EU FP 6 EmBounded: IST-2004-510255 2005-2008

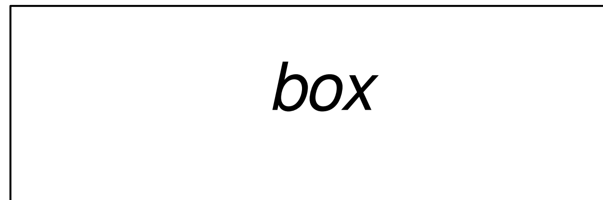
EmBounded Workshop, Budapest, September 2006

Programming in Hume

Greg Michaelson/Robert Pointon
School of Mathematical & Computer Sciences
Heriot-Watt University

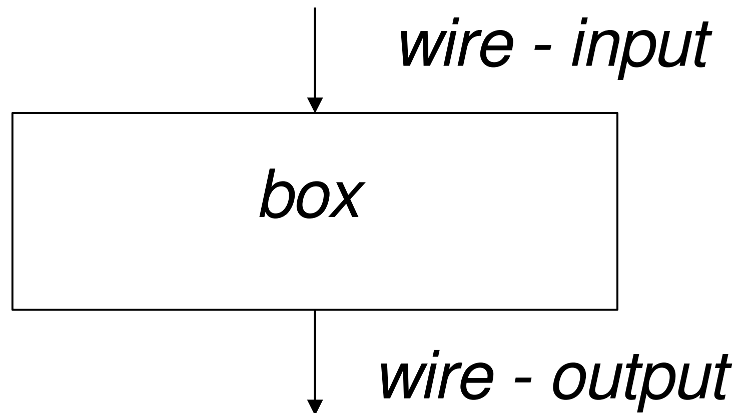
Boxes and wires

- Hume programs are built from *boxes*



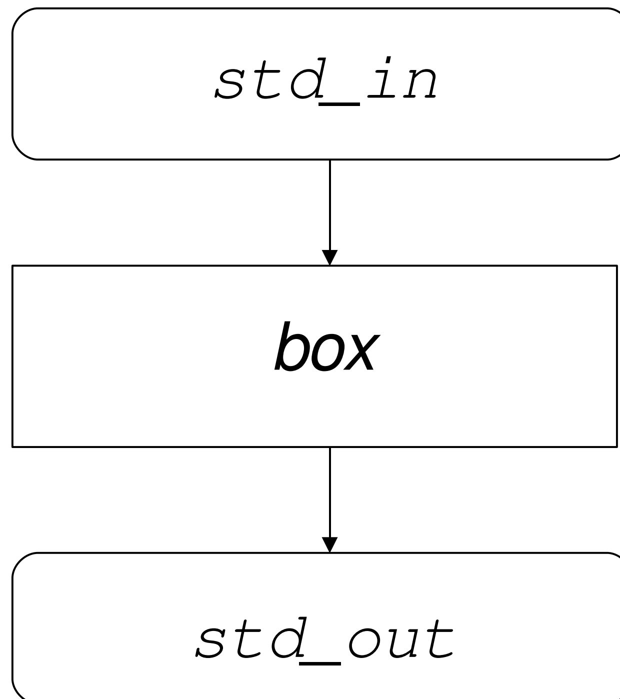
Boxes and wires

- boxes have *input* and *output wires*



