Assigning to a Variable

What is the result of this program?

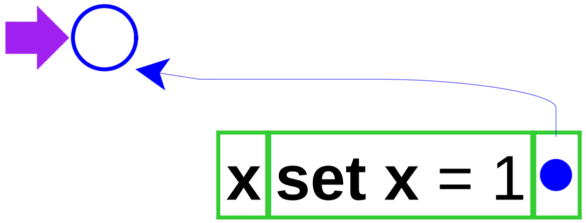
```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y  }
```

Is it 0 or 1?

Assigning to a Variable

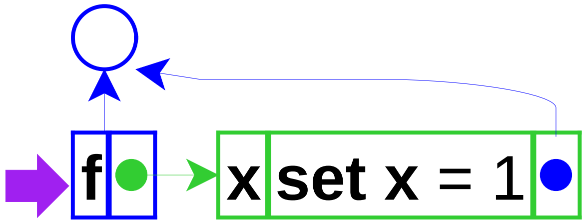
let f = **proc**(**x**) **set x** = 1
  in let y = 0
    in { (**f y**);
      y }

Assigning to a Variable



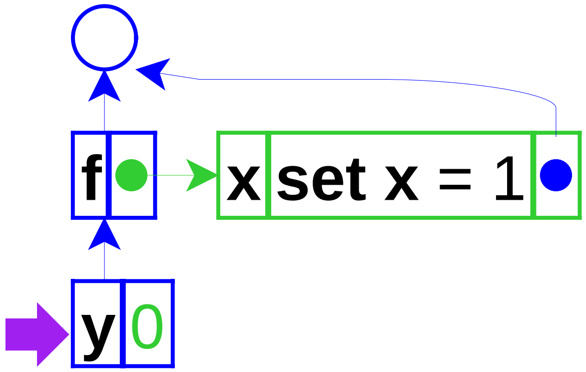

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y  }
```

Assigning to a Variable



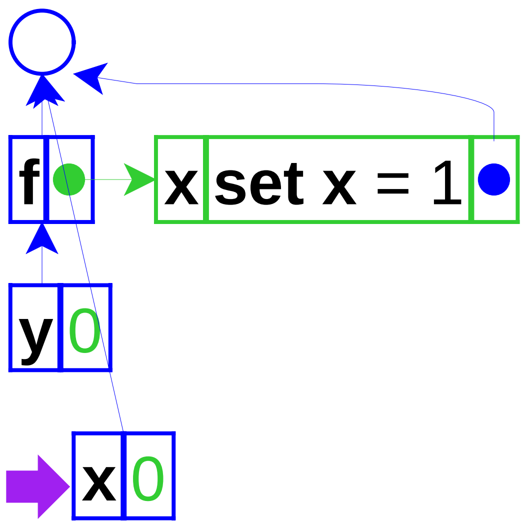

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y  }
```

Assigning to a Variable



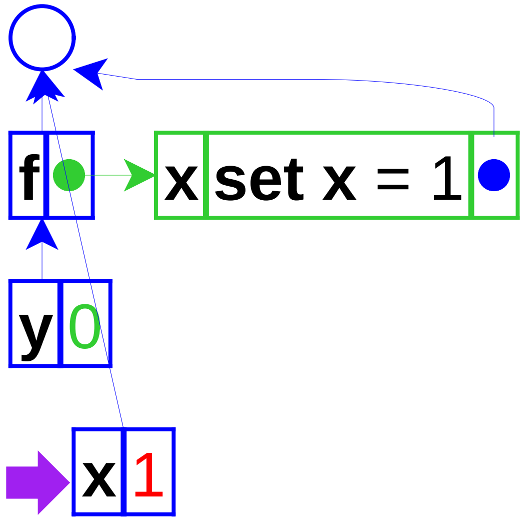
let f = **proc**(**x**) **set x** = 1
in let y = 0
in { **(f y)**;
y }

