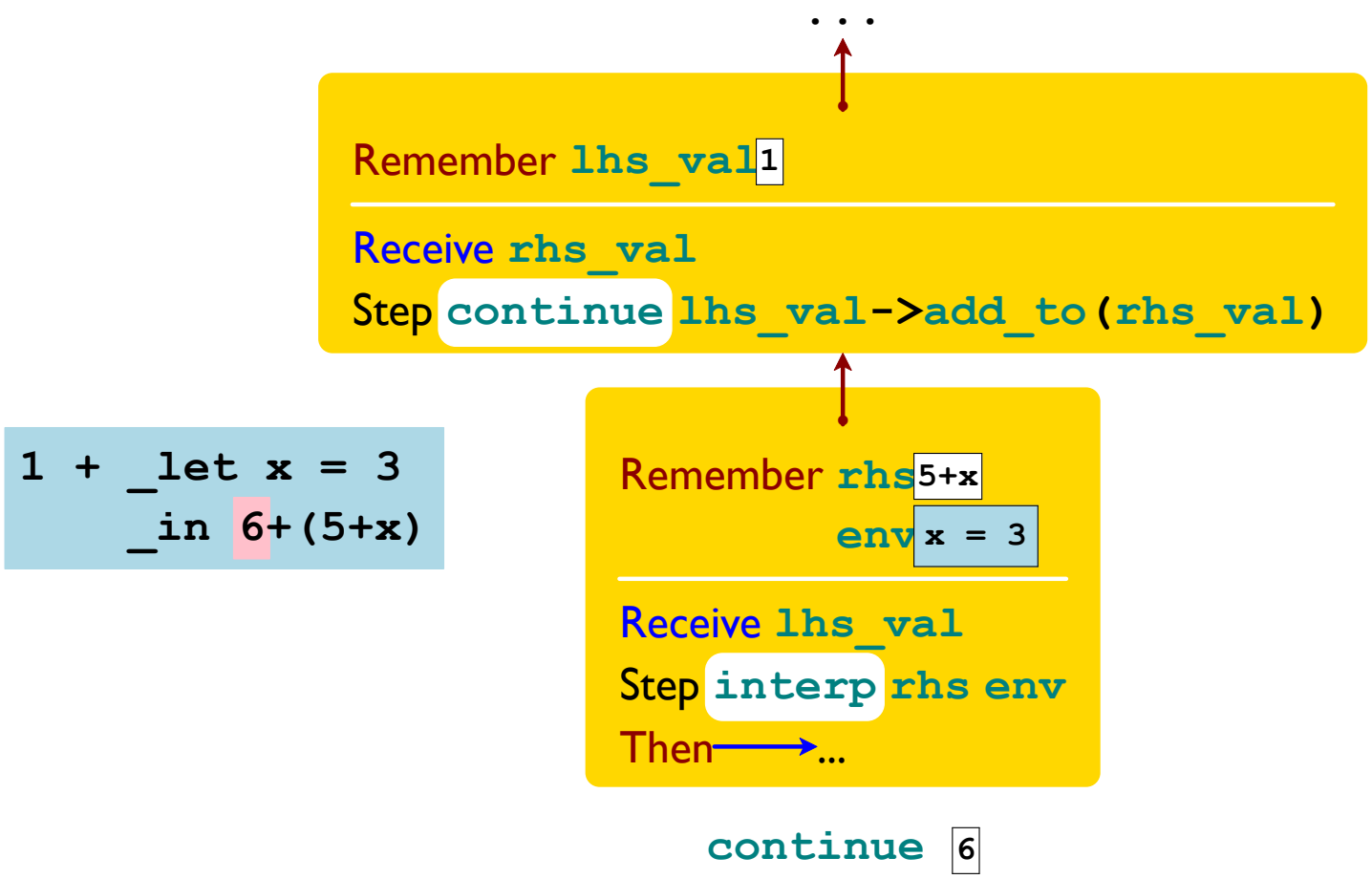
Receive `rhs_val`[2]