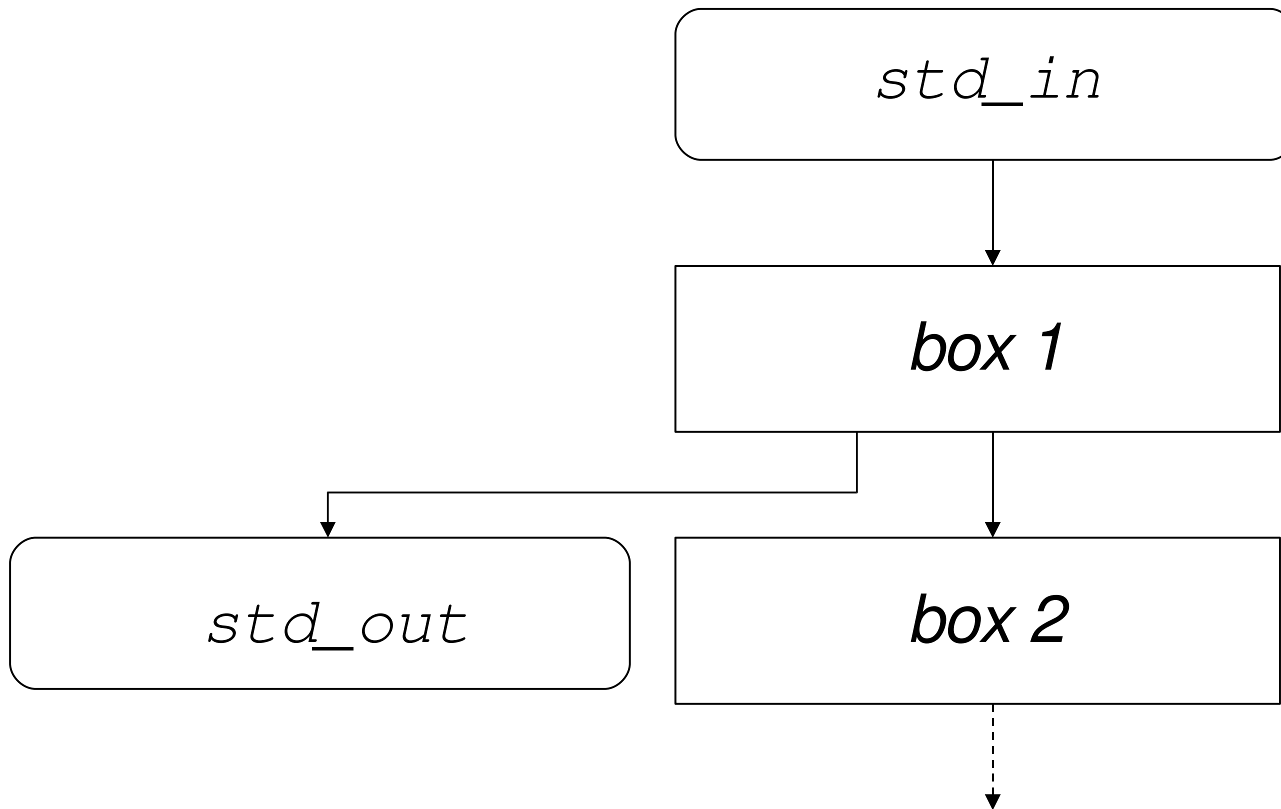
Boxes and wires

- wires may connect boxes to world via *streams*



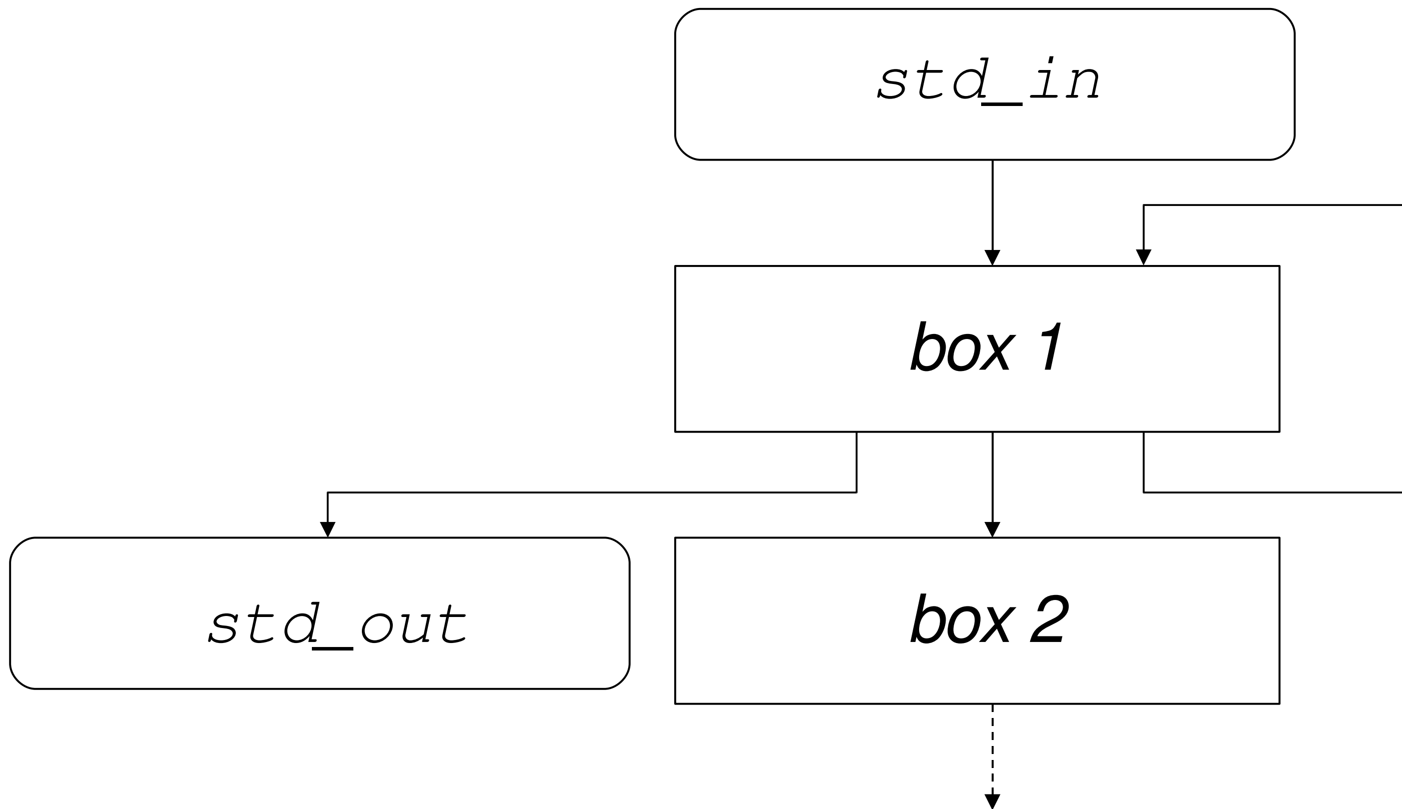
Boxes and wires

- wires may connect boxes to *other boxes*



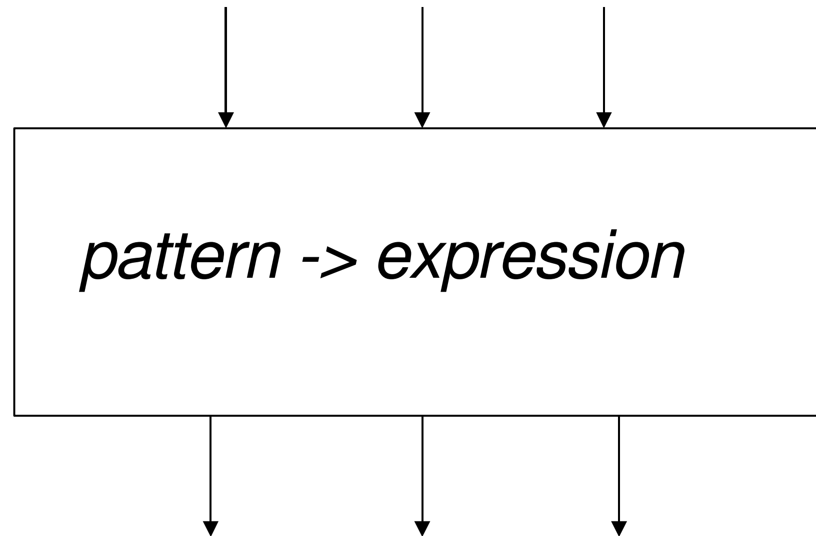
Boxes and wires

- wires may connect boxes *to themselves*



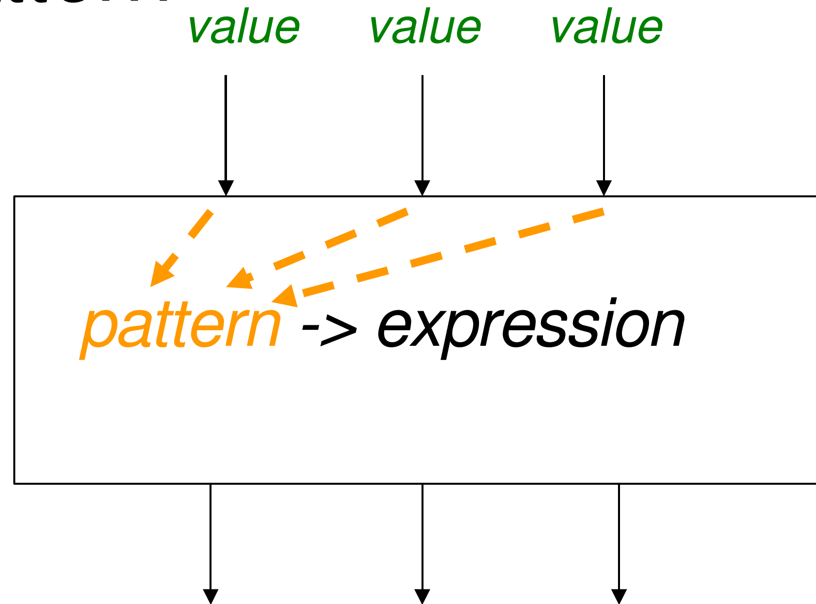
Inside boxes

- boxes are controlled by transitions from *patterns* to *expressions*



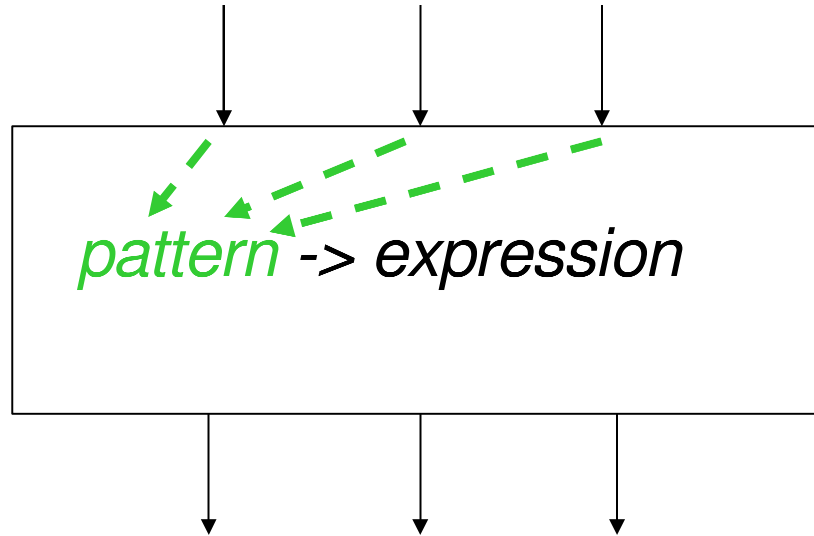
