Evaluation and Substitution

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

Evaluation and Substitution

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡    _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in 1 + x2 + ... x100
```

Evaluation and Substitution

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡       _let x2 = 2
    _in _let x3 = 3
    ...
    _in _let x100 = 100
    _in 1 + x2 + ... x100
```

each **+** is a `new AddExpr`

Evaluation and Substitution

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡    _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in 1 + x2 + ... x100
```

so, about 100 `new AddExprs`

Evaluation and Substitution

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡     _let x2 = 2
    _in _let x3 = 3
    ...
    _in _let x100 = 100
    _in 1 + x2 + ... x100
```

```
➡     _let x3 = 3
    ...
    _in _let x100 = 100
    _in 1 + 2 + ... x100
```

Evaluation and Substitution

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡     _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in 1 + x2 + ... x100
```

```
➡     _let x3 = 3
...
_in _let x100 = 100
_in 1 + 2 + ... x100
```

Substituting 100 times means 100 big copies

Separate Variable Dictionary

Idea: a dictionary on the side

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

Separate Variable Dictionary

Idea: a dictionary on the side

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
    _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

x1 = 1

Separate Variable Dictionary

Idea: a dictionary on the side

```
x2 = 2
x1 = 1
```

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡     _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡     _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

Separate Variable Dictionary

Idea: a dictionary on the side

```
x2 = 2
x1 = 1
```

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

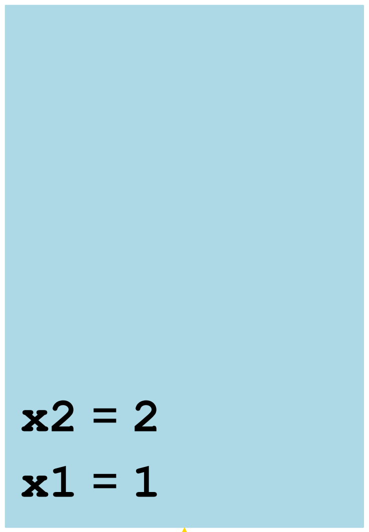
```
➡       _let x2 = 2
    _in _let x3 = 3
    ...
    _in _let x100 = 100
    _in x1 + x2 + ... x100
```

```
➡       _let x3 = 3
    ...
    _in _let x100 = 100
    _in x1 + x2 + ... x100
```

no copy needed in each step

Separate Variable Dictionary

Idea: a dictionary on the side



adding to dictionary can be fast

```
_let x1 = 1
_in _let x2 = 2
_in _let x3 = 3
...
_in _let x100 = 100
_in x1 + x2 + ... x100
```

```
➡     _let x2 = 2
   _in _let x3 = 3
   ...
   _in _let x100 = 100
   _in x1 + x2 + ... x100
```

```
➡     _let x3 = 3
   ...
   _in _let x100 = 100
   _in x1 + x2 + ... x100
```

no copy needed in each step

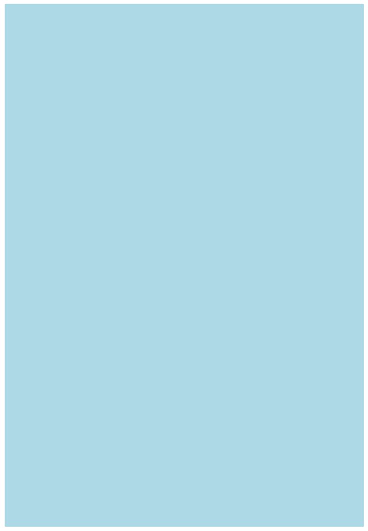
Separate Variable Dictionary

Idea: a dictionary on the side

```
x100 = 100
…
x2 = 2
x1 = 1
```

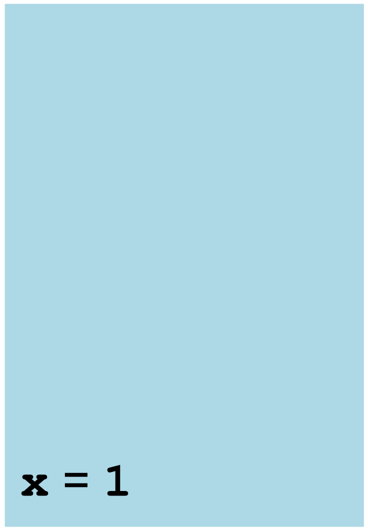
```
. . .
─────────────────────────────
➡ x1 + x2 + ... x100
```

Just One Dictionary?