Step `continue lhs_val->add_to(rhs_val)`

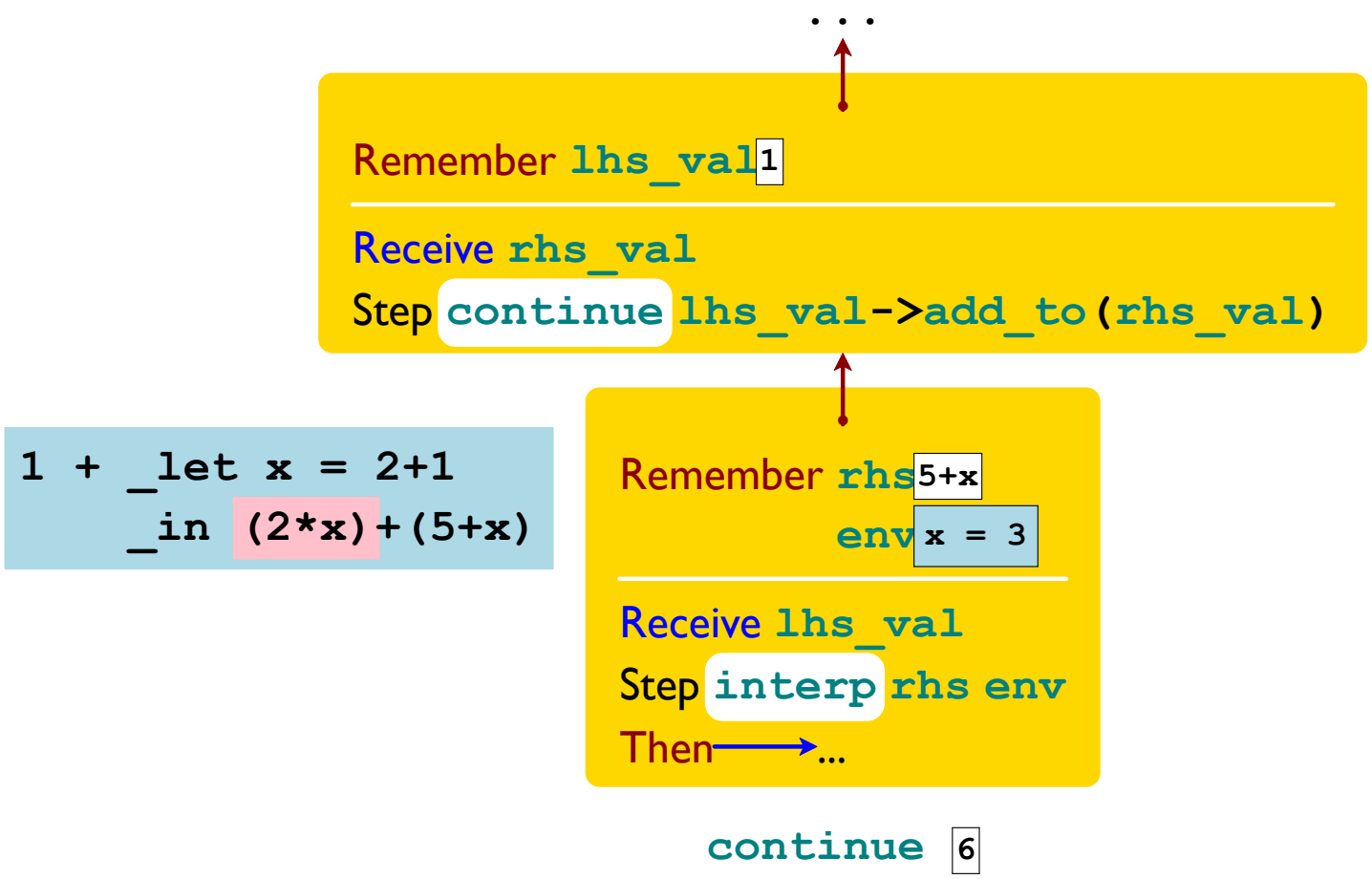
# Continuation and Steps



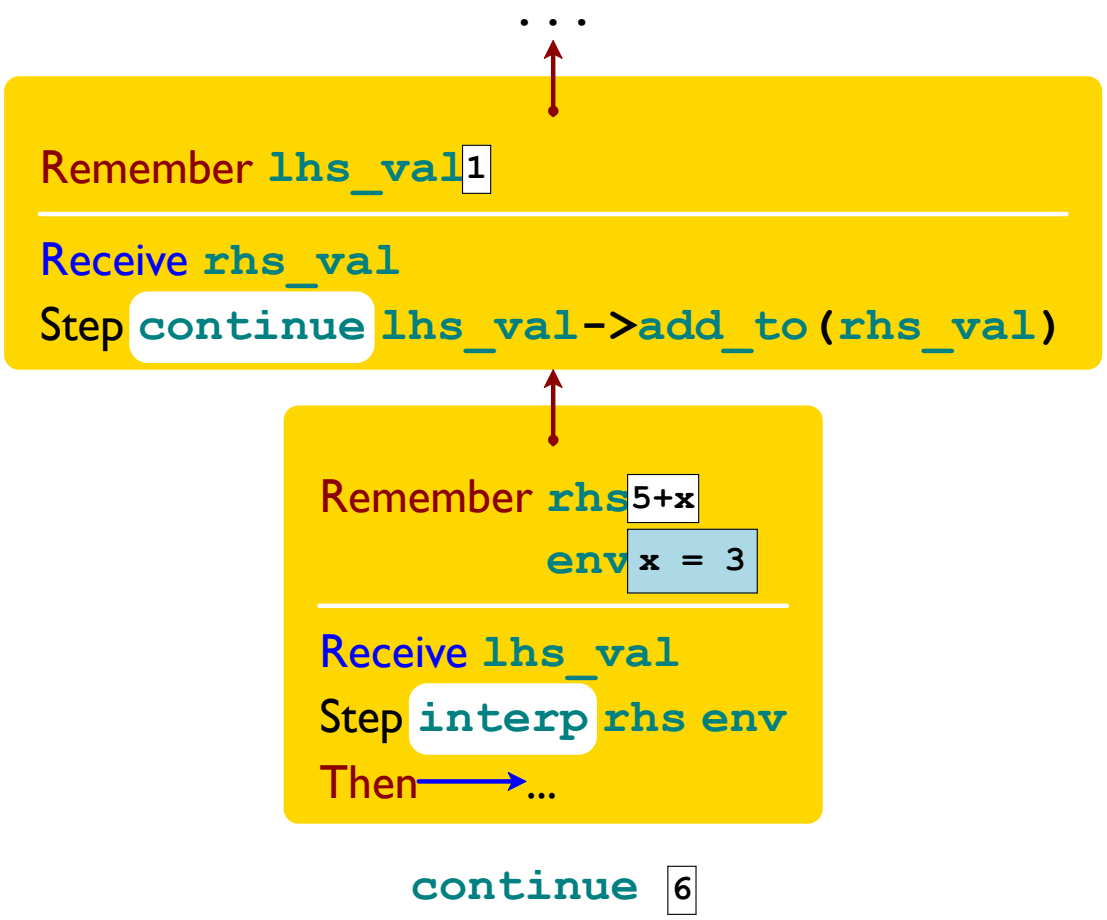
# Continuation and Steps



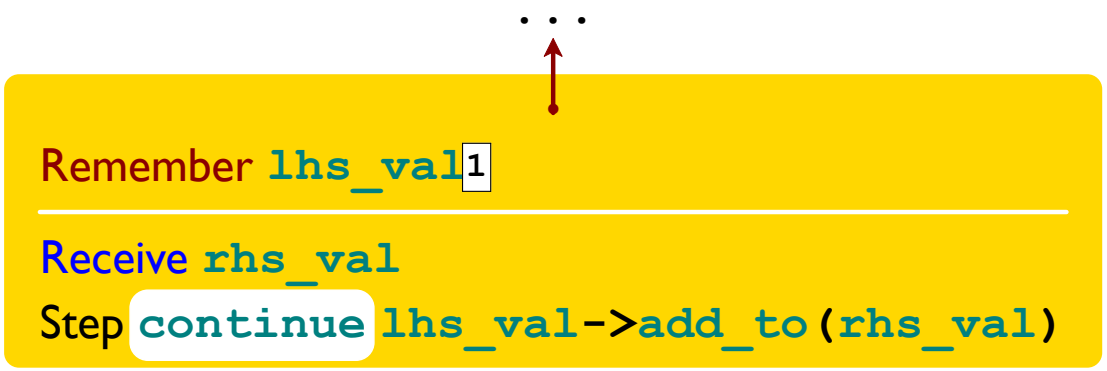
# Continuation and Steps



# Continuation and Steps



# Continuation and Steps



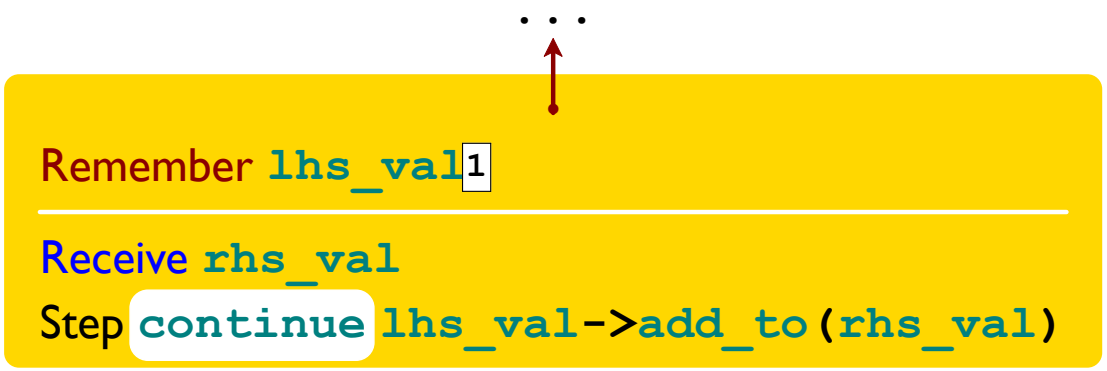

Remember `rhs` `5+x`
`env` `x = 3`

Receive `lhs_val` `6`

Step `interp rhs env`

Then ⟶ ...