Inside boxes

- box attempts to matches input *values* on wires to *pattern*



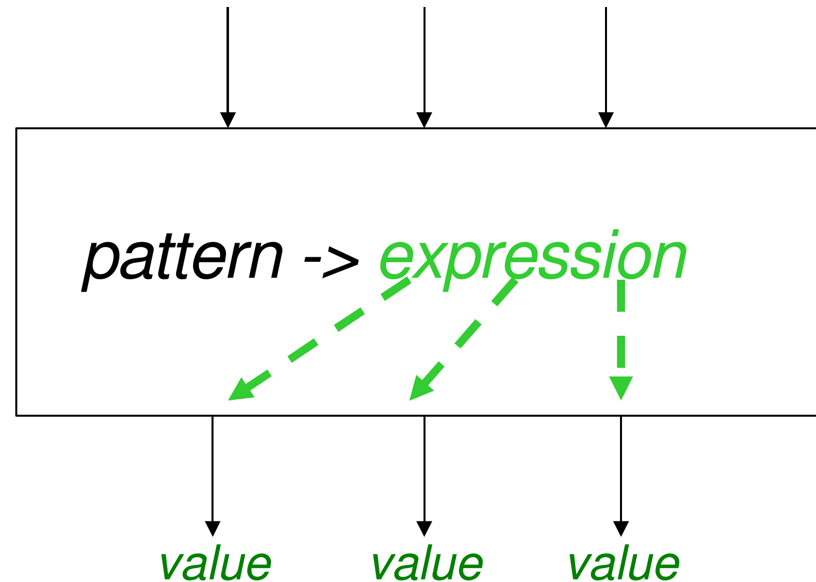
Inside boxes

- for successful match, input is consumed to bind *id*s in *pattern*



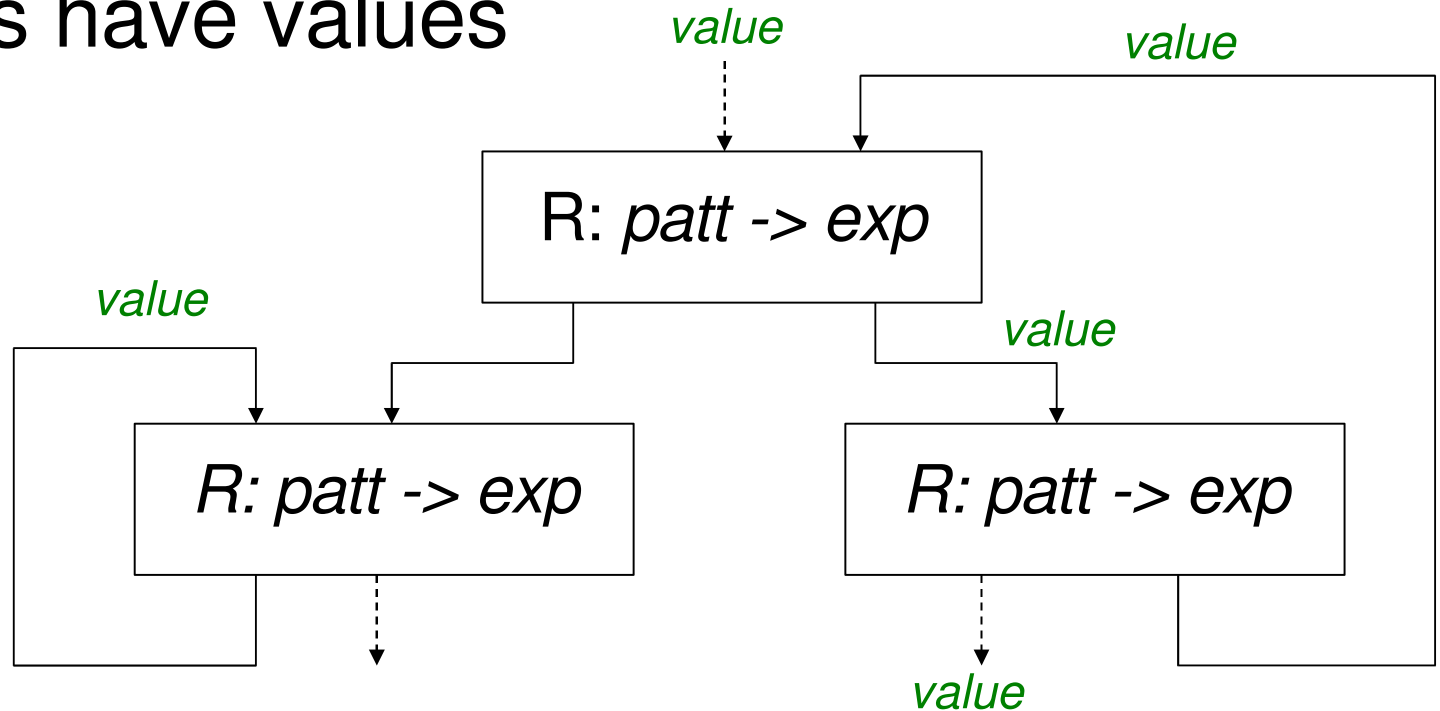
Inside boxes

- output values then generated by associated *expression*



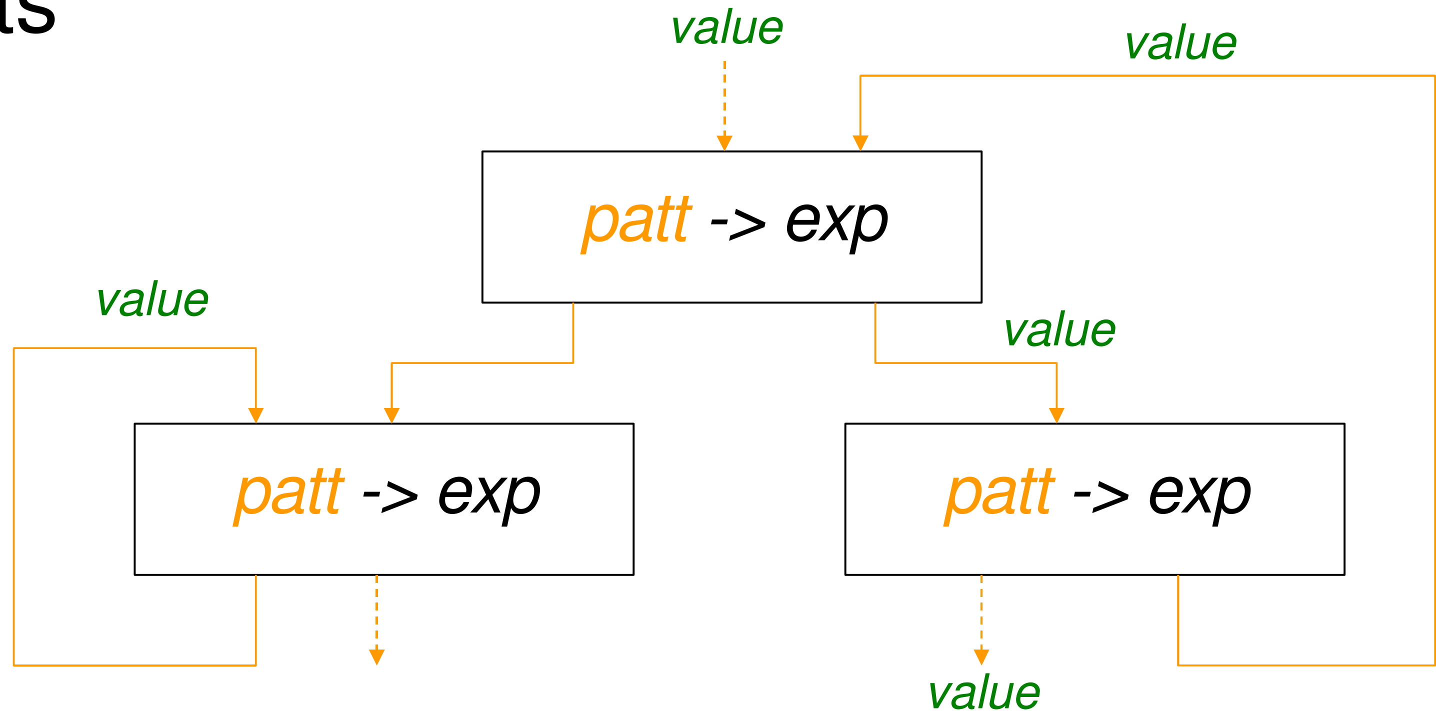
Program execution

- initially, all boxes are *ready* (R) and some wires have values