Assigning to a Variable



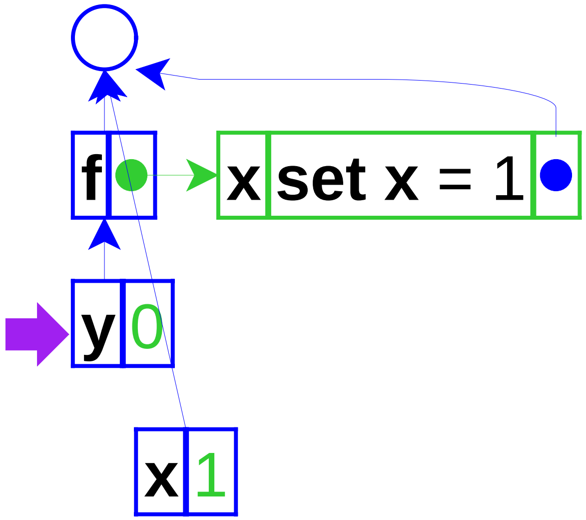

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y  }
```

Assigning to a Variable



let f = **proc**(**x**) **set x** = 1
in let y = 0
in { (**f y**);
y }

Assigning to a Variable



So the answer is 0

let f = **proc**(**x**) **set x** = 1
in let y = 0
in { (**f y**);
y }

Variables in C++

```
void f(int x) {
  x = 1;
}

int main() {
  int y = 0;
  f(y);
  return y;
}
```

The result above is 0, too

Variables in C++

```
void f(int& x) {
  x = 1;
}

int main() {
  int y = 0;
  f(y);
  return y;
}
```

But the result above is 1

Variables in C++

```
void f(int& x) {
  x = 1;
}

int main() {
  int y = 0;
  f(y);
  return y;
}
```

This example shows ***call-by-reference***.

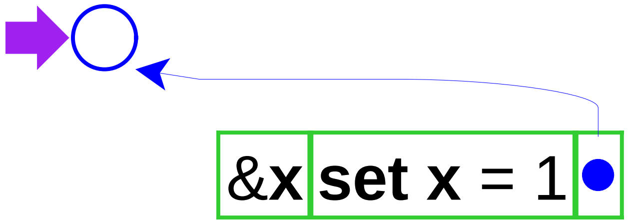
The previous example showed ***call-by-value***.

Assignment and Call-by-Reference

Adding call-by-reference parameters to our language

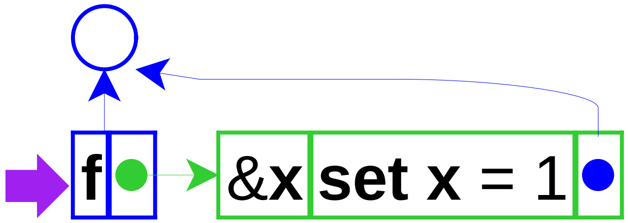
```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y  }
```

Assignment and Call-by-Reference



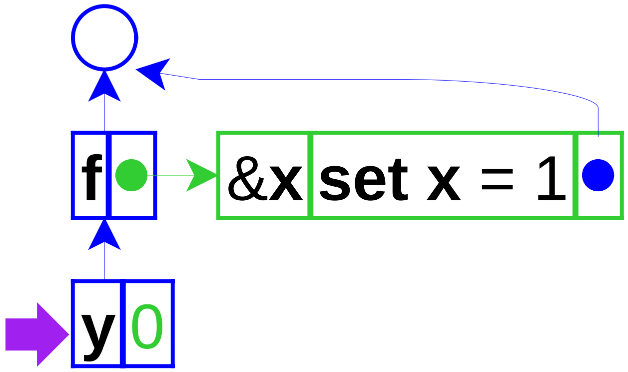

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y  }
```