# Continuation and Steps



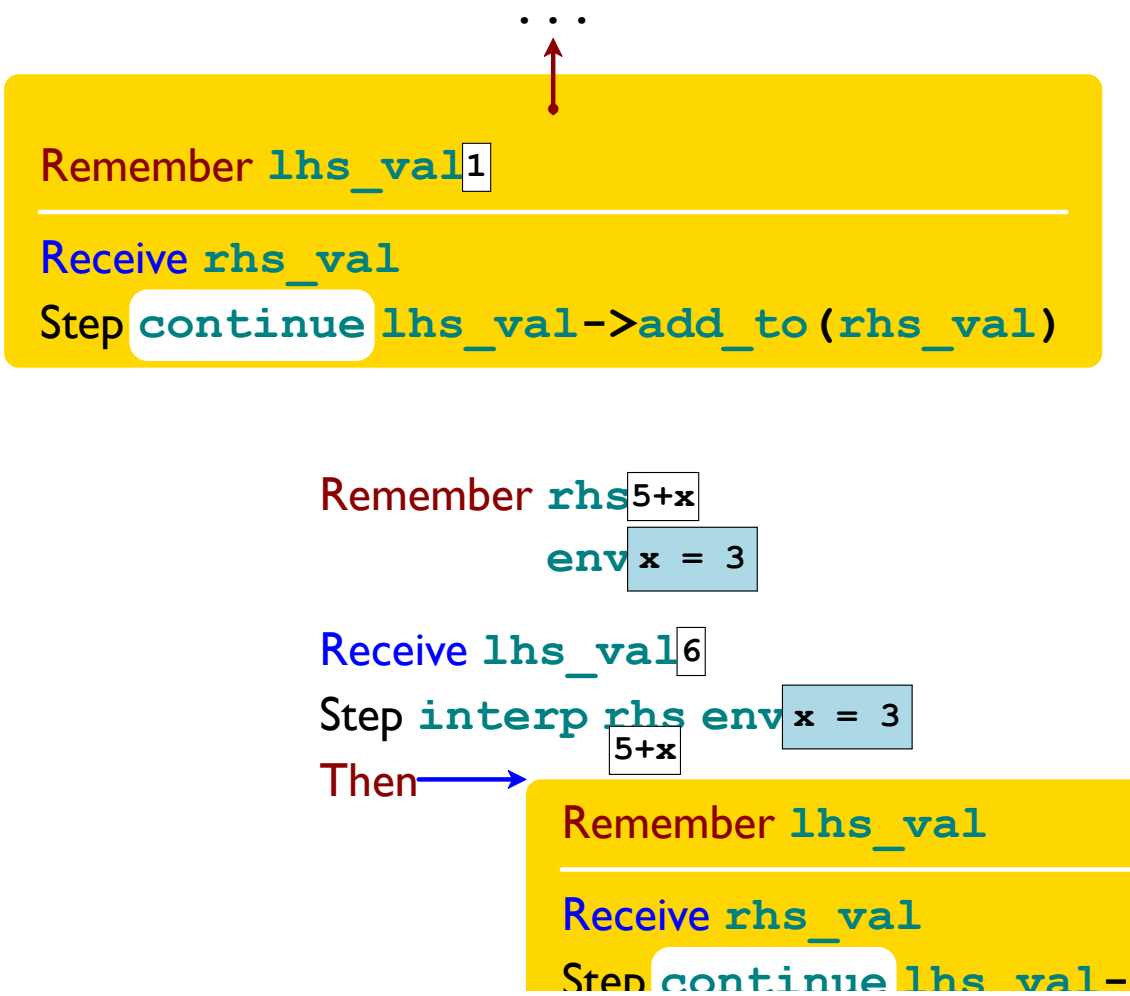

Remember `rhs` `5+x`
`env` `x = 3`

Receive `lhs_val` `6`

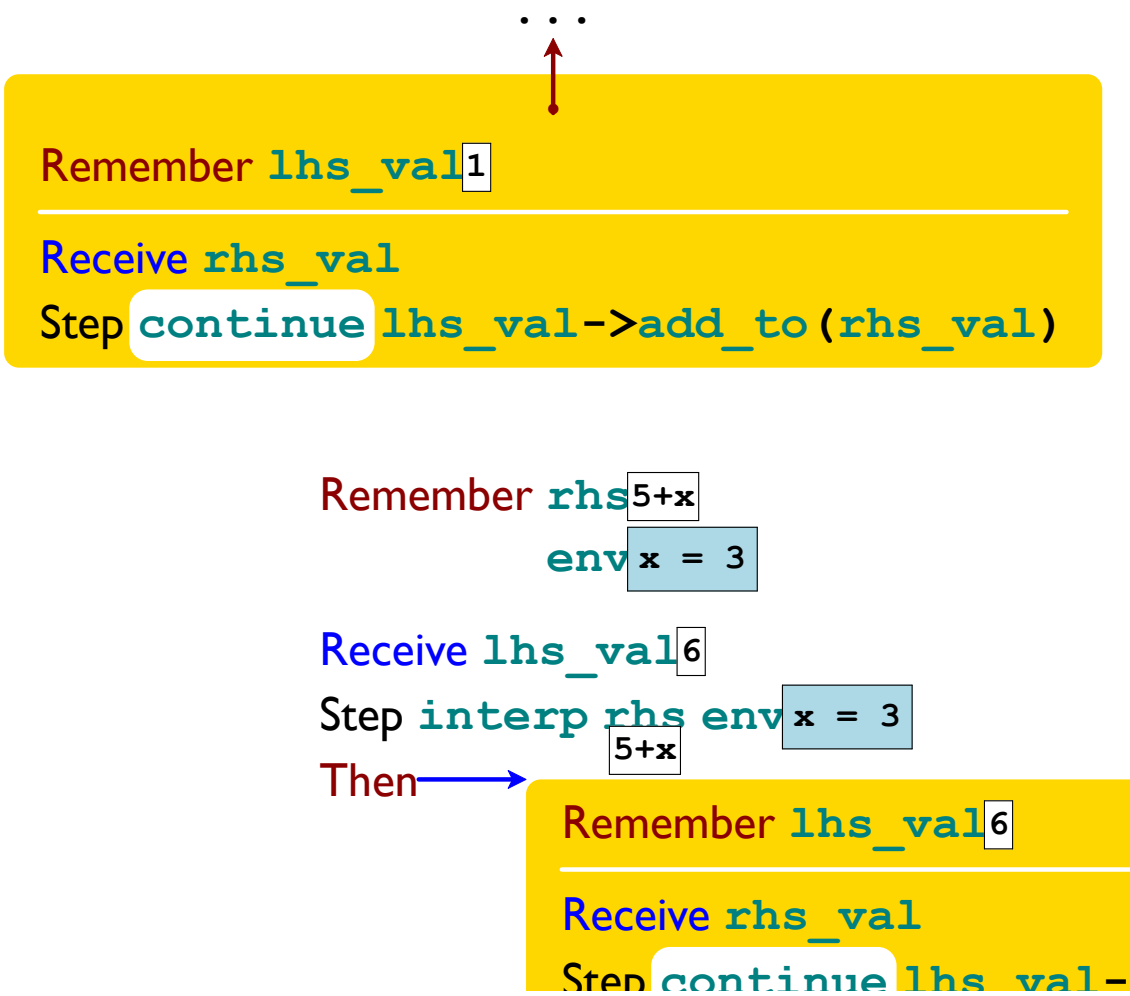
Step `interp rhs env` `x = 3`
`5+x`

Then → ...