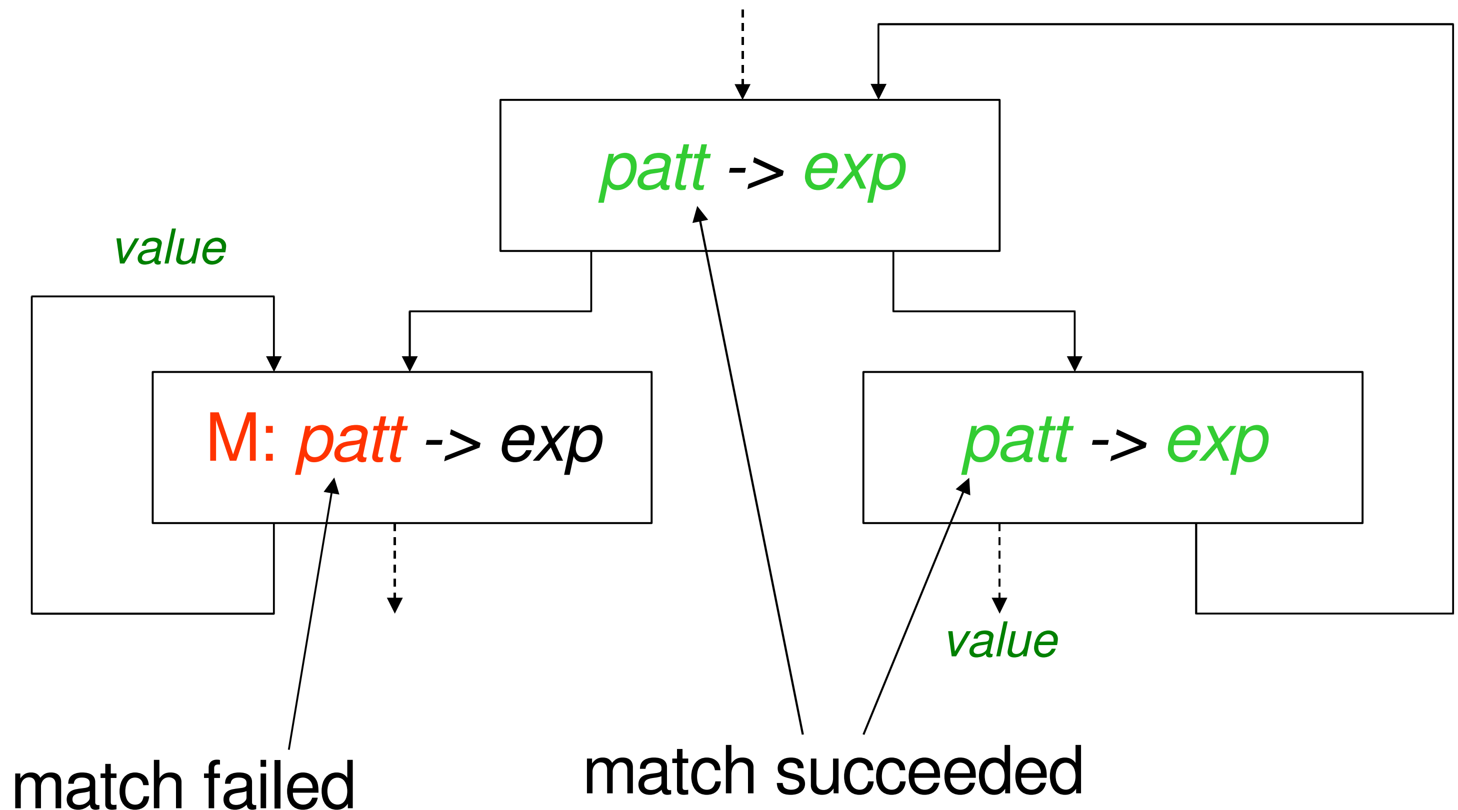
Program execution

- all boxes simultaneously attempt to *match* inputs



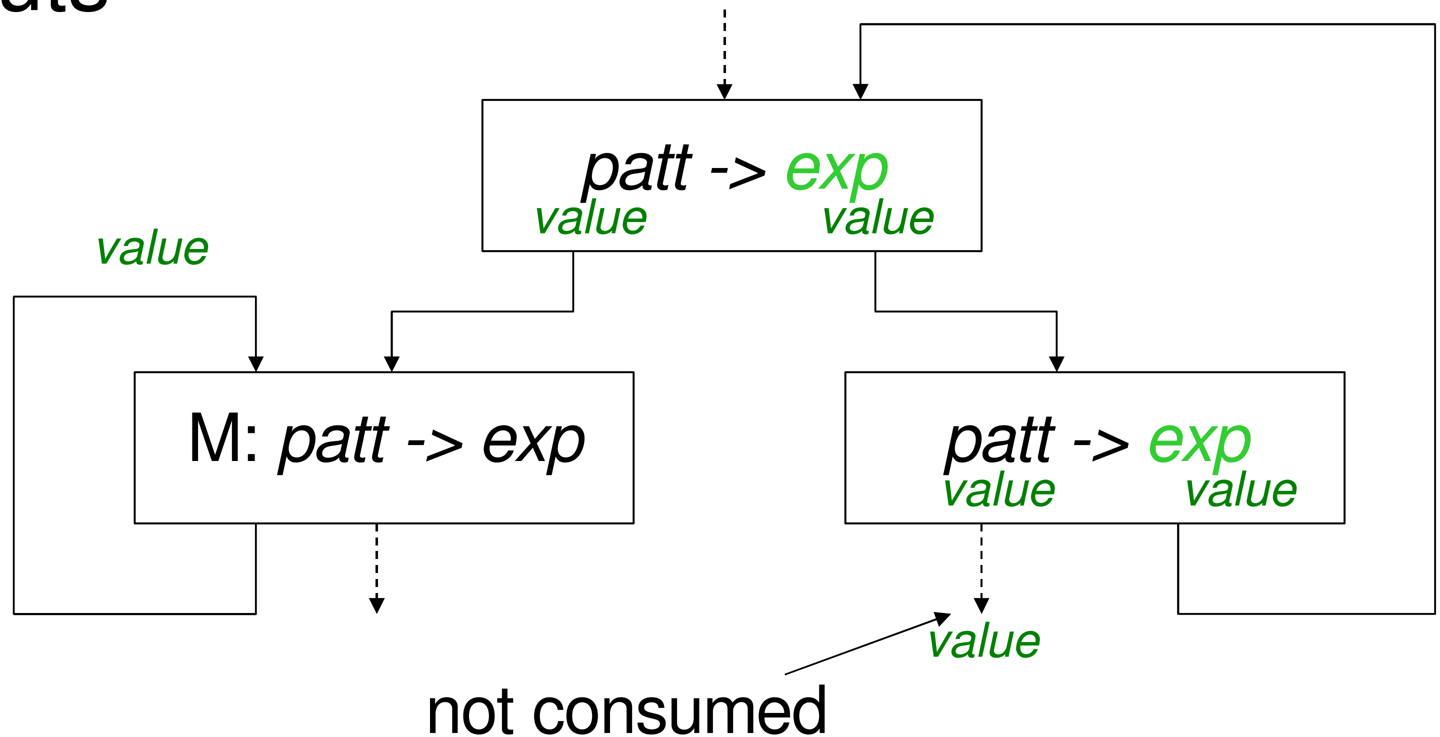
Program execution

- boxes that find matching inputs *consume* them & evaluate associated *expression*



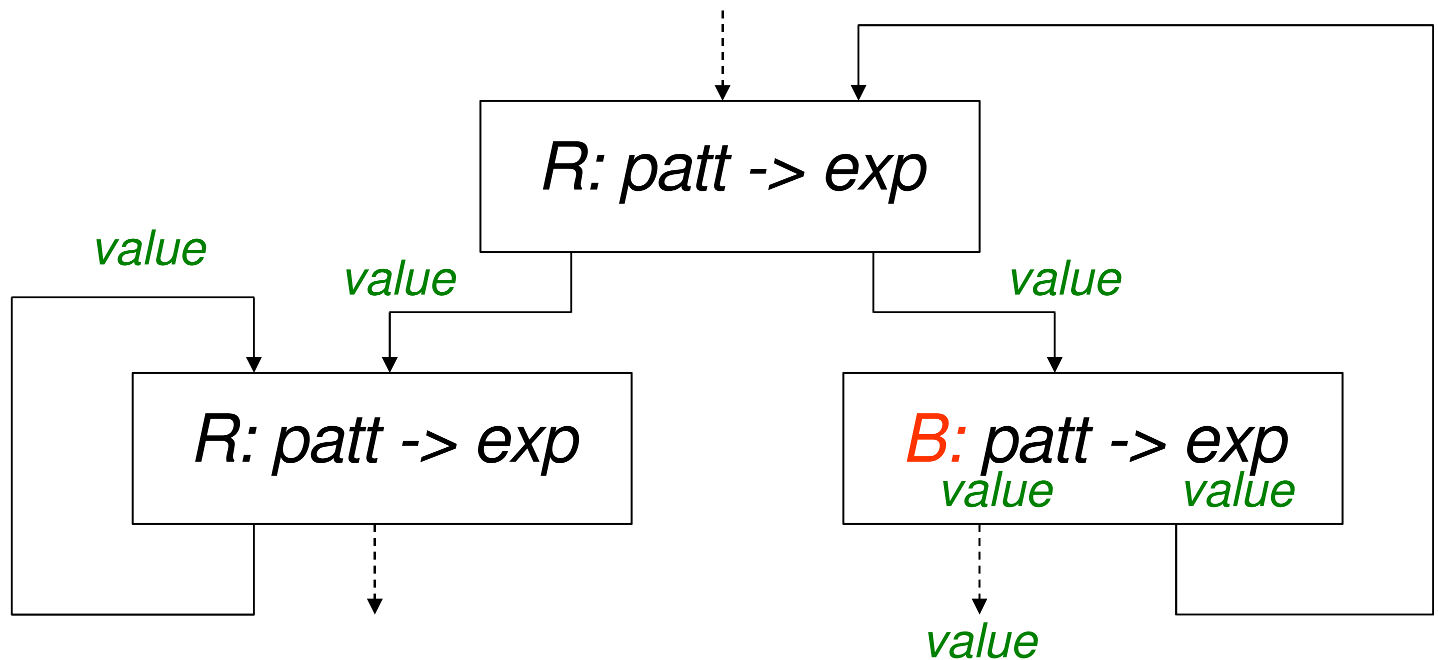
Program execution

- on the *superstep*, boxes attempt to *assert* outputs



Program execution

- boxes whose previous outputs are not consumed are *blocked (B)* until next cycle



