Part I

State

Substitution relies on an identifier having a fixed value

```
let x = 5:
  let f = (fun (y): x + y):
    ....
    f(1)
```

=

```
let f = (fun (y): 5 + y):
  ....
  f(1)
```

because **x** cannot change

State

In some languages, a variable's value *can* change

```
> block:
    def mutable x = 0
    fun f(y): x + y
    x := 100
    f(1)
- Int
101
```

In those languages, a variable has ***state***

Inessential State: Summing a List

The Java way:

```
int sum(List<Integer> lst) {
  int t = 0;
  for (Integer n : lst) {
    t = t + n;
  }
  return t;
}
```

The Shplait way:

```
fun sum(lst :: Listof(Int)) :: Int:
  match lst
  | []: 0
  | cons(f, rst): f + sum(rst)
```

Inessential State: Summing a List

The Java way:

```
int sum(List<Integer> lst) {
  int t = 0;
  for (Integer n : lst) {
    t = t + n;
  }
  return t;
}
```

The Shplait way:

```
fun sum(lst :: Listof(Int), t :: Int):
  match lst
  | []: t
  | cons(f, rst): sum(rst, f + t)
```

Inessential State: Summing a List

The Java way:

```
int sum(List<Integer> lst) {
  int t = 0;
  for (Integer n : lst) {
    t = t + n;
  }
  return t;
}
```

The Shplait way:

```
fun sum(lst :: Listof(Int)) :: Int:
  foldl(fun (x, y): x + y, 0, lst)
```

Inessential State: Feeding Fish

The Java way:

```
void feed(int[] aq) {
  for (int i = 0; i < aq.length; i++) {
    aq[i]++;
  }
}
```

The Shplait way:

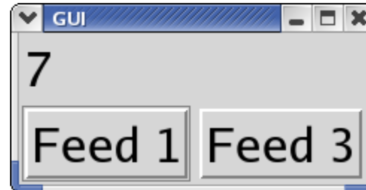
```
def feed :: Listof(Int) -> Listof(Int):
  fun (lst):
    map(fun (x): x + 1, lst)
```

Reasons to Avoid State

```
check:
  feed([4, 3, 7, 1])
  ~is [5, 4, 8, 2]
```

```
def today = [4, 3, 7, 1]
def tomorrow = feed(today)
compare(today, tomorrow)
```

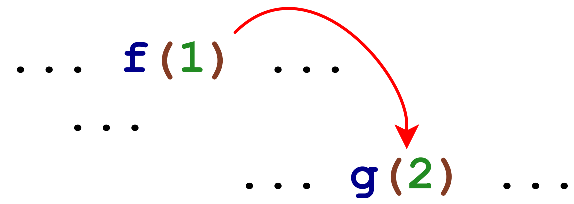
When State is Essential




```
def mutable weight = 0

def total_message = make_message(to_string(weight))

fun make_feed_button(label, amt):
  make_button(label,
              fun (evt):
                weight := weight + amt
                draw_message(total_message,
                             to_string(weight)))

create_window([[total-message],
               [make_feed_button("Feed 3", 3),
                make_feed_button("Feed 7", 7)]])
```

State as a Side Channel

State is a ***side channel*** for parts of a program to communicate



+ Programmer can add new channels at will

– Channels of communication may not be apparent

Part 2

Variables vs. Boxes

```
def mutable weight = 0

fun feed() :: void:
  weight := 1 + weight

fun get_size() :: Int:
  weight
```

Variables vs. Boxes

```
def weight = box(0)

fun feed() :: Void:
  set_box(weight, 1 + unbox(weight))

fun get_size() :: Int:
  unbox(weight)
```

```
box :: ?a -> Boxof(?a)
unbox :: Boxof(?a) -> ?a
set_box :: (Boxof(?a), ?a) -> Void
```

Boxes as Simple Objects

```
class Box<T> {
  T v;
  Box(T v) {
    this.v = v;
  }
}
```

```
let b = box(0):
  begin:
    set_box(b, 10)
    unbox(b)
```

```
Box b = new Box(0);

b.v = 10;
return b.v;
```

Boxes

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> - <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  box(<Exp>)                    NEW
         |  unbox(<Exp>)                  NEW
         |  set_box(<Exp>, <Exp>)         NEW
         |  begin: <Exp>; <Exp>           NEW
```