```
_let x = 1
_in _let x = 2
_in x
```

Just One Dictionary?



```
_let x = 1
_in _let x = 2
_in x
```

```
➡    _let x = 2
_in x
```

Just One Dictionary?

```
x = 2
x = 1
```

```
_let x = 1
_in _let x = 2
_in x
```

```
➡     _let x = 2
  _in x
```

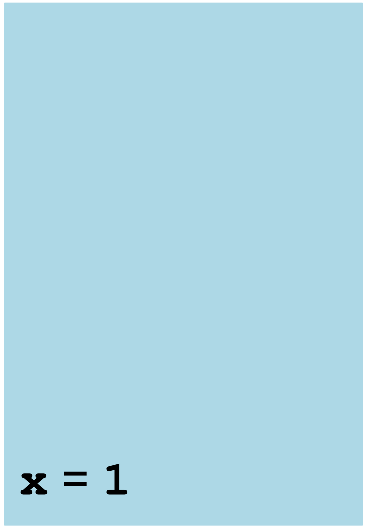
```
➡     x
```

Seems ok if we always use the newest value

Just One Dictionary?

```
_let x = 1
_in (_let x = 2 _in x) + x
```

Just One Dictionary?



```
_let x = 1
_in (_let x = 2 _in x) + x
```

```
➡      (_let x = 2 _in x) + x
```

Just One Dictionary?

```
x = 2
x = 1
```

```
    _let x = 1
    _in (_let x = 2 _in x) + x
```

```
➡       (_let x = 2 _in x) + x
```

```
➡       (               x) + x
```

Just One Dictionary?

```
x = 2
x = 1
```

```
      _let x = 1
      _in (_let x = 2 _in x) + x
➡           (_let x = 2 _in x) + x
➡           (               x) + x
➡           (               2) + x
```

Just One Dictionary?