Types

- sized types
  - `int` *size*; `float` *size*; `word` *size*; `char`
  - `vector` *size* `of` *type*
  - tuple: ($type_1$, $type_2$, ... $type_N$)
- unsized types
  - list: `[`*type*`]`
  - `string`
  - union: `data` *id* `=` $cons_1$ [$types_1$] `|`
    $cons_2$ [$types_2$] `| . . .`

Values

- sized values
  - `64; 64.78; 11011; 'a'`
  - `<<1,2,3,4,5>>`
  - `(1,1.0,"one")`
- unsized values
  - `[[1,2,3],[4,5,6],[7,8,9]]`
  - `"banana"`
  - `data STATE = ON | OFF`

Patterns

id	consume input & bind to *id*
constant	match input & consume
_	*wild card* - consume input
$(patt_1,patt_2,...patt_N)$	tuple
$[patt_1,patt_2,...patt_N]$	list
cons [*patts*]	constructor

- input must be present

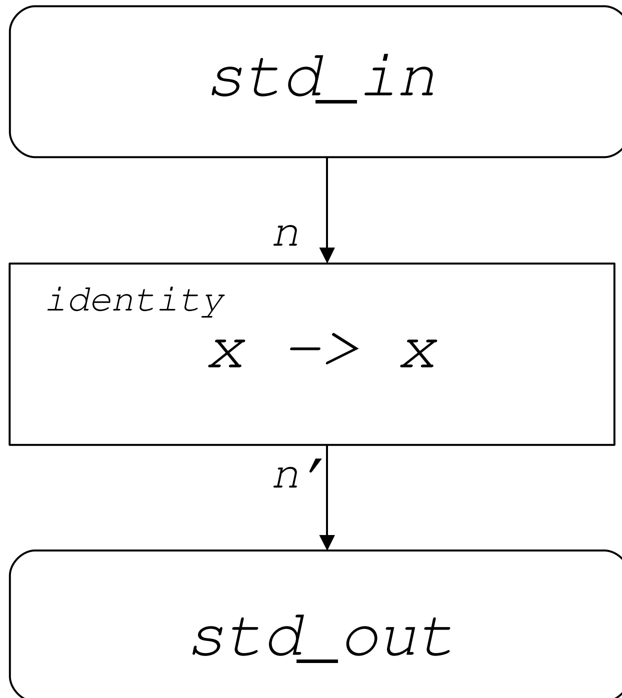
Ignore/no output

- * in *pattern*
  - ignore input i.e. do not consume
  - match always succeeds even if no input

- * in *expression*
  - do not generate output

identity.hume

- sends input to output




```
box identity
in (n::int 64)
out (n'::int 64)
match x -> x;

stream input from
   "std_in";

stream output to
   "std_out";

wire identity
      (input) (output);
```

identity.hume

$ hume identity.hume

HUME 0.2

Tuesday 01 August, 2006, 13:54

identity.hume

RUNNING

1

1 **2**

2 **3**

3 **4**

4...

- output not terminated by newline...

identity1.hume

```
box identity1
in (n::(int 64,char))
out (n'::(int 64,char))
match x -> (x,'\n');

stream input from
   "std_in";

stream output to
   "std_out";

wire identity
      (input) (output);
```

```
$ hume identity1.hume
...
RUNNING
1
1
2
2
3
3
4
4
```

square.hume

```
box square
in (n::int 64)
out (n'::(int 64,char))
match x -> (x*x,'\n');

stream input from
   "std_in";

stream output to
   "std_out";

wire square
   (input) (output);
```

```
$ hume square.hume
...
RUNNING
1
1
2
4
3
9
4
16
...
```

square1.hume

```
sq x = x*x;

box square1
in (n::int 64)
out (n'::(int 64,char))
match x -> (sq x,'\n');

stream input from
   "std_in";

stream output to
   "std_out";

wire square1
 (input) (output);
```

```
$ hume square.hume
...
RUNNING
1
1
2
4
3
9
4
16
...
```

calculator.hume

```
...
box calculator
in (calc::
   (integer,char,integer))
out (res::(integer,char))
match
 (x,'+',y) -> (x+y,'\n') |
 (x,'-',y) -> (x-y,'\n') |
 (x,'*',y) -> (x*y,'\n') |
 (x,'/',y) ->
              (x div y,'\n');
...
wire calculator
         (input) (output);
```

```
$hume calculator.hume
...
RUNNING
```
1+1

2

3*8

24

121/17

7

12-14

-2

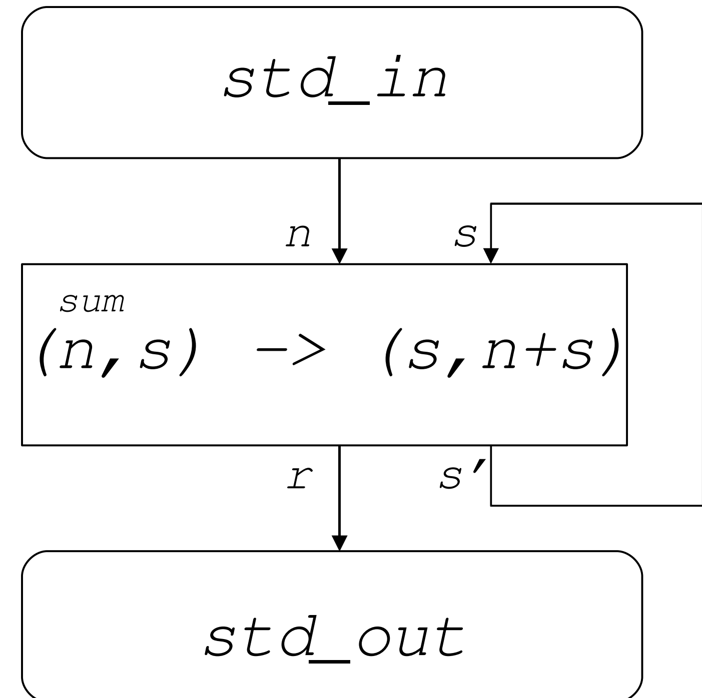
...