```
let b = box(0):
  begin:
    set_box(b, 10)
    unbox(b)
```

$\Rightarrow$ 10

Implementing Boxes

```
type Exp
...
| boxE(arg :: Exp)
| unboxE(arg :: Exp)
| setboxE(bx :: Exp,
          val :: Exp)
| beginE(l :: Exp,
         r :: Exp)
```

Part 3

Implementing Boxes with Boxes

```
let b = box(0):
  begin:
    set_box(b, 10)
    unbox(b)
```

⟹ 10

Implementing Boxes with Boxes

```
box(0)
```

Implementing Boxes with Boxes

```
box(0)
```

⇒ ... a box containing `intV(0)` ...

Implementing Boxes with Boxes

```
type Value
| intV(n :: Int)
| closV(arg :: Symbol,
        body :: Exp,
        env :: Env)
| boxV(b :: Boxof(Value))
```

Implementing Boxes with Boxes

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | ...
  | boxE(a):
      boxV(box(interp(a, env)))
  | unboxE(a):
      match interp(a, env)
      | boxV(b): unbox(b)
      | ~else: error(#'interp, "not a box")
  | setboxE(bx, val):
      match interp(bx, env)
      | boxV(b): let v = interp(val, env):
                   block: set_box(b, v)
                          v
      | ~else: error(#'interp, "not a box")
  | beginE(l, r): block:
                    interp(l, env)
                    interp(r, env)
```

This doesn't explain anything about boxes!

Part 4

State and `interp`

We don't need state to `interp` state

- We control all the channels of communication
- Communicate the current values of boxes explicitly

Boxes and Memory

```
let b = box(7):
  ....
```

⇒ `....`

Memory:

Memory:

			7	

Boxes and Memory

```
.... set_box(b, 10)
....
```

Memory:

			7	

⇒

```
.... unbox(b)
....
```

Memory:

			10	

Communicating Memory

```
interp(....,   ) ⇒ ....
```

Communicating Memory

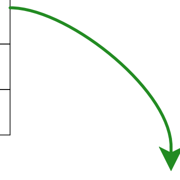
Memory:

			7	

```
interp(....,   ) ⇒ ....
```

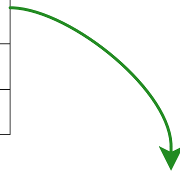
Communicating Memory

Memory:




```
interp(....,  ) ⇒ ....
```

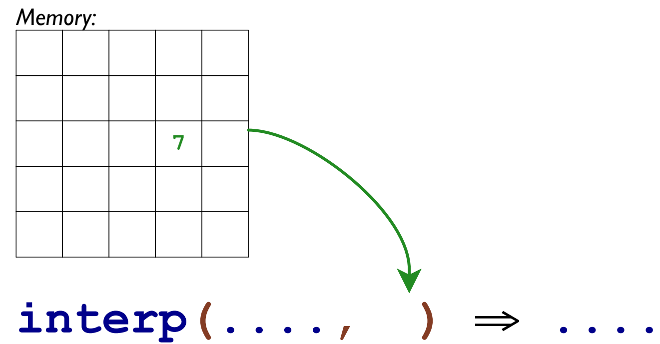
Communicating Memory




```
interp(....,  ) ⇒ ....
```

```
interp :: (Exp, Env) -> Value
```

Communicating Memory

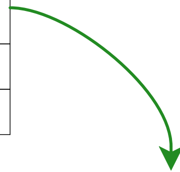



```
interp :: (Exp, Env, Store) -> Value
```

Communicating Memory

Memory:

7



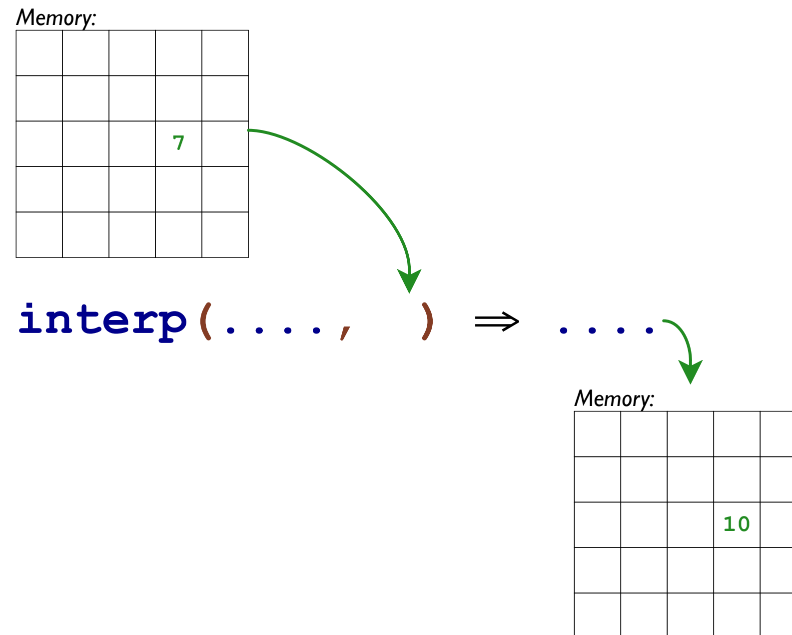
interp(....,   ) ⇒ ....

Memory:

10

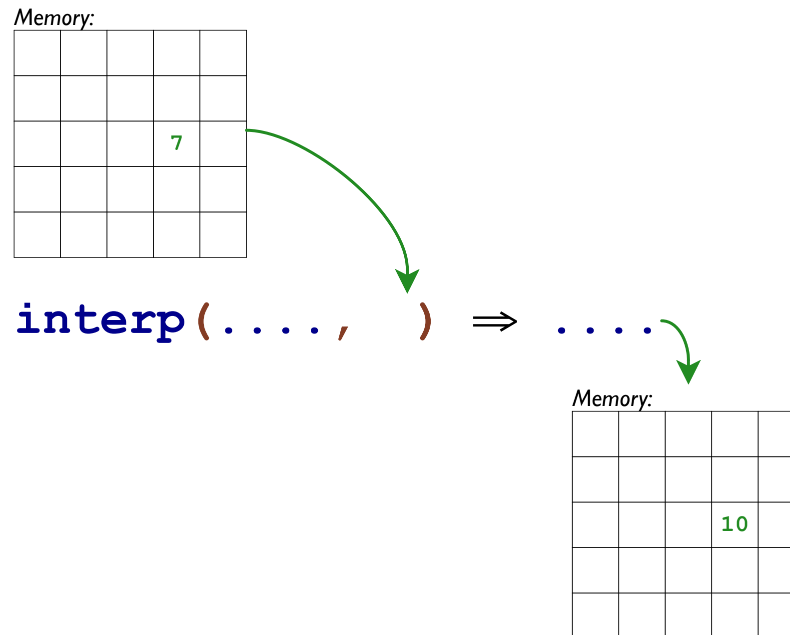
```
interp :: (Exp, Env, Store) -> Value
```

Communicating Memory



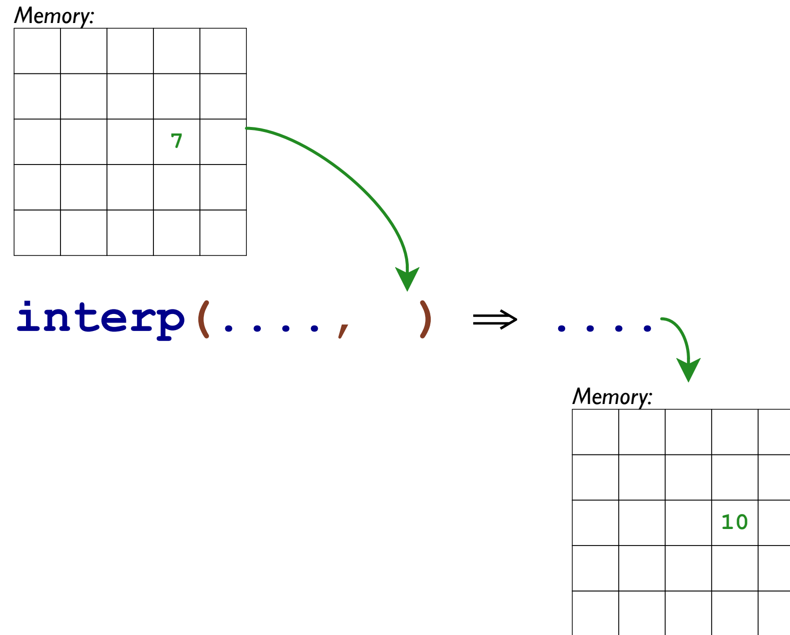

```
interp :: (Exp, Env, Store) -> Value
```