```
x = 2
x = 1
```

```
    _let x = 1
    _in (_let x = 2 _in x) + x
---------------------------------
➡       (_let x = 2 _in x) + x
---------------------------------
➡       (               x) + x
---------------------------------
➡       (               2) + x
---------------------------------
➡       (               2) + 2
```

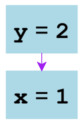
Not consistent with substitution, so it’s wrong

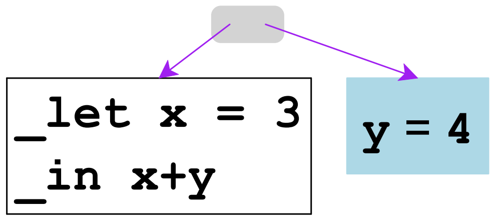
A single dictionary is wrong because it applies *everywhere*, but substitution applies to a specific expression

Closures

To accurately imitate substitution, pair an expression and a dictionary

The pair is called a ***closure***

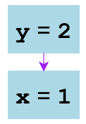
The dictionary is called an ***environment*** 

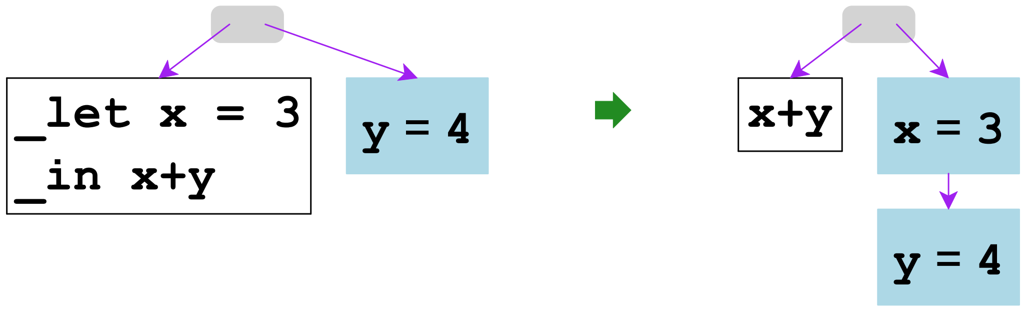


Closures

To accurately imitate substitution, pair an expression and a dictionary

The pair is called a ***closure***

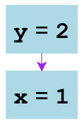
The dictionary is called an ***environment*** 

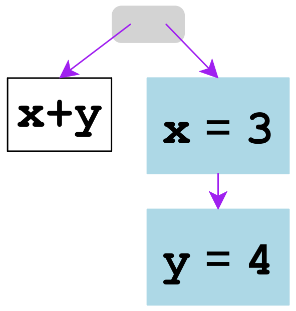


Closures

To accurately imitate substitution, pair an expression and a dictionary

The pair is called a ***closure***

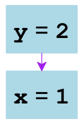
The dictionary is called an ***environment*** 

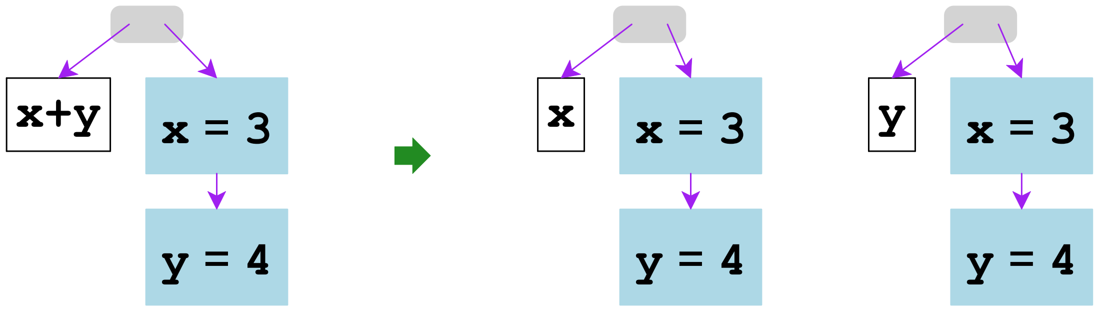


Closures

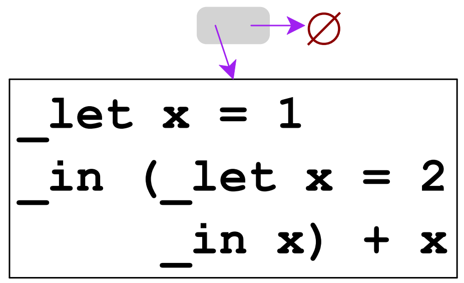
To accurately imitate substitution, pair an expression and a dictionary

The pair is called a ***closure***

The dictionary is called an ***environment*** 

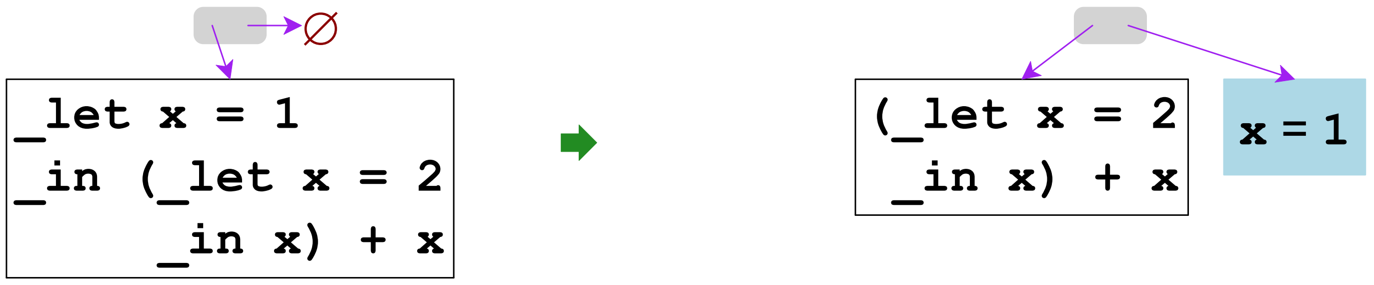


Different Environments