Feedback wires

- boxes are *stateless*
  - do not keep local state between execution cycles
- can maintain intermediate values on *feedback* wires
  - wire from box to itself
- must *initialise* feedback wires for first cycle to succeed

sum.hume

- show running total of input values
- keep current total on feedback wire
- show current total on each cycle



sum.hume

```
...
box sum
in (n,s::integer)
out (r::(integer,char),
     s'::integer)
match
 (n,s) -> ((s,'\n'),n+s);
  ...
wire sum
 (input,
  sum.s' initially 0)
 (output,sum.s);
```

```
$ hume sum.hume
...
RUNNING
1
0
2
1
3
3
4
6
...
```

sqdouble.hume

- alternate between squaring and doubling inputs
- use constructor to denote state:

```
data STATE = SQUARE | DOUBLE;
```

- flip state on each cycle

sqdouble.hume

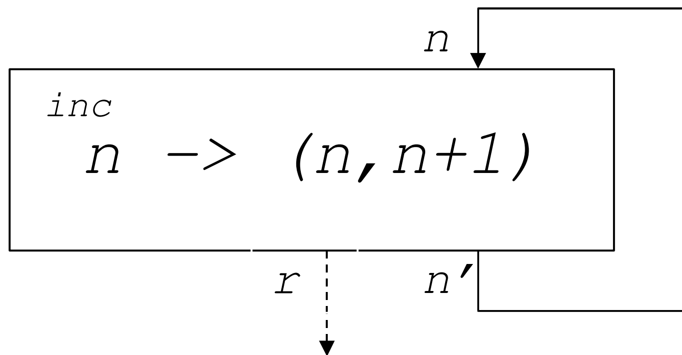
```
...
twice x = 2*x;

box sqdouble
in (s::STATE,n::integer)
out (s'::STATE,
     r::(integer,char))
match
(SQUARE,n) ->
 (DOUBLE,(sq n,'\n')) |
(DOUBLE,n) ->
 (SQUARE,(twice n,'\n'));
...
wire sqdouble
(sqdouble.s'
  initially SQUARE,input)
  (sqdouble.s,output);
```

```
$ hume sqdouble.hume
...
RUNNING
```

2
```
4
```
3
```
6
```
4
```
16
```
5
```
10
...
```

inc.hume



- generate integers

```
box inc
in (n::int 64)
out (r,n’::int 64)
match
 n -> (n,n+1);

wire inc
(inc.n’ initially 0)
(... , inc.n)
```

vending.hume

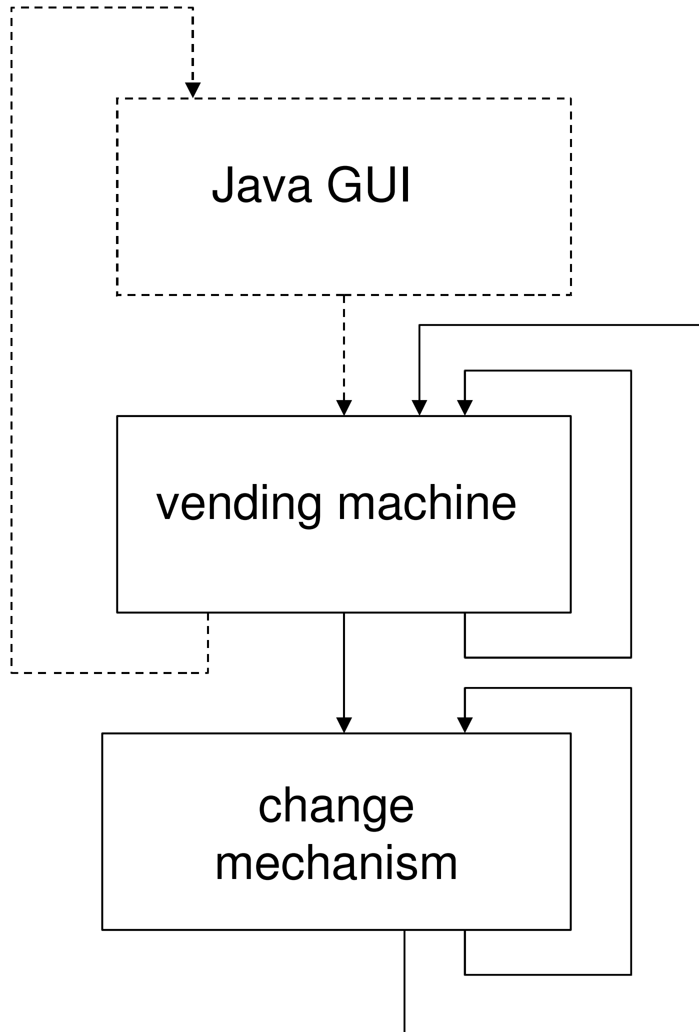
- vending machine simulation
- user actions
  1. insert money
  2. select product
  3. request & receive remaining credit as change

vending.hume

- vending machine actions

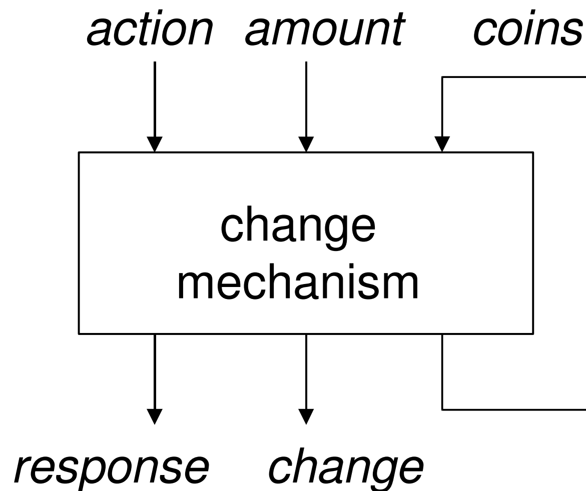
1. add money to credit
2. if enough credit & change then dispense product
3. dispense credit/change as smallest number of coins

vending.hume



- *vending machine*
  - keeps track of credit & current purchase price
  - requests change check & coin release from *change mechanism*
- *change mechanism*
  - keeps track of coins & dispenses change
- *Java GUI* talks to Hume via socket

vending.hume



- *action* & *amount* from vending machine
- *response* & *change* to vending machine
- current *coins* held on feedback

vending.hume

```
type integer = int 64;
type COINS = vector 1 .. 8 of integer;
data ACTION = ADD | CHECK | RELEASE;
data ACK = OK | FAIL;

box change
in (action::ACTION,amount::integer,coins::COINS)
out (response::ACK,change::COINS,coins'::COINS)
match
(ADD,coin,coins) -> let coins' = add coin coins 1

                    in (*,*,coins') |
(CHECK,amount,coins) ->
                    (check amount coins 1,*,coins)
(RELEASE,amount,coins) ->
  case release amount <<0,0,0,0,0,0,0,0>> coins 1 of
   (change,coins') -> (*,change,coins');
```

vending.hume

```
values = <<200,100,50,20,10,5,2,1>>;
exception BAD_COIN::integer;

add coin coins i =
 if i>MAXCOINS
 then raise BAD_COIN coin
 else
  if coin==values@i
  then update coins i (coins@i+1)
  else add coin coins (i+1);
```

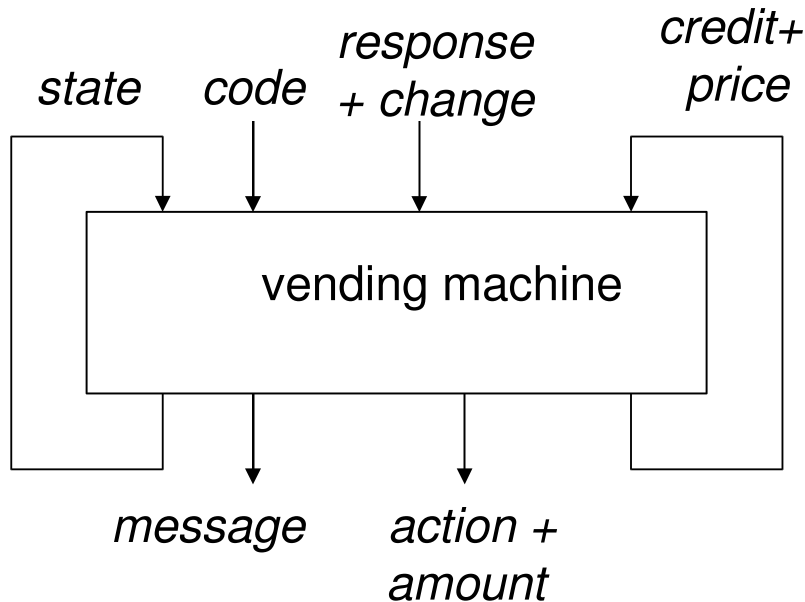
vending.hume

```
check 0 = OK;
check amount coins i =
 if i>MAXCOINS
 then FAIL
 else
  let m = amount div (values@i)
  in
   if m>coins@i
   then check (amount-values@i*coins@i) coins (i+1)
   else check (amount-values@i*m) coins (i+1);
```

vending.hume

```
release 0 change coins _ = (change,coins);
release amount change coins i =
 let m = amount div (values@i)
 in
  if m>coins@i
  then release (amount-values@i*coins@i)
               (update change i (change@i+coins@i))
               (update coins i 0) (i+1)
  else release (amount-values@i*m)
               (update change i (change@i+m))
               (update coins i (coins@i-m)) (i+1);
```