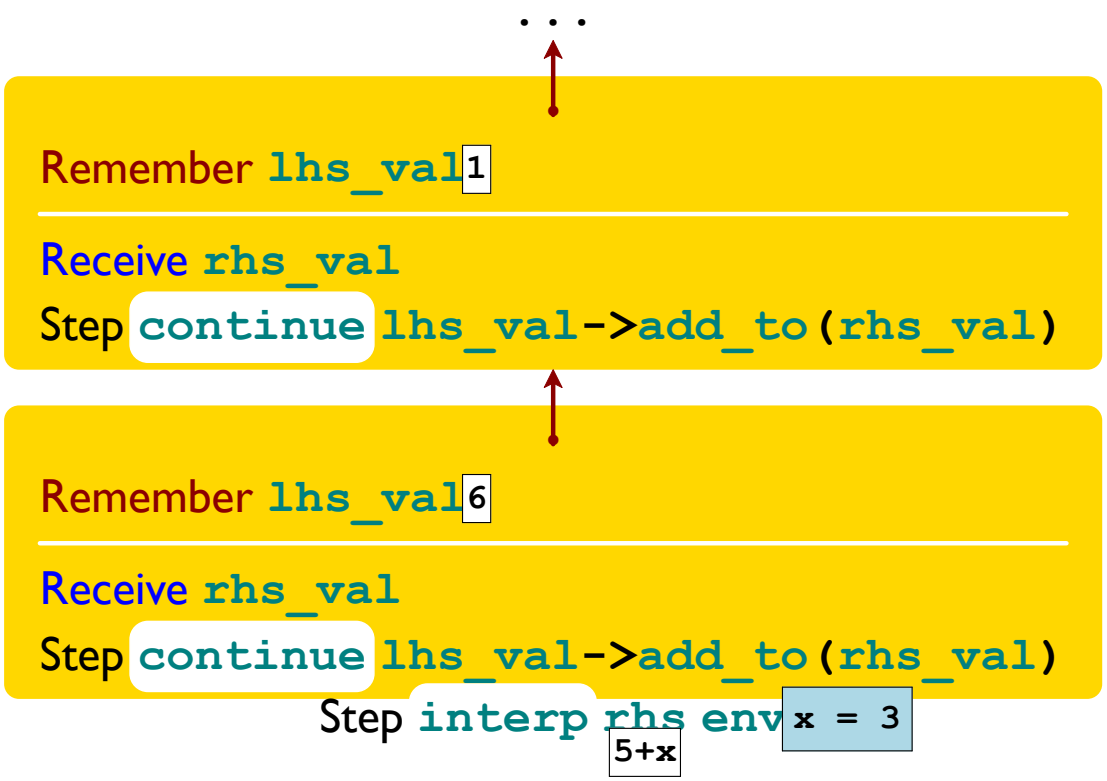
# Continuation and Steps



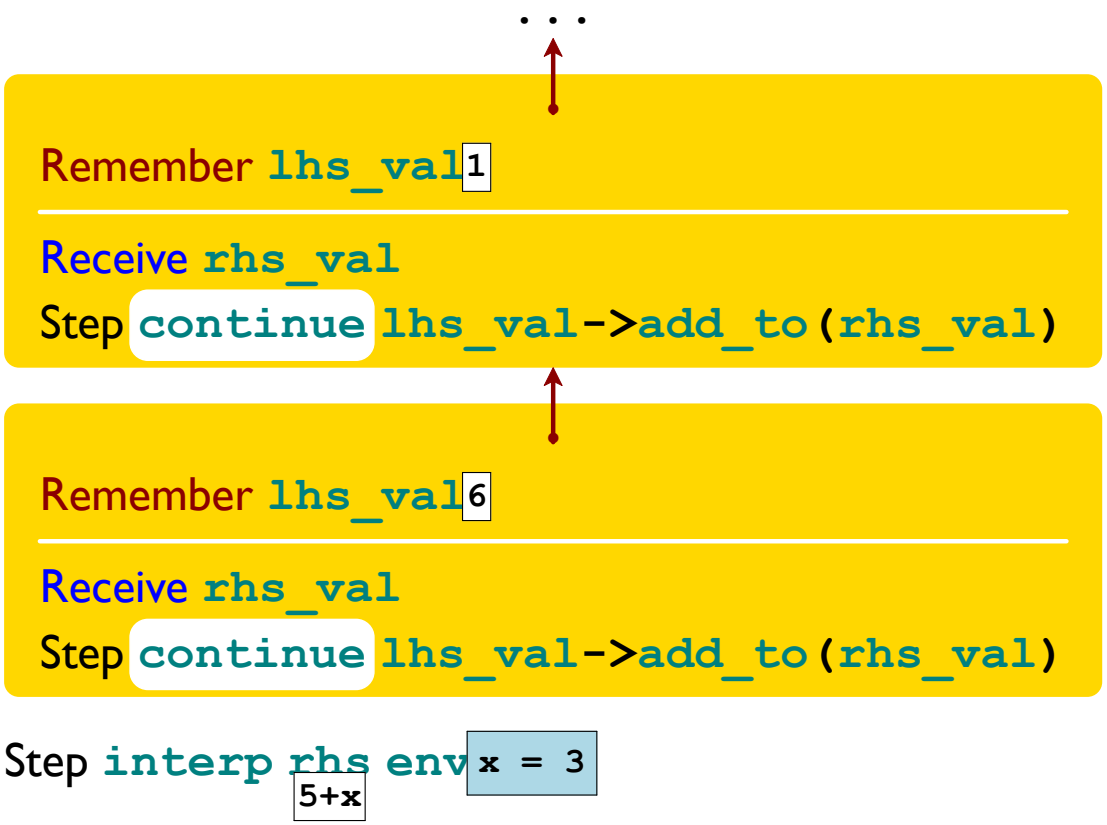
# Continuation and Steps



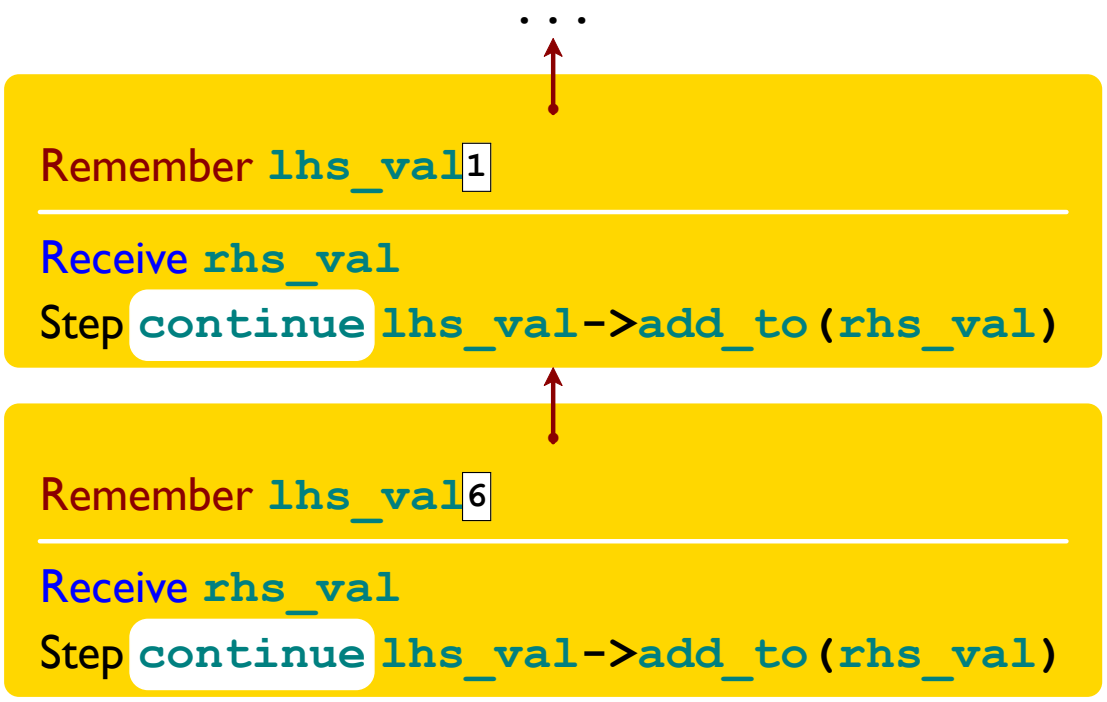
# Continuation and Steps



# Continuation and Steps



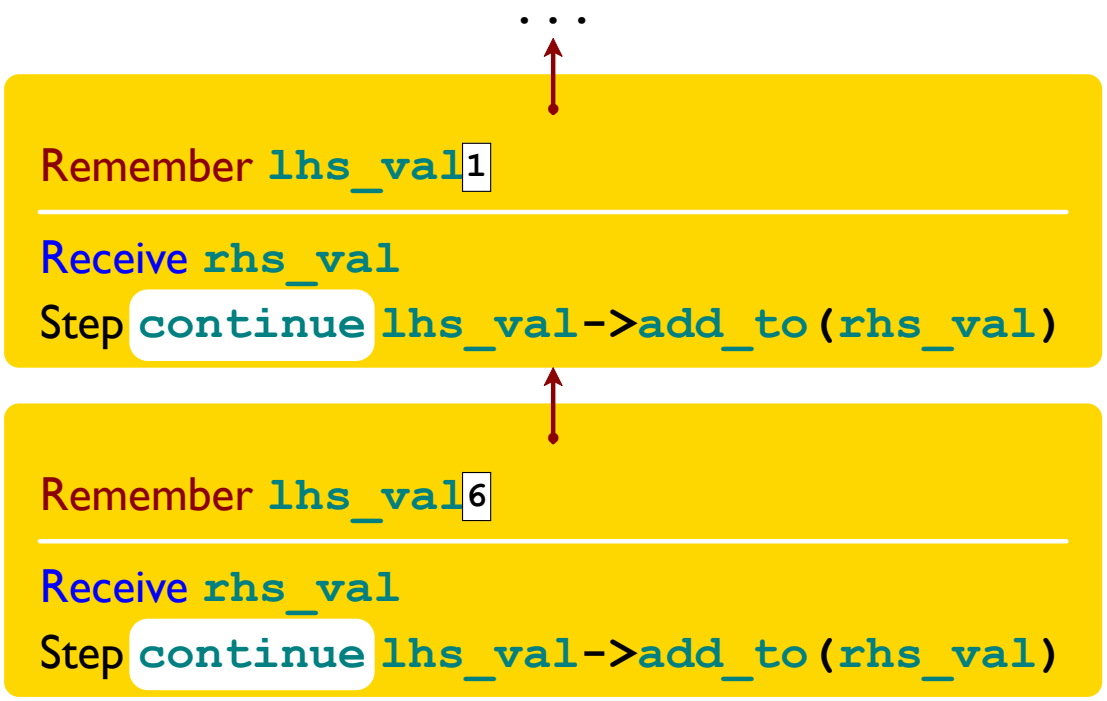