Communicating Memory




```
interp :: (Exp, Env, Store) -> Result
```

Communicating Memory




```
interp :: (Exp, Env, Store) -> Result

type Result
| res(v :: Value, s :: Store)
```

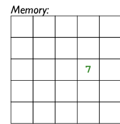
Communicating the Store

```
num_plus(interp(l, env), interp(r, env))
```

Communicating the Store

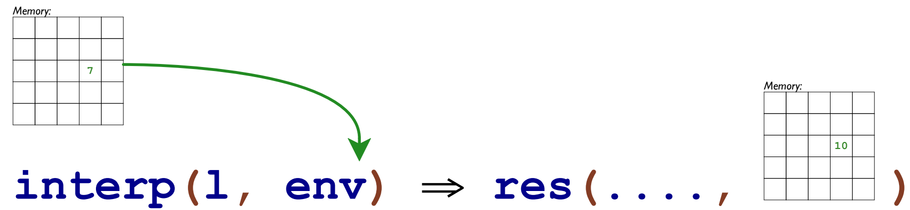
```
interp(l, env)


interp(r, env)

num_plus(...., ....)
```

Communicating the Store



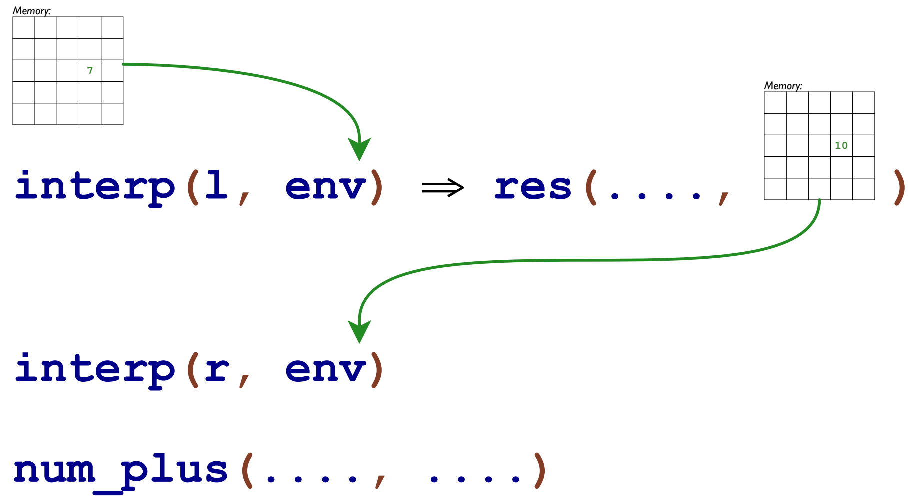

```
interp(l, env)


interp(r, env)

num_plus(...., ....)
```

Communicating the Store

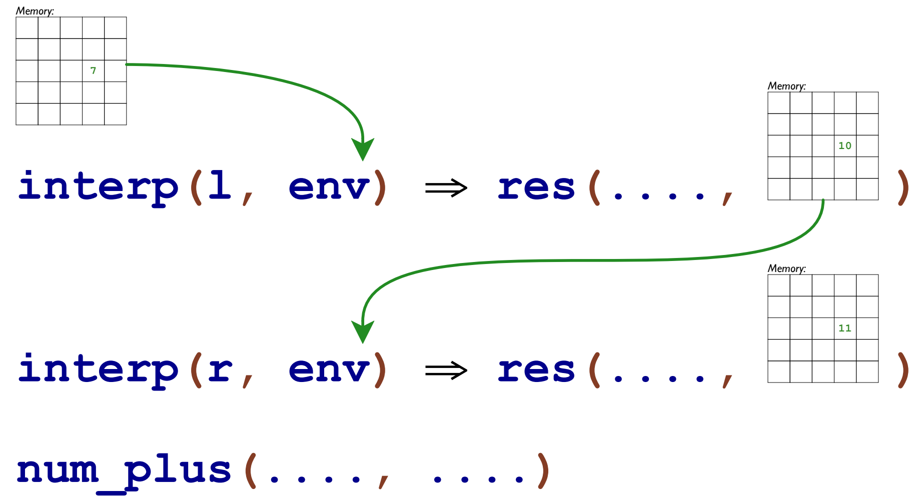



```
interp(r, env)

num_plus(...., ....)
```