vending.hume



- *state* to indicate GUI or change mechanism interaction
- *code* from GUI for cash/purchase/ change
- *action* + *amount* to change mechanism
- *response* + *change* from change mech.
- *message* to GUI

vending.hume

```
data STATE = INPUT | CHECKING | CHANGE;

box vending
in (s::STATE,code::(char,integer),credit::integer,
    price::integer,response::ACK,change::COINS)
out (s'::STATE,m::string,credit'::integer,
     price'::integer,action::ACTION,amount::integer)
match
 (INPUT,('c',0),credit,*,*,*) ->
  (CHANGE,"GETTING CHANGE\n",0,*,RELEASE,credit) |
 (INPUT,('c',coin),credit,*,*,*) ->
  (INPUT, showCredit (credit+coin),credit+coin,*,ADD,coin)
 (INPUT,('d',amount),credit,*,*,*) ->
  if amount>credit
  then (INPUT,"NOT ENOUGH CREDIT\n"++showCredit credit,
        credit,*,*,*)
  else (CHECKING,"CHECKING CHANGE\n",
        credit,amount,CHECK,amount) | ...
```

vending.hume

```
(CHECKING,*,credit,price,OK,*) ->
 (INPUT,"PURCHASE MADE\n"++showCredit (credit-price),
  credit-price,*,*,*)
(CHECKING,*,credit,_,FAIL,*) ->
 (INPUT,"NO CHANGE\n"++showCredit credit,credit,*,*,*) |
(CHANGE,*,credit,*,*,change) ->
 (INPUT,showChange change 1++showCredit credit,
   credit,*,*,*);

showChange c i =
 if i>MAXCOINS
 then "\n"
 else
  if c@i==0
   then showChange c (i+1)
   else (c@i) as string++"*"++(values@i) as string++"p; "++
        showChange c (i+1);

   showCredit credit = "CREDIT "++credit as string++"\n";
```

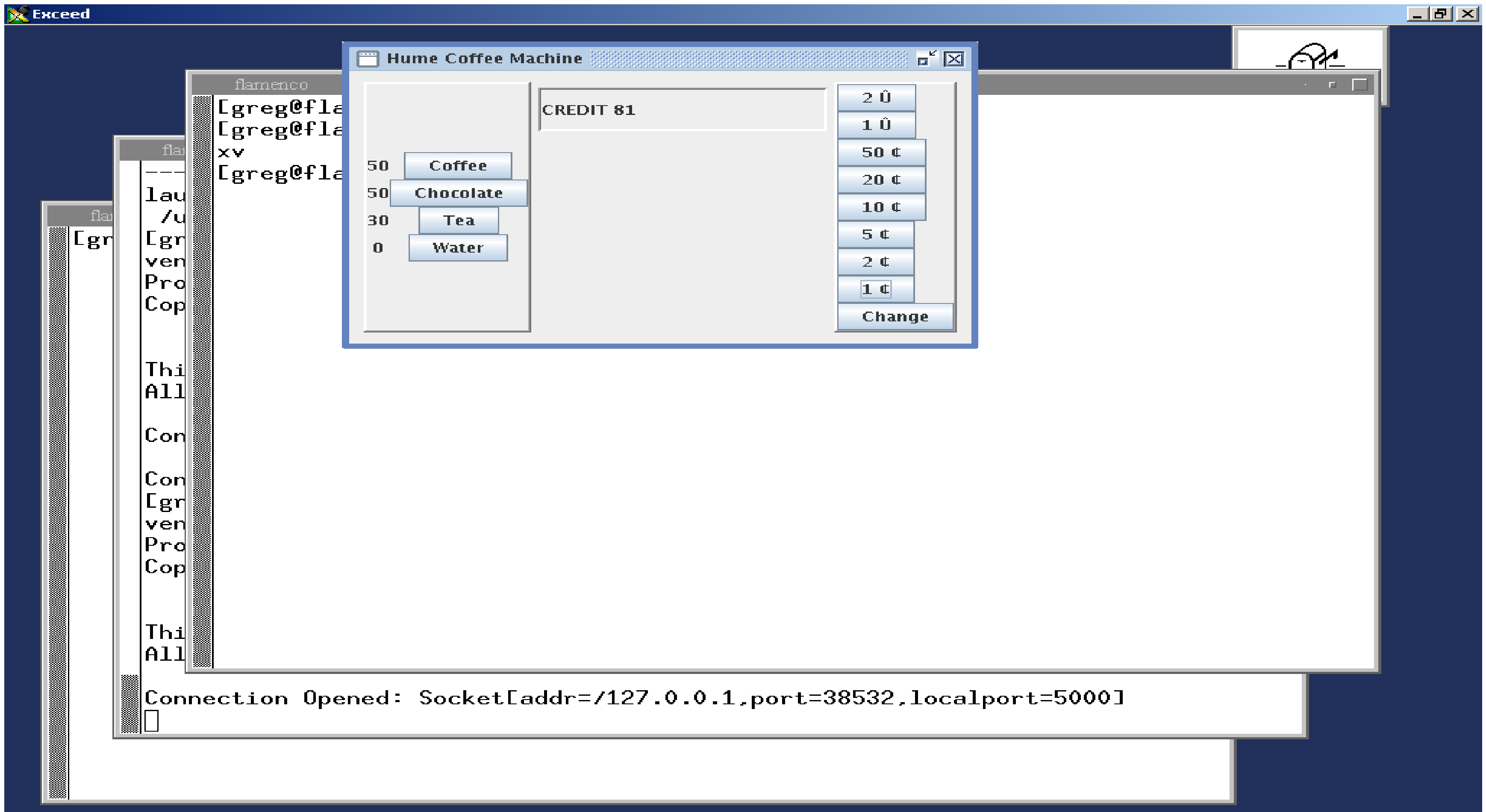
vending.hume

```
stream input from "data://localhost:5000";
stream output to "data://localhost:5000";

wire change
(vending.action,vending.amount,
 change.coins' initially <<0,0,0,0,0,0,0,0>>
(vending.response,vending.change,change.coins);

wire vending
(vending.s' initially INPUT,input,
 vending.credit' initially 0,vending.price',
 change.response,change.change
(vending.s,output,vending.credit,vending.price,
 change.action,change.amount);
```

vending.hume



Transformation and costing

- wish to cost/predict Hume program resource use
- static analysis of arbitrary recursion is undecidable
- static analysis of restricted recursion e.g. primitive, is complex and imprecise
- static analysis of purely finite state constructs is strongly bounded

Transformation and costing

- can transform many recursive forms to finite state
- typically, change bounded recursion over arbitrary size data structure to box iteration over finite input
- replace accumulator variable in recursion with feedback wire

Linear recursion to iteration

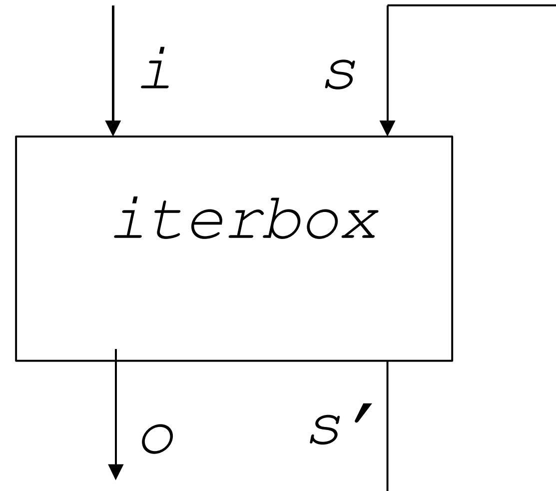
```
linrec f g h a =
 if f a
 then return g a
 else return linrec f g h (h a) <==>

iter a =
 while true do
  if f a
   then return g a
   else a := h a
```

Iteration to Looping Box

```
iter x =
  while not (f x)
   x := h x
  return g x <==>

box iterbox
in  (i::t1, s::t1)
out (o::t3, s'::t1)
match
 (*, x) -> if f x
           then (g x, *)
           else (*, h x) |
 (x, *) -> (*, x);
```