```
_let x = 1
_in (_let x = 2
     _in x) + x
```

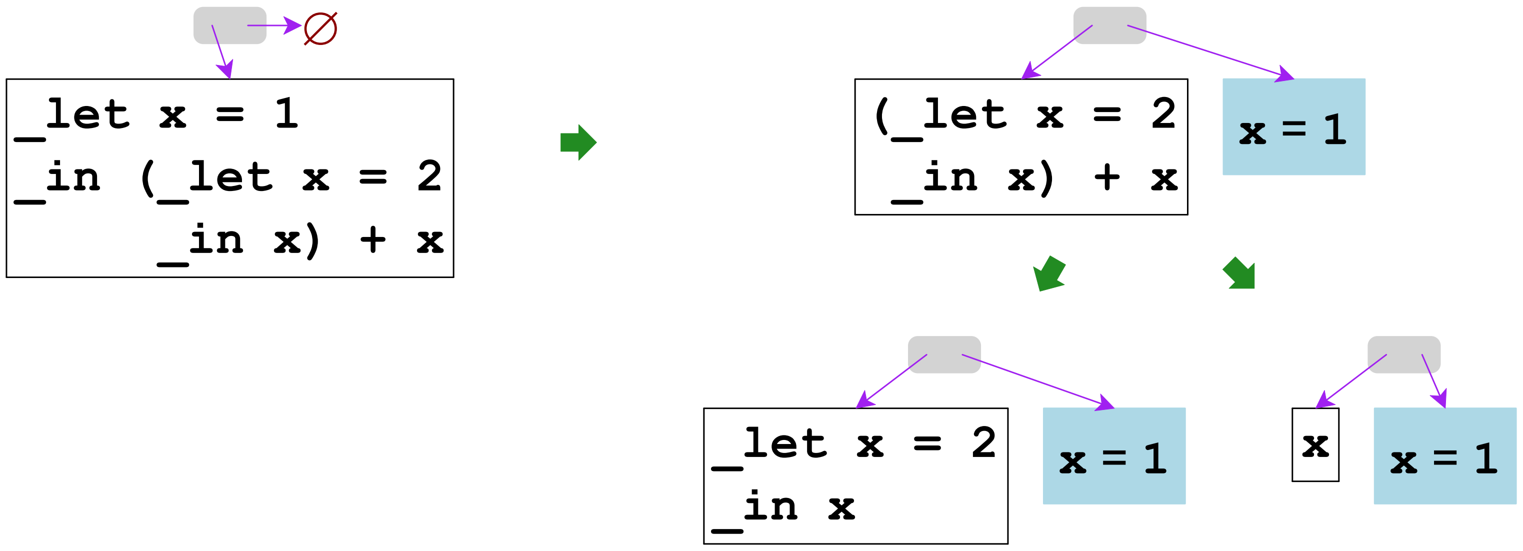
Different Environments