# Continuation and Steps




Step `interp` `rhs` `5+x` `env` `x = 3`

Stepping loop handles `interp` or `continue`

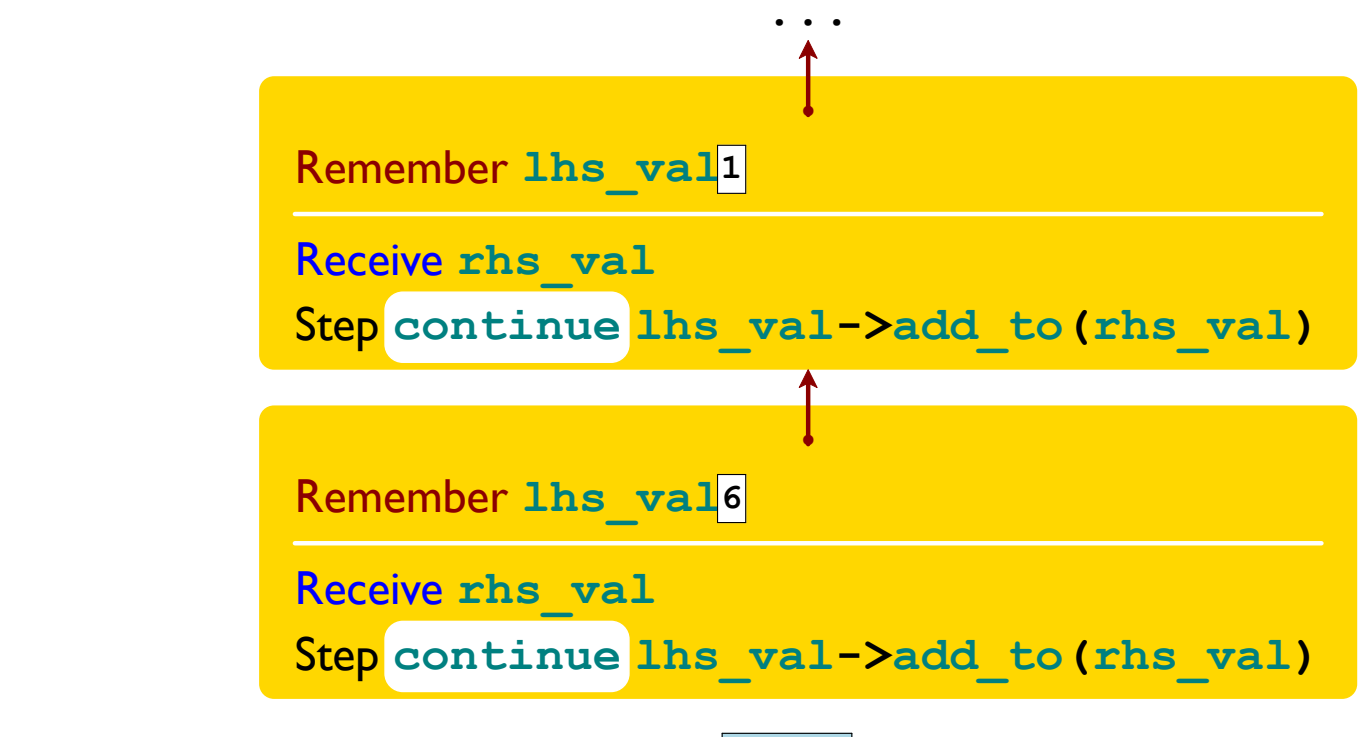
# Continuation and Steps



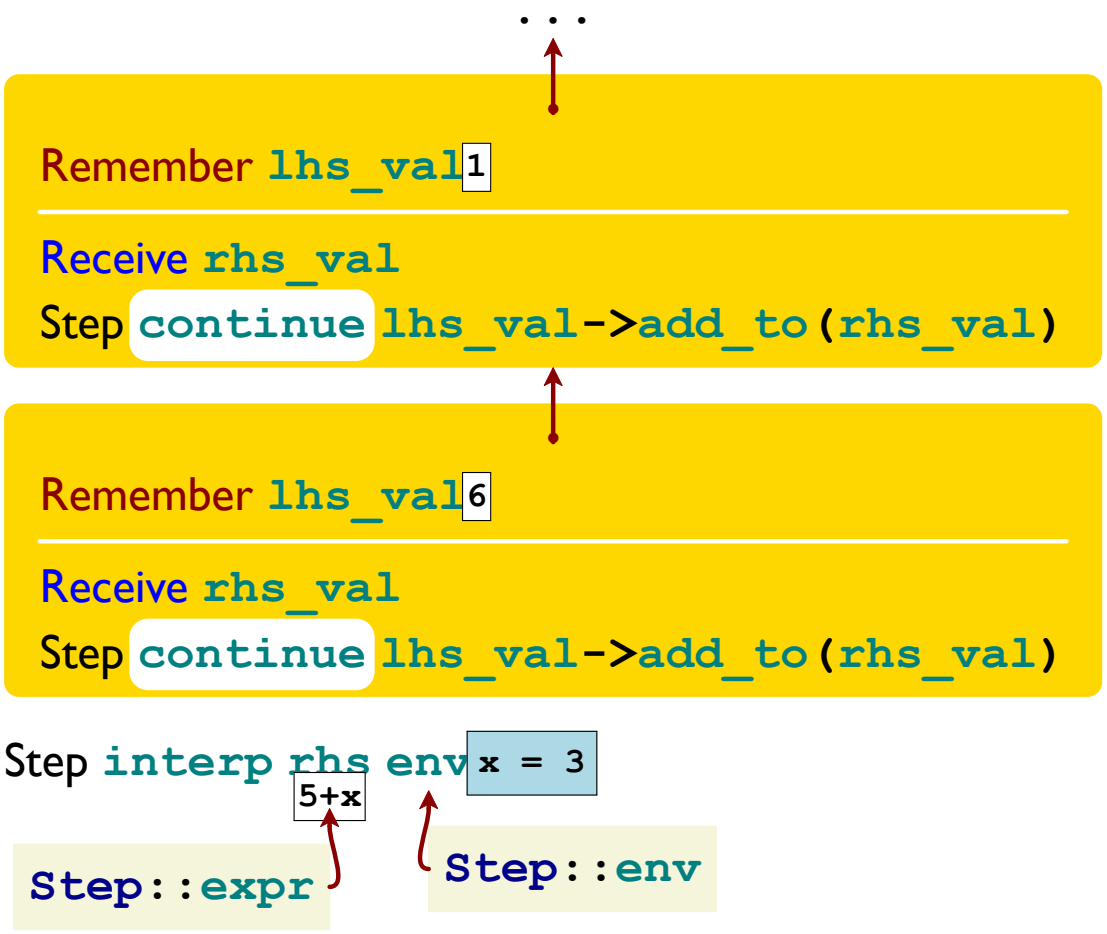

Step `interp` `rhs` `5+x` `env` `x = 3`

```
interp ⇒ Expr::step_interp
continue ⇒ Cont::step_continue
```