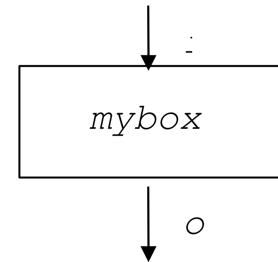


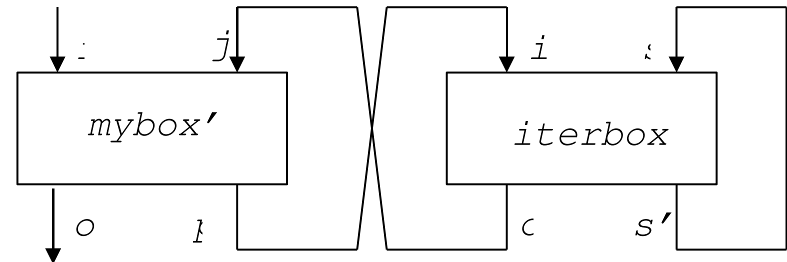
Expanding the Box

```
box mybox
in  (i::t1)
out (o::t2)
match
  (x) -> (…linrec x…);

       <==>

box mybox'
in  (i::t1, j::t2)
out (o::t2, p::t1)
match
  (*, r) -> (r…, *) |
  (x, *) -> (*, …x);
```





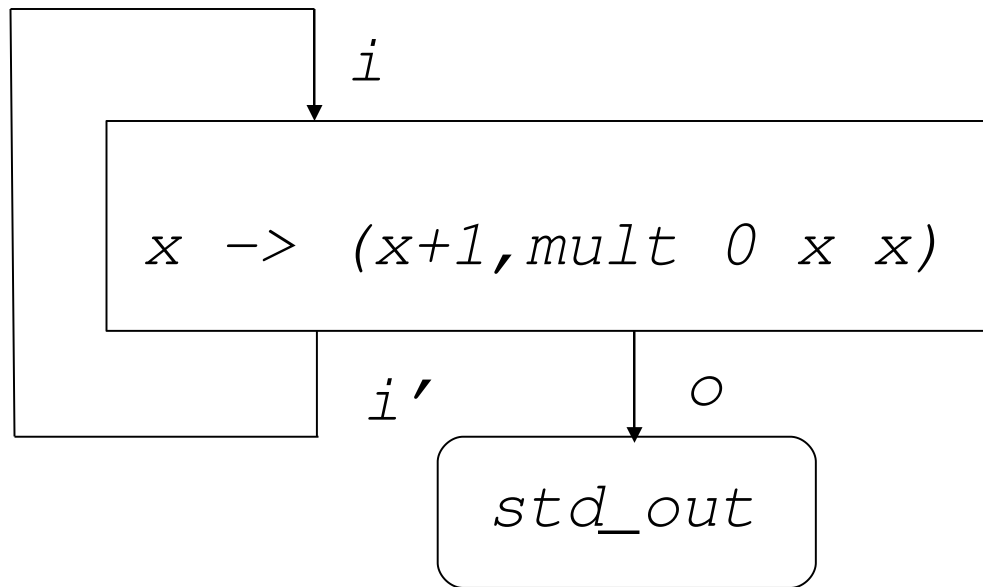
Example: Multiplication 1

```
mult r x 0 = r;
mult r x y = mult (r+x) x (y-1);

e.g.      mult 0 2 3 ==>
     mult 2 2 2 ==>
     mult 4 2 1 ==>
     mult 6 2 0 ==>
     6
```

Example: multiplication 2

- use to generate squares from 0



Example: multiplication 3

```
type integer = int 64;

box mult1
in (i::(integer,integer))
out (i'::integer,o::string)
match
 x -> (x+1,mult 0 x x as string++"\n");

stream output to "std_out";

wire mult1 (mult1.i' initially 0) (mult1.i,output);
```

Example: Multiplication 4

```
mult r x 0 = r;
mult r x y = mult (r+x) x (y-1); ==>

mult (r,x,y) =
 if y=0
 then return r
 else return mult (r+x,x,y-1)

f (r,x,y) = y=0
g (r,x,y) = r
h (r,x,y) = (r+x,x,y-1)
```

Example: Multiplication 5

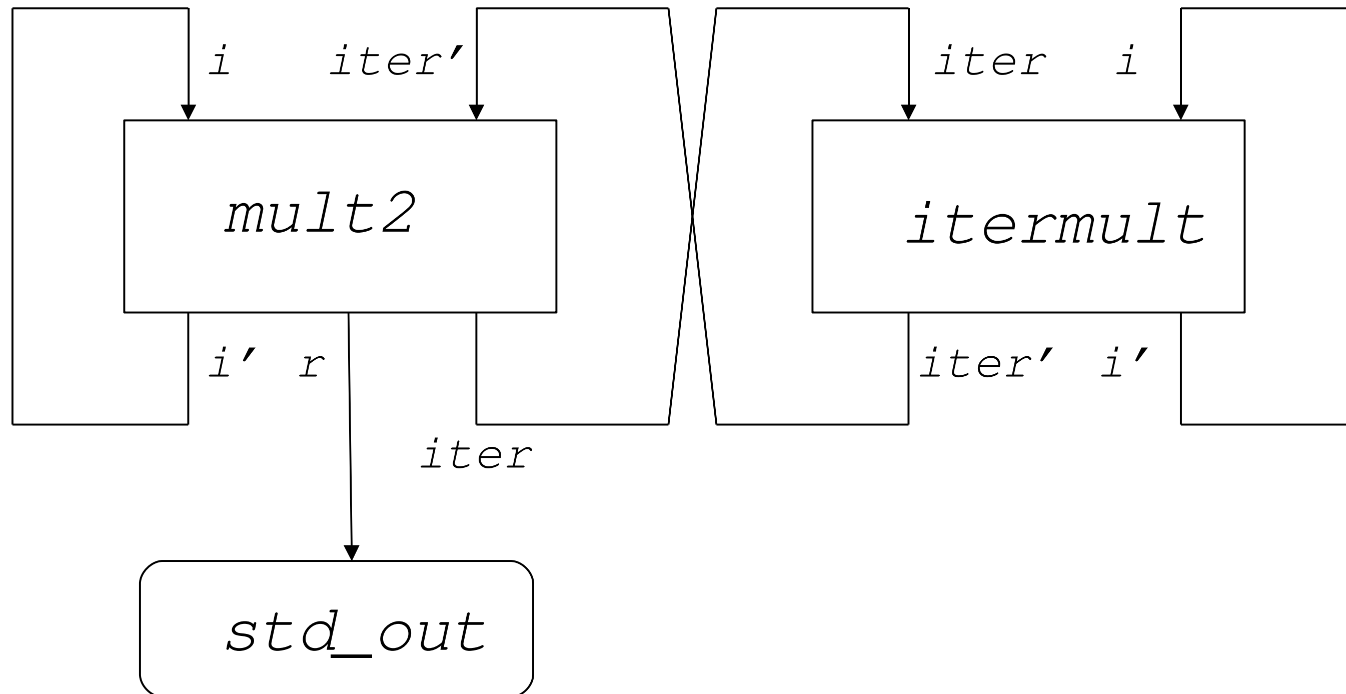
- recursion transforms to:

```
mult (r,x,y) =
 while true do
  if y=0
  then return r
  else (r,x,y) := (r+x,x,y-1)
```

Example: Multiplication 6

```
box itermult
in (i::(integer,integer,integer),
    iter::(integer,integer,integer))
out (iter'::(integer,integer,integer),r::integer)
match
 ((r,x,y),*) -> ((r,x,y),*) |
 (*,(r,x,y)) -> if y==0
                then (*,r)
                else ((r+x,x,y-1),*);
```

Example: Multiplication 7



Example: Multiplication 8

```
box mult2
in (i::integer,iter'::integer)
out (i'::integer,iter::(integer,integer,integer),r::string)
match
(x,r) -> (x+1,(0,x,x),r as string++"\n");

stream output to "std_out";

wire mult2 (mult2.i' initially 0,itermult.r initially 0)
           (mult2.i,itermult.i,output);

wire itermult (mult2.iter,itermult.iter')
              (itermult.iter,mult2.iter');
```

Exercise

1. write a Hume function to return
   1. $e(N) = \Sigma\ 1/i! : 1 <= i <= N$
2. write a Hume program to display successive values of *e* for *N>0*
3. transform the program from 2. to use box iteration for *e(N)*
4. transform the program from 3. to use box iteration for *i!* in *e(N)*