```
_let x = 1
_in (_let x = 2
      _in x) + x
```

```
(_let x = 2
  _in x) + x
```

Different Environments



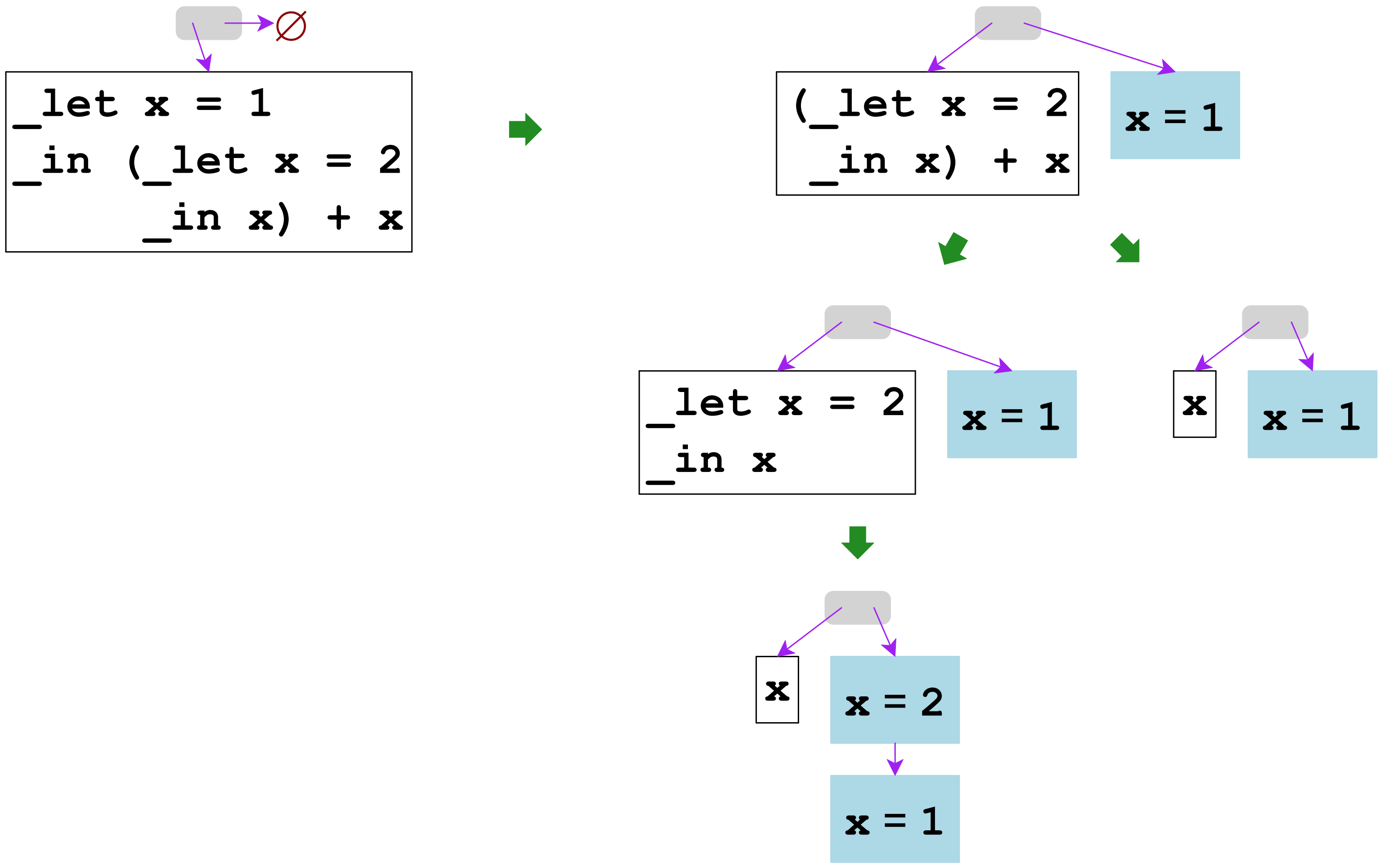
```
_let x = 1
_in (_let x = 2
      _in x) + x
```

```
(_let x = 2
  _in x) + x
```

```
_let x = 2
_in x
```

```
x
```

Different Environments



Representing Environments

```
class Env {
  virtual PTR(Val) lookup(std::string find_name) = 0;
};
```

Representing Environments

```
class Env {
  virtual PTR(Val) lookup(std::string find_name) = 0;
};
```

used by `VarExp::interp`

Representing Environments

```
class Env {
  virtual PTR(Val) lookup(std::string find_name) = 0;
};
```

An *environment* is either

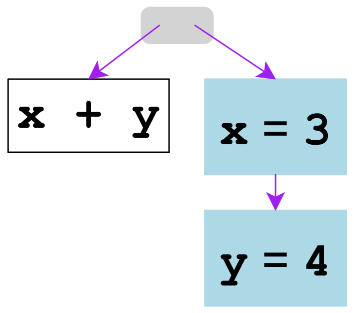
- empty
- a name and value added to an *environment*