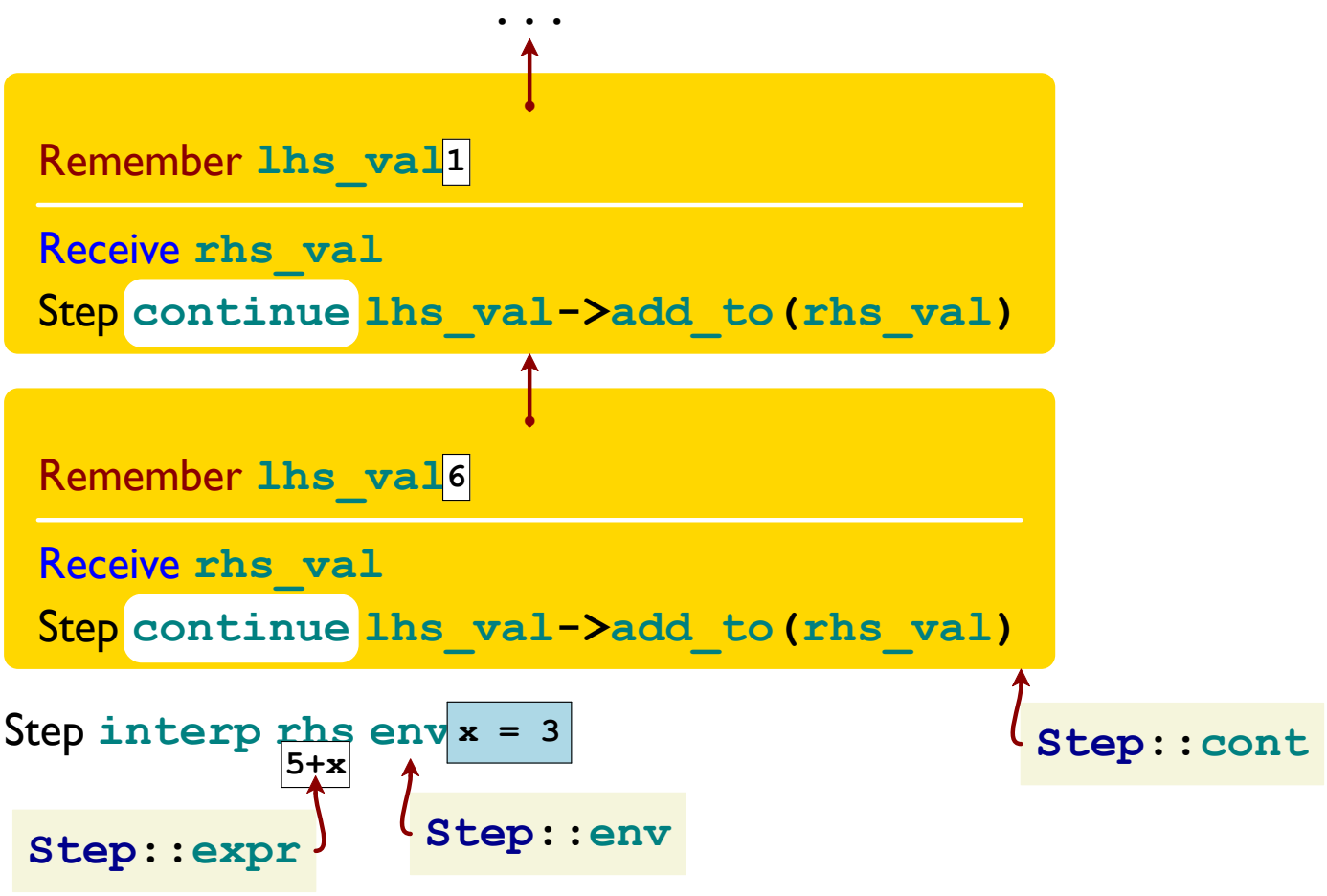
# Continuation and Steps



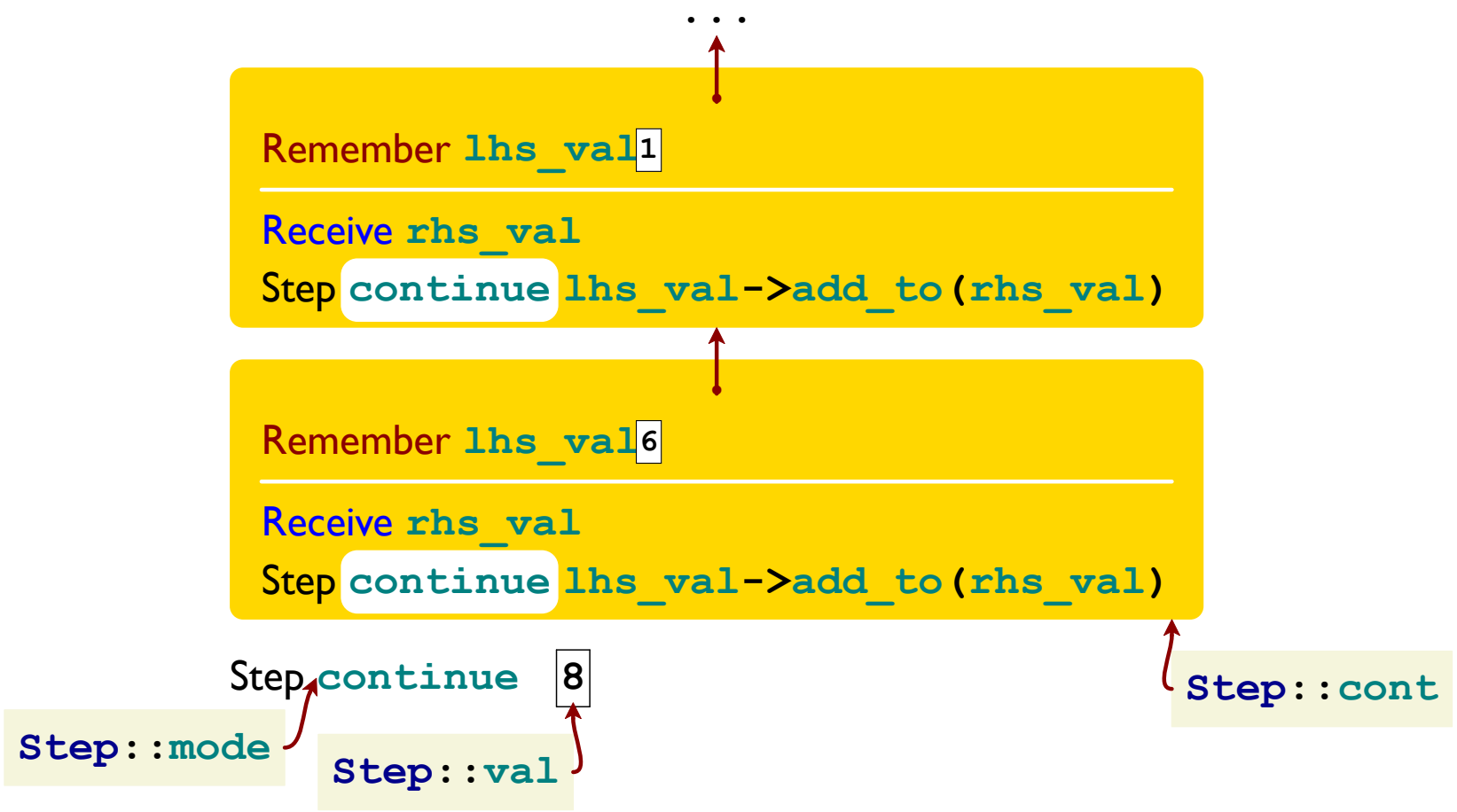
# Continuation and Steps



# Continuation and Steps



# Continuation and Steps



# Stepper State

step.hpp

```
class Step {
  typedef enum {
    interp_mode,
    continue_mode
  } mode_t;
  static mode_t mode;     /* chooses mode */

  static PTR(Expr) expr; /* for interp_mode */
  static PTR(Env)  env;  /* for interp_mode */

  static PTR(Val)  val;  /* for continue_mode */

  static PTR(Cont) cont; /* all modes */
};
```

# Stepper Loop

step.cpp

```
PTR(Val) interp_by_steps(PTR(Expr) e) {
  Step::mode = Step::interp_mode;
  Step::expr = e;
  Step::env = Env::empty;
  Step::val = nullptr;
  Step::cont = Cont::done;

  while (1) {
    if (Step::mode == Step::interp_mode)
      Step::expr->step_interp();
    else {
      if (Step::cont == Cont::done)
        return Step::val;
      else
        Step::cont->step_continue();
     }
  }
}
```

# Stepper Loop

step.cpp

```
PTR(Val) interp_by_steps(PTR(Expr) e) {
  Step::mode = Step::interp_mode;
  Step::expr = e;
  Step::env = Env::empty;
  Step::val = nullptr;
  Step::cont = Cont::done;

  while (1) {
    if (Step::mode == Step::interp_mode)
      Step::expr->step_interp();
    else {
      if (Step::cont == Cont::done)
        return Step::val;
      else
        Step::cont->step_continue();
    }
  }
}
```

**`step_interp`** and **`step_continue`** read and write the **`Step::`** variables

# `interp` to `step_interp`

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

Remember `rhs`
`env`

---

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

---

Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# interp to step_interp

```
PTR(Val) AddExpr::interp(PTR(Env) env)
  PTR(Val) lhs_val = lhs->interp(env);
  PTR(Val) rhs_val = rhs->interp(env);
  return lhs_val->add_to(rhs_val);
}
```