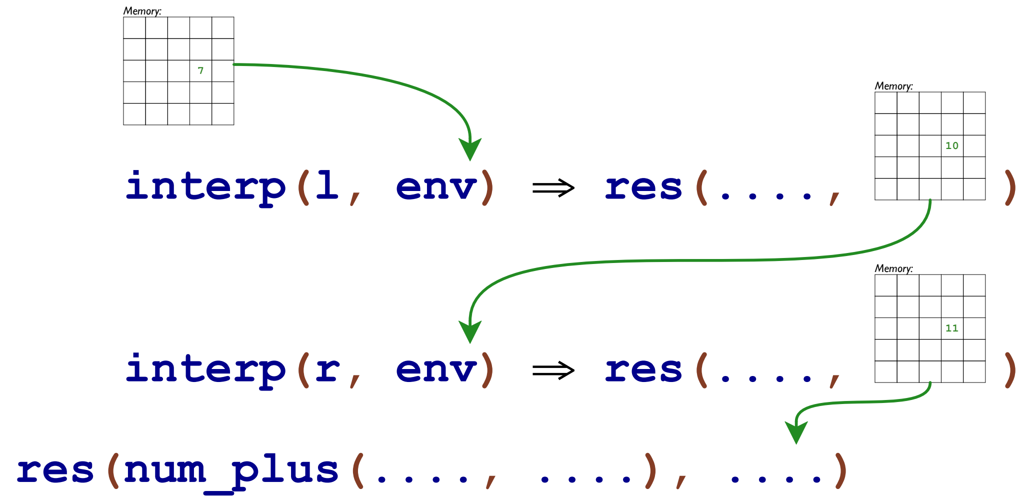
Communicating the Store



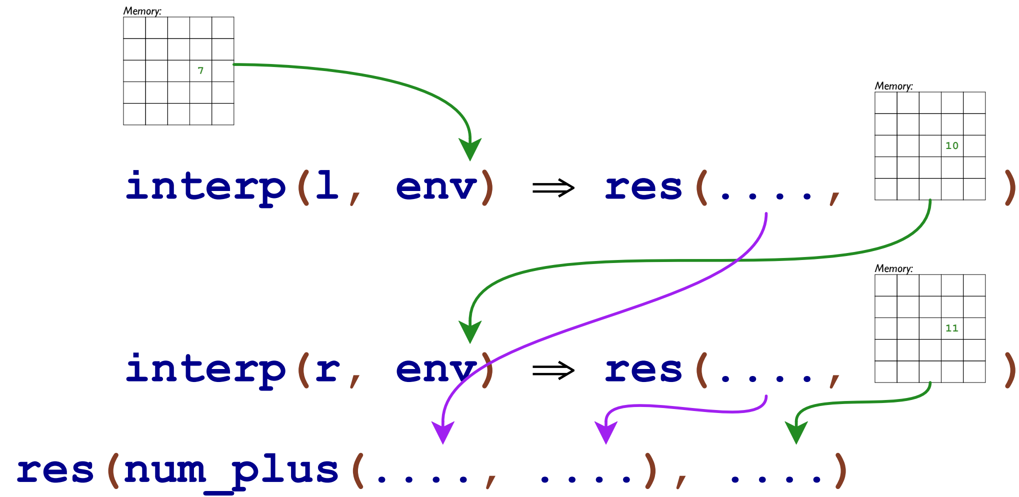
Communicating the Store



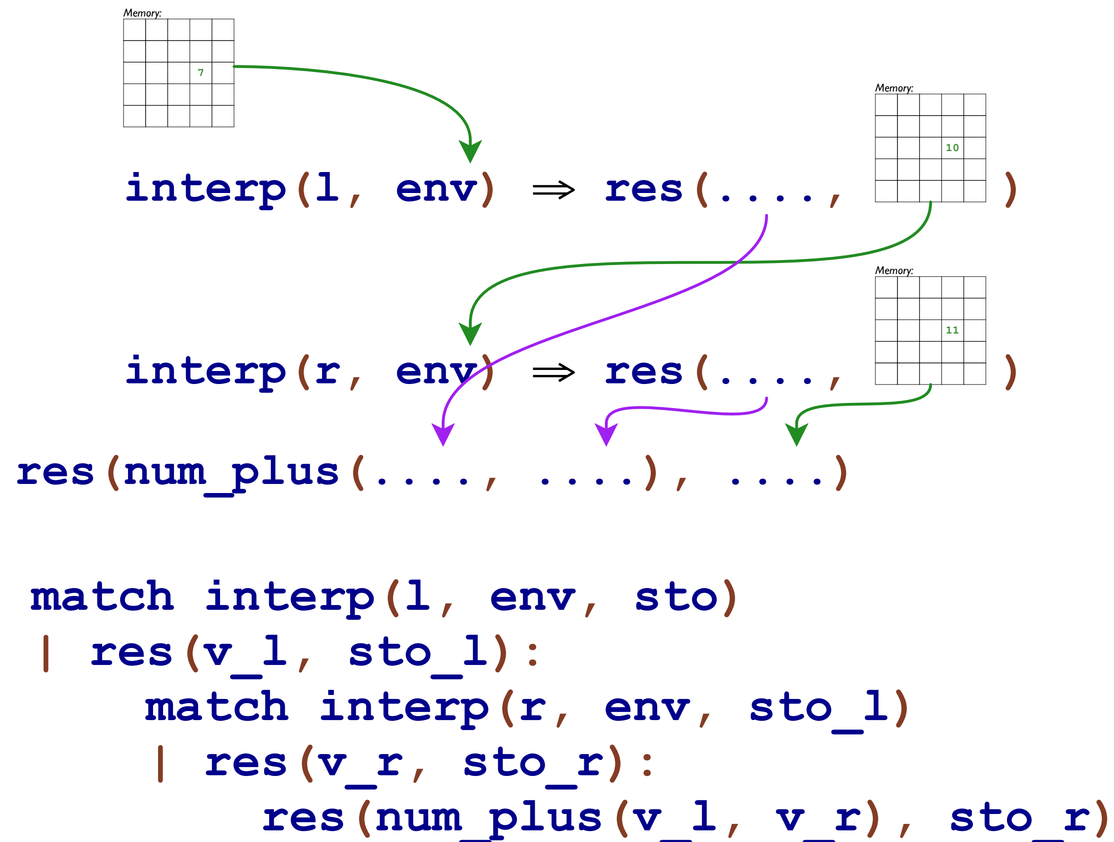
Communicating the Store



Communicating the Store



Communicating the Store



The Store

```
type Location = Int

type Storage
| cell(location :: Location, val :: Value)

type Store = Listof(Storage)
def mt_store = empty
def override_store = cons
```

Memory:

			10	

```
override_store(cell(13, intV(10)),
               mt_store)
```

The Store

```
type Location = Int

type Storage
| cell(location :: Location, val :: Value)

type Store = Listof(Storage)
def mt_store = empty
def override_store = cons
```

Memory:

	4			
			10	

```
override_store(cell(1, intV(4)),
               override_store(cell(13, intV(10)),
                              mt_store))
```

Part 5

Store Examples

```
interp :: (Exp, Env) -> Value

check: interp(intE(5), mt_env)
       ~is intV(5)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Value

check: interp(intE(5), mt_env, mt_store)
       ~is intV(5)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(intE(5), mt_env, mt_store)
       ~is res(intV(5), mt_store)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(boxE(intE(5)), mt_env, mt_store)
       ~is ....
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(boxE(intE(5)), mt_env, mt_store)
       ~is res(....,
               ....)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(boxE(intE(5)), mt_env, mt_store)
       ~is res(boxV(....),
               ....)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(boxE(intE(5)), mt_env, mt_store)
       ~is res(boxV(....),
               ..., intV(5), ....)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(boxE(intE(5)), mt_env, mt_store)
       ~is res(boxV(....),
               ...., cell(1, intV(5)), ....)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(boxE(intE(5)), mt_env, mt_store)
       ~is res(boxV(1),
               ...., cell(1, intV(5)), ....)
```

```
type Value
| intV(n :: Int)
| closV(arg :: Symbol,
        body :: Exp,
        env :: Env)
| boxV(l :: Location)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(boxE(intE(5)), mt_env, mt_store)
       ~is res(boxV(1),
               override_store(cell(1, intV(5)),
                              mt_store))
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('set_box(box(5), 6)'),
              mt_env,
              mt_store)
       ~is res(....,
                ....)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('set_box(box(5), 6)'),
              mt_env,
              mt_store)
       ~is res(intV(6),
               ....)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('set_box(box(5), 6)'),
              mt_env,
              mt_store)
       ~is res(intV(6),
               ....,
               ....,
               override_store(cell(1, intV(5)),
                              mt_store),
               ....)
```