```
class EmptyEnv : public Env {
  PTR(Val) lookup(std::string find_name) {
    throw std::runtime_error("free variable: "
                             + find_name);
  }
};
```

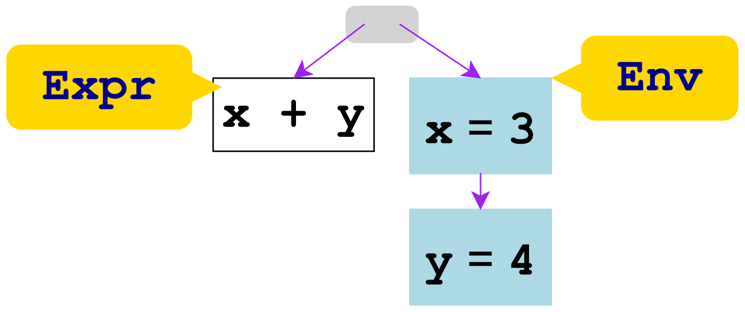
```
class ExtendedEnv : public Env {
  std::string name;
  PTR(Val) val;
  PTR(Env) rest;

  PTR(Val) lookup(std::string find_name) {
    if (find_name == name)
      return val;
    else
      return rest->lookup(find_name);
  }
};
```

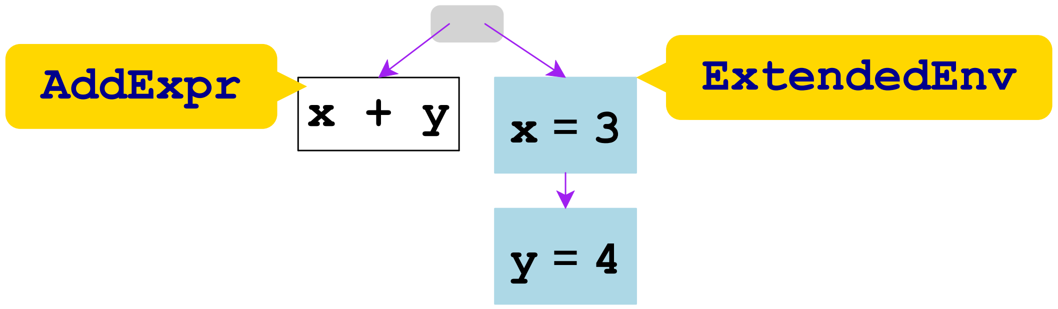
Closure Parts



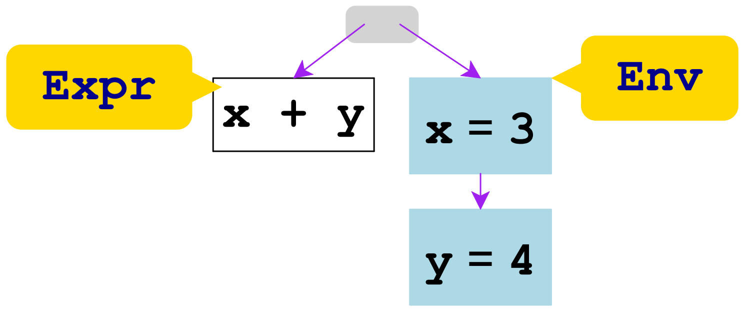
Closure Parts



Closure Parts



Closure Parts



Implicit Closures

Shortcut: Don't actually allocate a closure to interp it; instead, pass an environment to `interp`

```
class Expr {
  ....
  virtual PTR(Val) interp(PTR(Env) env) = 0;
};
```

So,

`expr->interp(env)`

evaluates the closure combining `body` and `env`

Implicit Closures

Shortcut: Don't actually allocate a closure to interp it; instead, pass an environment to `interp`

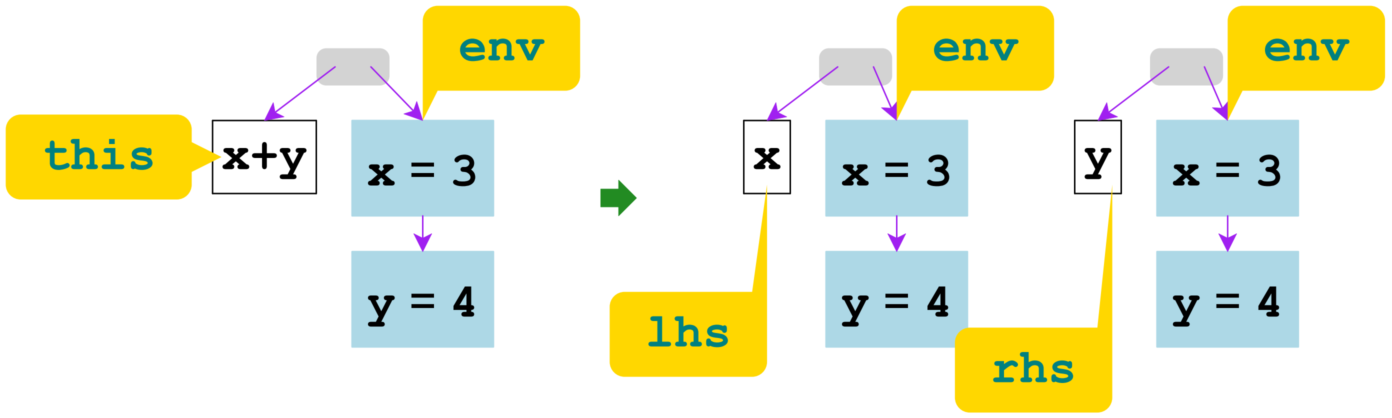
```
class Expr {
  ....
  virtual PTR(Val) interp(PTR(Env) env) = 0;
};
```

So,