Step `interp` `lhs` `env`
Then

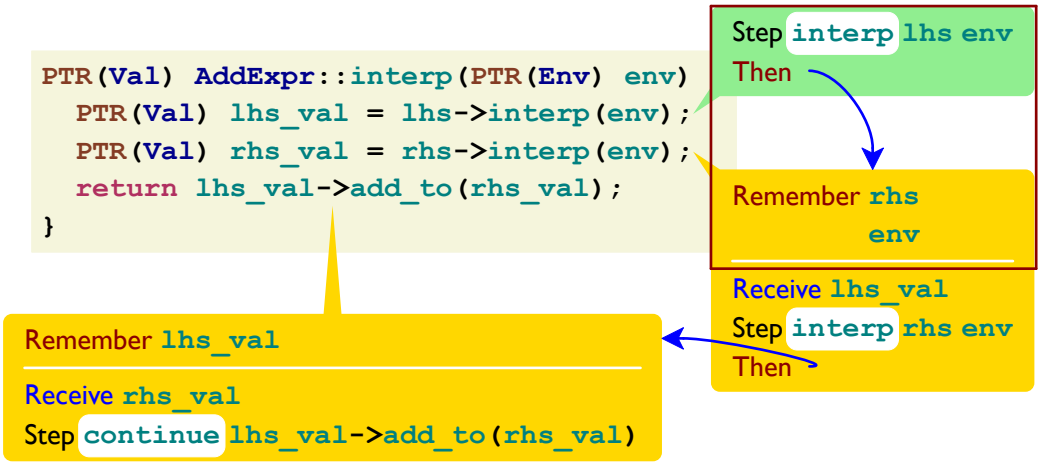
Remember `rhs`
`env`

Receive `lhs_val`
Step `interp` `rhs` `env`
Then

Remember `lhs_val`

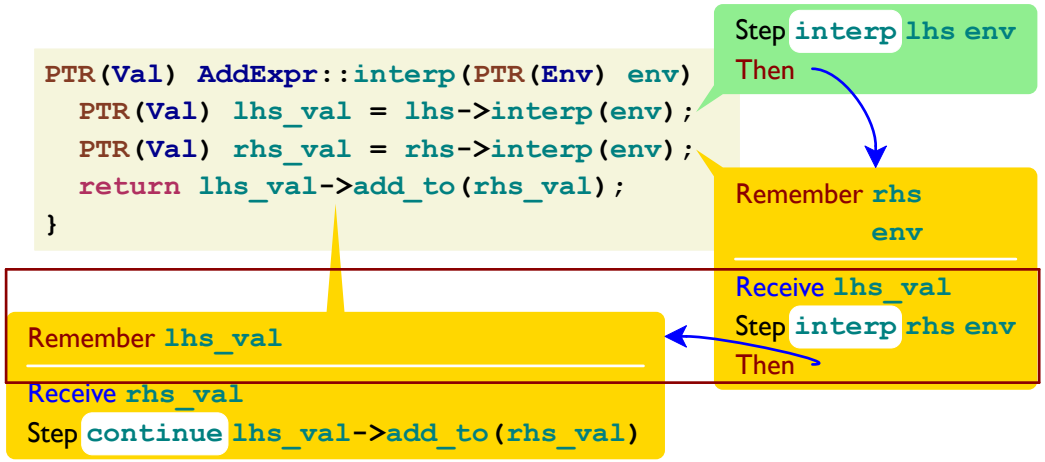
Receive `rhs_val`
Step `continue` `lhs_val->add_to(rhs_val)`

# interp to step_interp



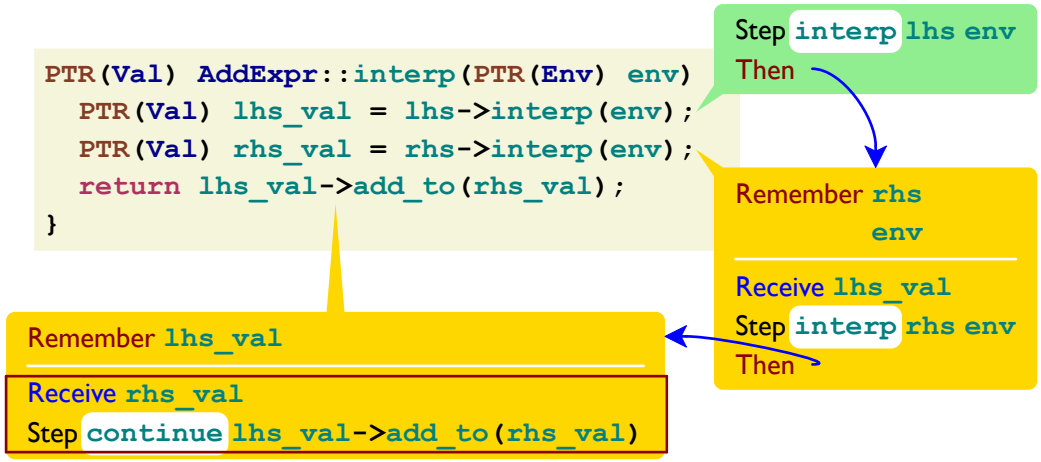

```
void AddExpr::step_interp() {
  Step::mode = Step::interp_mode;
  Step::expr = lhs;
  Step::env = Step::env; /* no-op, so could omit */
  Step::cont = NEW(RightThenAddCont)(rhs, Step::env, Step::cont);
}
```

# interp to step_interp




```
void RightThenAddCont::step_continue() {
  PTR(Val) lhs_val = Step::val;
  Step::mode = Step::interp_mode;
  Step::expr = rhs;
  Step::env = env;
  Step::cont = NEW(AddCont)(lhs_val, rest);
}
```

# interp to step_interp




```
void AddCont::step_continue() {
  PTR(Val) rhs_val = Step::val;
  Step::mode = Step::continue_mode;
  Step::val = lhs_val->add_to(rhs_val);
  Step::cont = rest;
}
```

# Expressions and Continuations

```
class Expr {
  virtual void step_interp() = 0;
};
```

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

  void step_interp();
};
```

…

```
class Cont {
  virtual void step_continue() = 0;
};
```

```
class RightThenAddCont : public Cont {
  PTR(Expr) rhs;
  PTR(Env)  env;
  PTR(Cont) rest;

  void step_continue();
};
```

```
class AddCont : public Cont {
  PTR(Val) lhs_val;
  PTR(Cont) rest;

  void step_continue();
};
```

…

# Expressions and Continuations

```
class Expr {
  virtual void step_interp() = 0;
};
```

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

  void step_interp();
};
```

…

**Exprs** have **step_interp**

**Conts** have **step_continue**

```
class Cont {
  virtual void step_continue() = 0;
};
```

```
class RightThenAddCont : public Cont {
  PTR(Expr) rhs;
  PTR(Env)  env;
  PTR(Cont) rest;

  void step_continue();
};
```

```
class AddCont : public Cont {
  PTR(Val) lhs_val;
  PTR(Cont) rest;

  void step_continue();
};
```

…

# Expressions and Continuations

```
class Expr {
  virtual void step_interp() = 0;
};
```

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

  void step_interp();
};
```

…

Each **Expr** subclass needs some number of **Cont** subclasses

```
class Cont {
  virtual void step_continue() = 0;
};
```

```
class RightThenAddCont : public Cont {
  PTR(Expr) rhs;
  PTR(Env)  env;
  PTR(Cont) rest;

  void step_continue();
};
```

```
class AddCont : public Cont {
  PTR(Val) lhs_val;
  PTR(Cont) rest;

  void step_continue();
};
```

…

# Expressions and Continuations

```
class Expr {
  virtual void step_interp() = 0;
};
```

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

  void step_interp();
};
```

…

`step_intep` can set the mode to `interp_mode` or `continue_mode`

```
class Cont {
  virtual void step_continue() = 0;
};
```

```
class RightThenAddCont : public Cont {
  PTR(Expr) rhs;
  PTR(Env)  env;
  PTR(Cont) rest;

  void step_continue();
};
```

```
class AddCont : public Cont {
  PTR(Val) lhs_val;
  PTR(Cont) rest;

  void step_continue();
};
```

…

# Expressions and Continuations

```
class Expr {
  virtual void step_interp() = 0;
};
```

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

  void step_interp();
};
```

…

`step_continue` can set the mode to `interp_mode` or `continue_mode`

```
class Cont {
  virtual void step_continue() = 0;
};
```

```
class RightThenAddCont : public Cont {
  PTR(Expr) rhs;
  PTR(Env)  env;
  PTR(Cont) rest;

  void step_continue();
};
```

```
class AddCont : public Cont {
  PTR(Val) lhs_val;
  PTR(Cont) rest;

  void step_continue();
};
```

…

# More Function Rules

`step_interp` should never call

- `step_interp`
- `step_continue`
- `interp_by_steps`
- `interp`
- `optimize`
- `subst`

# More Function Rules

`step_continue` should never call

- `step_interp`
- `step_continue`
- `interp_by_steps`
- `interp`
- `optimize`
- `subst`

# More Function Rules

Only `interp_by_steps` should call

- `step_interp`
- `step_continue`

# Converting `_if`

```
PTR(Val) IfExpr::interp(PTR(Env) env) {
  PTR(Val) test_val = test_part->interp(env);
  if (test_val->is_true())
    return then_part->interp(env);
  else
    return else_part->interp(env);
}
```

Step `interp test_part env`

Remember `then_part` and `else_part` and `env`
Receive `test_val`
Either
- Step `interp then_part env`
- Step `interp else_part env`

# Converting `_if`