Assignment and Call-by-Reference



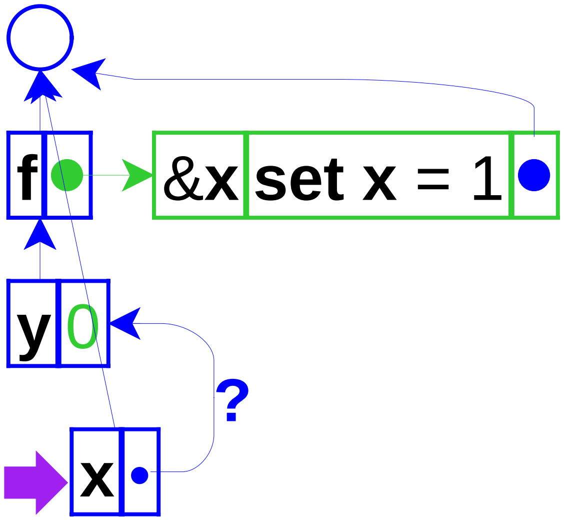

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y }
```

Assignment and Call-by-Reference



let f = **proc**(&**x**) **set x** = 1
  in let y = 0
    in { **(f y)**;
      y }

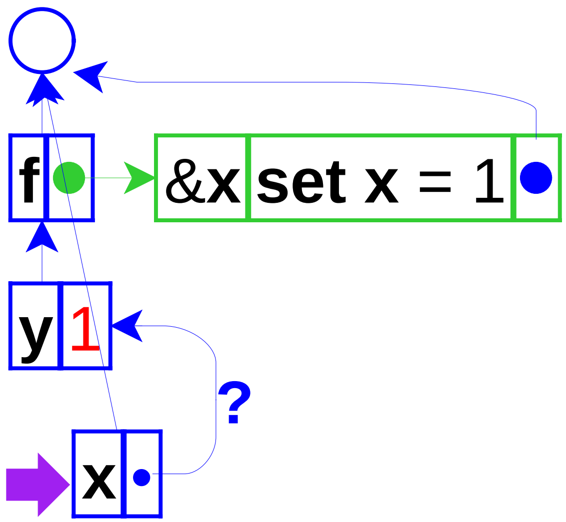
Assignment and Call-by-Reference



The pointer from one environment frame to another is questionable, because frames are supposed to point to values

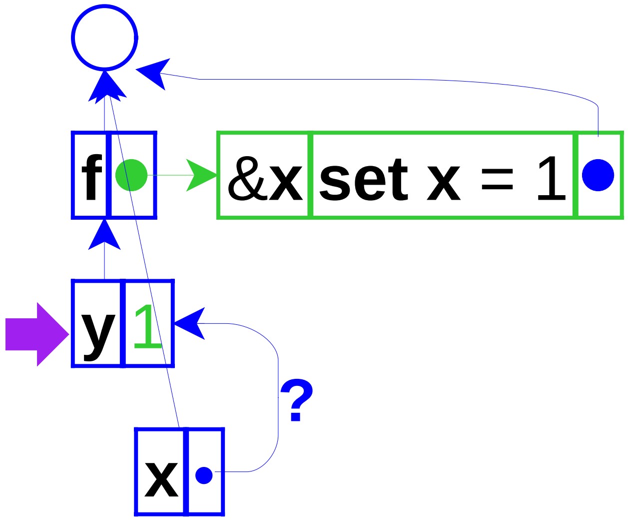
let f = **proc**(&**x**) **set x** = 1
in let y = 0
in { (**f y**);
y }

Assignment and Call-by-Reference



let f = **proc**(&**x**) **set x** = 1
in let y = 0
in { (**f y**);
y }

Assignment and Call-by-Reference




```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y }
```

Interpreter Changes

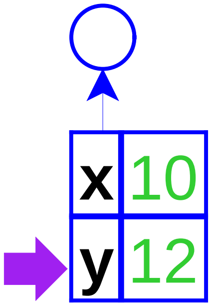
Same as before:

- Expressed values: Number + Proc
- Denoted values: Ref(Expressed Value)

The difference is that application doesn't always create a new location for a new variable binding

=> Separate *location* creation from *environment* extension

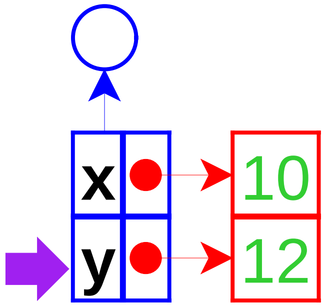
Assignment and Call-by-Reference



The old way

let x = 10
    y = 12
  in +(**x**,**y**)

Assignment and Call-by-Reference



The new way

let x = 10

y = 12

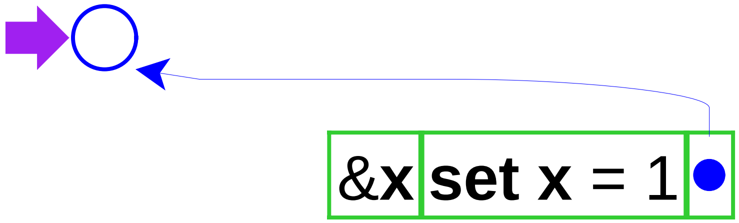
in +(**x**,**y**)

Call-by-Reference

Do the previous evaluation the new way...

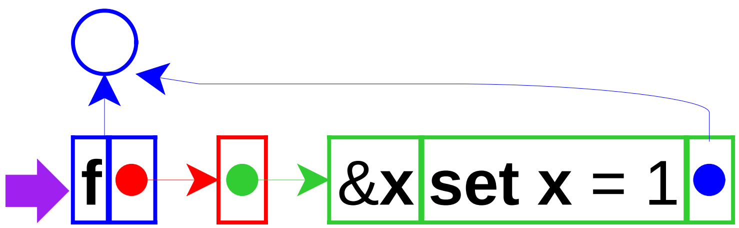
let f = **proc**(&**x**) **set x** = 1
  in let y = 0
    in { (**f y**);
      y }

Call-by-Reference



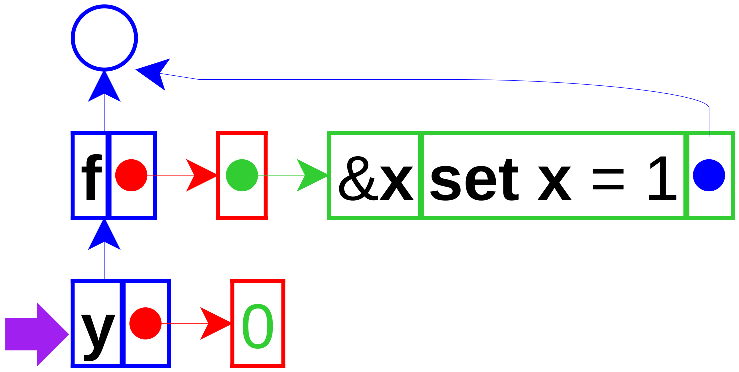

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y }
```