```
Expr        Env
expr->interp(env)
```

evaluates the closure combining `body` and `env`

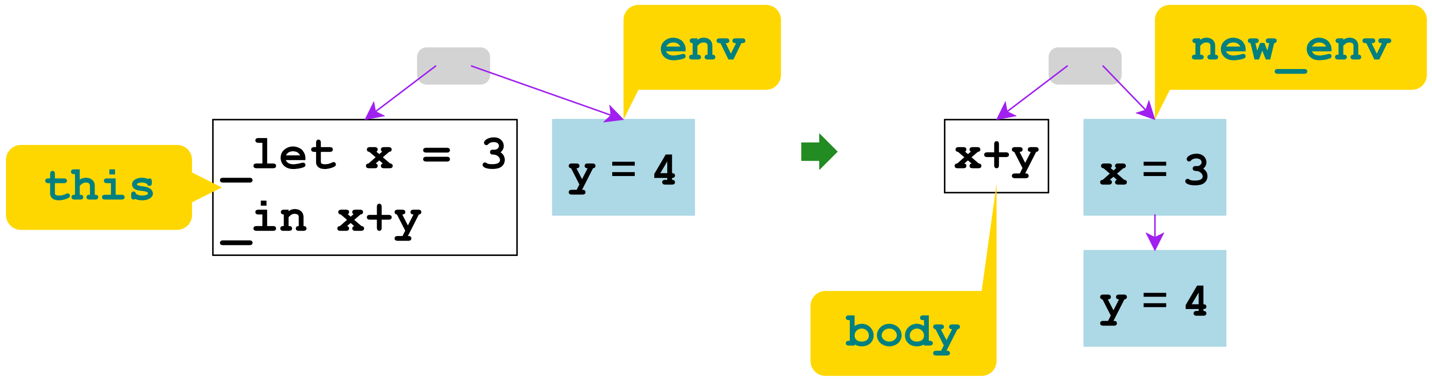
Interpreting Subexpressions



Passing **env** to subexpressions propagates the environment:

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

Interpreting Subexpressions



Extend **env** to to add a binding:

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

Allocating Explicit Closures

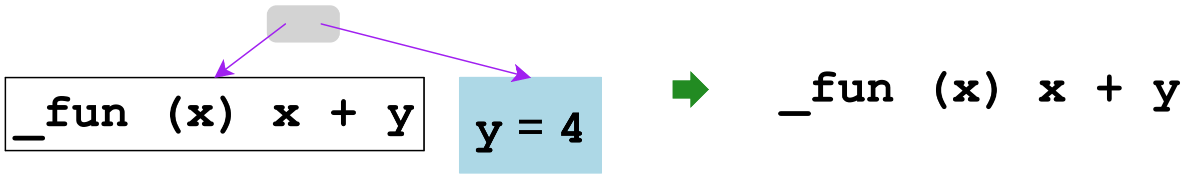
Passing an **`Env`** to **`interp`** mostly avoids the need to allocate closures

```
PTR(Val) FunExpr::interp(PTR(Env) env) {
  return NEW(FunVal)(formal_arg, body);
}
```

Allocating Explicit Closures

Passing an `Env` to `interp` mostly avoids the need to allocate closures