```
PTR(Val) IfExpr::interp(PTR(Env) env) {
  PTR(Val) test_val = test_part->interp(env);
  if (test_val->is_true())
    return then_part->interp(env);
  else
    return else_part->interp(env);
}
```

Step `interp test_part env`

Remember `then_part` and `else_part` and `env`
Receive `test_val`
Either
- Step `interp then_part env`
- Step `interp else_part env`

`IfBranchCont`

# Converting `_if`

```
void IfExpr::step_interp() {
  Step::mode = Step::interp_mode;
  Step::expr = test_part;
  Step::env = Step::env;
  Step::cont = NEW(IfBranchCont)(then_part, else_part, Step::env, Step::cont);
}
```

```
class IfBranchCont: : public Cont {
  PTR(Expr) then_part;
  PTR(Expr) else_part;
  PTR(Env) env;
  PTR(Cont) rest;
};
```

```
void IfBranchCont::step_continue()
{
  PTR(Val) test_val = Step::val;
  Step::mode = interp_mode;
  if (test_val->is_true())
    Step::expr = then_part;
  else
    Step::expr = else_part;
  Step::env = env;
  Step::cont = rest;
}
```

# Converting Number Expressions

```
PTR(Val) NumExpr::interp(PTR(Env) env) {
  return NEW(NumVal)(rep);
}
```

```
continue NEW(NumVal)(rep)
```

```
void NumExpr::step_interp() {
  Step::mode = continue_mode;
  Step::val = NEW(NumVal)(rep);
  Step::cont = Step::cont; /* no-op */
}
```

# Converting `_let`

```
PTR(Val) LetExpr::interp(PTR(Env) env)
{
  PTR(Val) rhs_val = rhs->interp(env);
  return body->interp(NEW(ExtendedEnv)(var, rhs_val, env));
}
```

Step `interp rhs env`

Remember `body` and `var` and `env`
Receive `rhs_val`
Step `interp body NEW(ExtendedEnv)(var, rhs_val, env)`

# Converting `_let`

```
PTR(Val) LetExpr::interp(PTR(Env) env)
{
  PTR(Val) rhs_val = rhs->interp(env);
  return body->interp(NEW(ExtendedEnv)(var, rhs_val, env));
}
```

Step `interp rhs env`

Remember `body` and `var` and `env` — `LetBodyCont`
Receive `rhs_val`
Step `interp body NEW(ExtendedEnv)(var, rhs_val, env)`

# Converting `_let`

```
void LetExpr::step_interp() {
  Step::mode = interp_mode;
  Step::expr = rhs;
  Step::env = Step::env;
  Step::cont = NEW(LetBodyCont)(var, body, Step::env, Step::cont);
}
```

```
class LetBodyCont : public Cont {
  std::string var;
  PTR(Expr) body;
  PTR(Env) env;
  PTR(Cont) rest;
};
```

```
void LetBodyCont::step_continue()
{
  Step::mode = interp_mode;
  Step::expr = body;
  Step::env = NEW(ExtendedEnv)(var,
                               Step::val,
                               env);
  Step::cont = rest;
}
```

# Converting Function Calls

```
PTR(Val) CallExpr::interp(PTR(Env) env) {
  PTR(Val) to_be_called_val = to_be_called->interp(env);
  PTR(Val) actual_arg_val = actual_arg->interp(env);
  return to_be_called_val->call(actual_arg_val);
}
```

```
PTR(Val) FunVal::call(PTR(Val) actual_arg) {
  return body->interp(NEW(ExtendedEnv)(formal_arg, actual_arg, env));
}
```

Step `interp to_be_called env`

Remember `actual_arg` and `env`
Receive `to_be_called_val`
Step `interp actual_arg env`

Remember `to_be_called_val`
Receive `actual_arg_val`
Step `interp body NEW(ExtendedEnv)(formal_arg, actual_arg, env)`

# Converting Function Calls

```
PTR(Val) CallExpr::interp(PTR(Env) env) {
  PTR(Val) to_be_called_val = to_be_called->interp(env);
  PTR(Val) actual_arg_val = actual_arg->interp(env);
  return to_be_called_val->call(actual_arg_val);
}
```

```
PTR(Val) FunVal::call(PTR(Val) actual_arg) {
  return body->interp(NEW(ExtendedEnv)(formal_arg, actual_arg, env));
}
```

Step `interp to_be_called env`

Remember `actual_arg` and `env` — **`ArgThenCallCont`**
Receive `to_be_called_val`
Step `interp actual_arg env`

Remember `to_be_called_val` — **`CallCont`**
Receive `actual_arg_val`
Step `interp body NEW(ExtendedEnv)(formal_arg, actual_arg, env)`

# Converting Function Calls

```
PTR(Val) CallExpr::interp(PTR(Env) env) {
  PTR(Val) to_be_called_val = to_be_called->interp(env);
  PTR(Val) actual_arg_val = actual_arg->interp(env);
  return to_be_called_val->call(actual_arg_val);
}
```

```
PTR(Val) FunVal::call(PTR(Val) actual_arg) {
  return body->interp(NEW(ExtendedEnv)(formal_arg, actual_arg, env));
}
```

Step `interp to_be_called env`

Remember `actual_arg` and `env`
Receive `to_be_called_val`
Step `interp actual_arg env`

**`ArgThenCallCont`**

Remember `to_be_called_val`
Receive `actual_arg_val`
Step `interp body NEW(ExtendedEnv)(formal_arg, actual_arg, env)`

**`CallCont`**

Step goes in **`FunVal::step_call`**

# Converting Function Calls

```
void CallExpr::step_interp() {
  Step::mode = interp_mode;
  Step::expr = to_be_called;
  Step::cont = NEW(ArgThenCallCont)(actual_arg, Step::env, Step::cont);
}
```

```
class ArgThenCallCont : public Cont {
  PTR(Expr) actual_arg;
  PTR(Env)  env;
  PTR(Cont) rest;
};
```

```
void ArgThenCallCont::step_continue() {
  Step::mode = interp_mode;
  Step::expr = actual_arg;
  Step::env = env;
  Step::cont = NEW(CallCont)(Step::val, rest);
}
```

```
class CallCont : public Cont {
  PTR(Val)  to_be_called_val;
  PTR(Cont) rest;
};
```

```
void CallCont::step_continue() {
  to_be_called_val->call_step(Step::val, rest);
}
```

```
void FunVal::call_step(PTR(Val) actual_arg_val, PTR(Cont) rest) {
  Step::mode = interp_mode;
  Step::expr = body;
  Step::env = NEW(ExtendedEnv)(formal_arg_val, actual_arg, env);
  Step::cont = rest;
}
```

# Converting Function Calls

```
void CallExpr::step_interp() {
  Step::mode = interp_mode;
  Step::expr = to_be_called;
  Step::cont = NEW(ArgThenCallCont)(actual_arg, Step::env, Step::cont);
}
```

```
class ArgThenCallCont : public Cont {
  PTR(Expr) actual_arg;
  PTR(Env)  env;
  PTR(Cont) rest;
};
```

```
void ArgThenCallCont::step_continue() {
  Step::mode = interp_mode;
  Step::expr = actual_arg;
  Step::env = env;
  Step::cont = NEW(CallCont)(Step::val, rest);
}
```

```
class CallCont : public Cont {
  PTR(Val)  to_be_called_val;
  PTR(Cont) rest;
};
```

```
void CallCont::step_continue() {
  to_be_called_val->call_step(Step::val, rest);
}
```

A function call *does not* extend the continuation

```
void FunVal::call_step(PTR(Val) actual_arg_val, PTR(Cont) rest) {
  Step::mode = interp_mode;
  Step::expr = body;
  Step::env = NEW(ExtendedEnv)(formal_arg_val, actual_arg, env);
  Step::cont = rest;
}
```

# Expressions and Associated Continuations

```
NumExpr  ⇒ continue
BoolExpr ⇒ continue
VarExpr  ⇒ continue
FunExpr  ⇒ continue
AddExpr  ⇒ RightThenAddCont
           AddCont
MultExpr ⇒ RightThenMultCont
           MultCont
CompExpr ⇒ RightThenCompCont
           CompCont
CallExpr ⇒ ArgThenCallCont
           CallCont
IfExpr   ⇒ IfBranchCont
LetExpr  ⇒ LetBodyCont
```