Store Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('set_box(box(5), 6)'),
              mt_env,
              mt_store)
       ~is res(intV(6),
               override_store(cell(1, intV(6)),
                              override_store(cell(1, intV(5)),
                                             mt_store)))
```

Store Examples

See **`store.rhm`** for more examples

Part 6

`interp` with a Store

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | intE(n): res(intV(n), sto)
    ....
```

interp with a Store

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | idE(s): res(lookup(s, env), sto)
    ....
```

`interp` with a Store

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | plusE(l, r):
        match interp(l, env, sto)
        | res(v_l, sto_l):
            match interp(r, env, sto_l)
            | res(v_r, sto_r):
                res(num_plus(v_l, v_r), sto_r)
    ....
```

`interp` with a Store

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | boxE(a):
        match interp(a, env, sto)
        | res(v, sto_v):
            block:
              def l = new_loc(sto_v)
              res(boxV(l),
                  override_store(cell(l, v),
                                 sto_v))
    ....
```

`interp` with a Store

```
fun new_loc(sto :: Store) :: Location:
  1 + max_address(sto)

fun max_address(sto :: Store) :: Location:
  match sto
  | empty: 0
  | cons(c, rst_sto): max(cell.location(c),
                          max_address(rst_sto))
```

`interp` with a Store

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | unboxE(a):
        match interp(a, env, sto)
        | res(v, sto_v):
            match v
            | boxV(l): res(fetch(l, sto_v),
                           sto_v)
            | ~else: error(#'interp,
                           "not a box")
    ....
```

`interp` with a Store

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | setboxE(bx, val):
        match interp(bx, env, sto)
        | res(v_b, sto_b):
            match interp(val, env, sto_b)
            | res(v_v, sto_v):
                match v_b
                | boxV(l):
                    res(v_v,
                        override_store(cell(l, v_v),
                                       sto_v))
                | ~else: error(#'interp,
                               "not a box")
    ....
```

`interp` with a Store

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | beginE(l, r):
        match interp(l, env, sto)
        | res(v_l, sto_l):
            interp(r, env, sto_l)
    ....
```

Part 7

Awkward Syntax

```
match interp(l, env, sto)
| res(v_l, sto_l):
    match interp(r, env, sto_l)
    | res(v_r, sto_r):
        res(num_plus(v_l, v_r), sto_r)
```

```
match call
| res(v_id, sto_id):
    body
```

Better Syntax

```
match call
| res(v_id, sto_id):
    body
```

Better Syntax

```
match call
| res(v_id, sto_id):
    body

reslet (v_id, sto_id) = call:
  body
```

Better Syntax

```
match $call
| res($v_id, $sto_id):
    $body

reslet ($v_id, $sto_id) = $call:
  $body
```

Better Syntax

```
    reslet ($v_id, $sto_id) = $call:
      $body
match $call
| res($v_id, $sto_id):
    $body
```

Better Syntax

```
macro 'reslet ($v_id, $sto_id) = $call:
           $body':
   'match $call
    | res($v_id, $sto_id):
        $body'
```

Better Syntax

```
macro 'reslet ($v_id, $sto_id) = $call:
         $body':
  'match $call
   | res($v_id, $sto_id):
       $body'
```

```
reslet (v_r, sto_r) = interp(r, env, sto_l):
  res(num_plus(v_l, v_r), sto_r)
```

Better Syntax

```
macro 'reslet ($v_id, $sto_id) = $call:
         $body':
  'match $call
   | res($v_id, $sto_id):
       $body'
```

```
reslet (v_r, sto_r) = interp(r, env, sto_l):
  res(num_plus(v_l, v_r), sto_r)
⇒
match interp(r, env, sto_l)
| res(v_r, sto_r):
    res(num_plus(v_l, v_r), sto_r)
```

Better Syntax

```
reslet (v_l, sto_l) = interp(l, env, sto):
  reslet (v_r, sto_r) = interp(r, env, sto_l):
    res(num_plus(v_l, v_r), sto_r)
```

See **`store_reslet.rhm`**