```
PTR(Val) FunExpr::interp(PTR(Env) env) {
  return NEW(FunVal)(formal_arg, body);
}
```

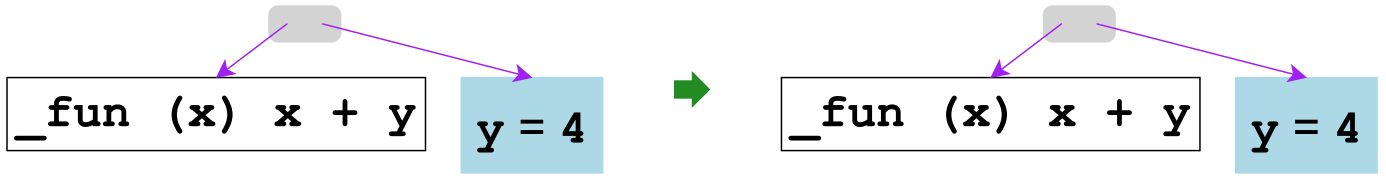


This would be **wrong**, because `body` loses its environment in a `FunVal`

Allocating Explicit Closures

Passing an **Env** to **interp** mostly avoids the need to allocate closures

```
~~PTR(Val) FunExpr::interp(PTR(Env) env) {~~
~~  return NEW(FunVal)(formal_arg, body);~~
~~}~~
```

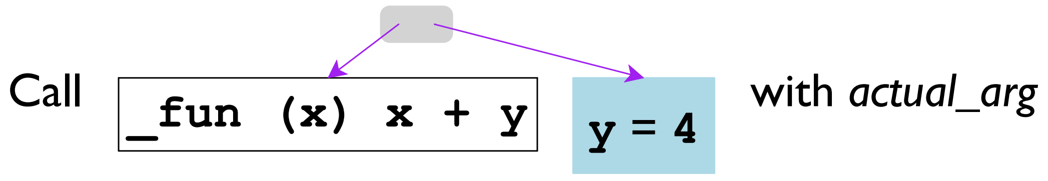


```
PTR(Val) FunExpr::interp(PTR(Env) env) {
  return NEW(FunVal)(formal_arg, body, env);
}
```

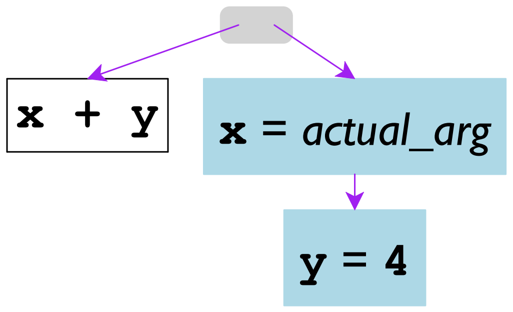
This is **right**, because **env** is kept with **body** in a **FunVal**

So, add an **env** field to **FunVal** (but not **FunExpr**)

Calling Functions



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



Interpreter Changes

- `Expr::interp` should not call `Expr::subst`, anymore
- `Expr::subst` can be removed
- `Val::to_expr` can be removed
- `Val::to_string` is needed to print `Expr::interpret` results
    - print function values as just `[function]`

Performance

`fib(fib)(28)`

	Debug	*Release*
substitution		
no `free`	4.38	2.49
`shared_ptr`	23.98	7.43
environment		
no `free`	1.05	0.59
`shared_ptr`	5.16	1.60
`racket -j`		0.14

Performance

`fib(fib)(28)`

	Debug	*Release*
substitution		
no `free`	4.38	2.49
`shared_ptr`	23.98	7.43
environment		
no `free`	1.05	0.59
`shared_ptr`	5.16	1.60
`racket -j`		0.14
`racket`		0.008

Performance

`fib(fib)(28)`

	Debug	*Release*
substitution		
no **`free`**	4.38	2.49
`shared_ptr`	23.98	7.43
environment		
no **`free`**	1.05	0.59
`shared_ptr`	5.16	1.60

`racket -j`	0.14
`racket`	0.008
`racket` direct	0.002
`g++ -O2` direct	0.002

"direct" means **`fib`** as a normal recursive function