Call-by-Reference



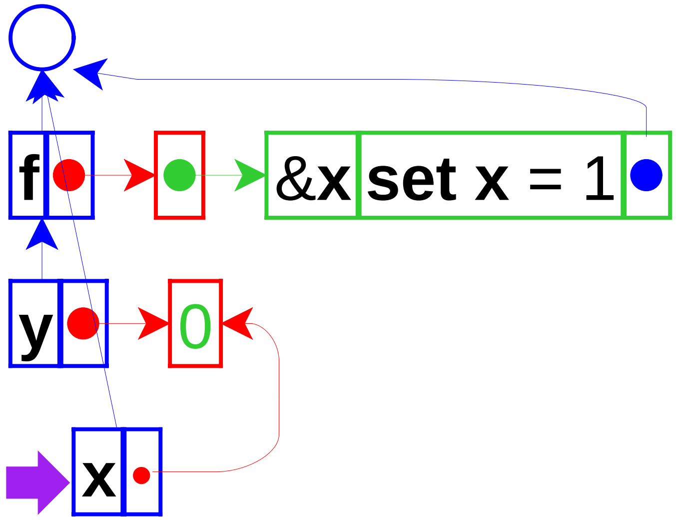
let f = **proc**(&**x**) **set x** = 1
   in let y = 0
      in { (**f y**);
         y }

Call-by-Reference



let f = **proc**(&**x**) **set x** = 1
  in let y = 0
    in { **(f y)**;
      y }

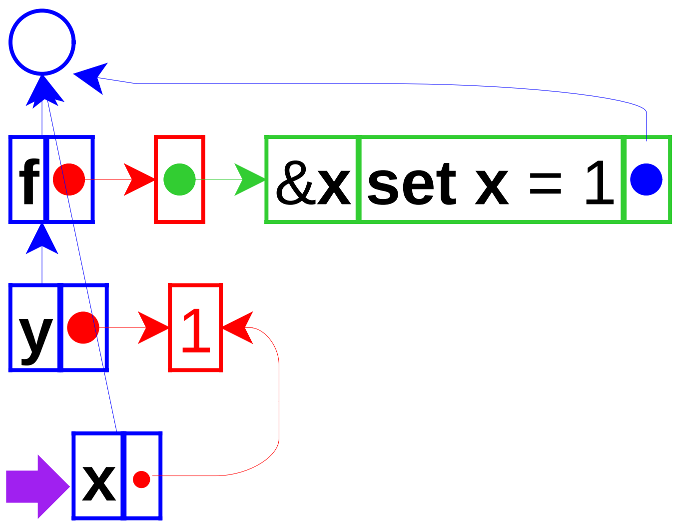
Call-by-Reference



This time, the new environment frame points to a location box, which is consistent with other frames

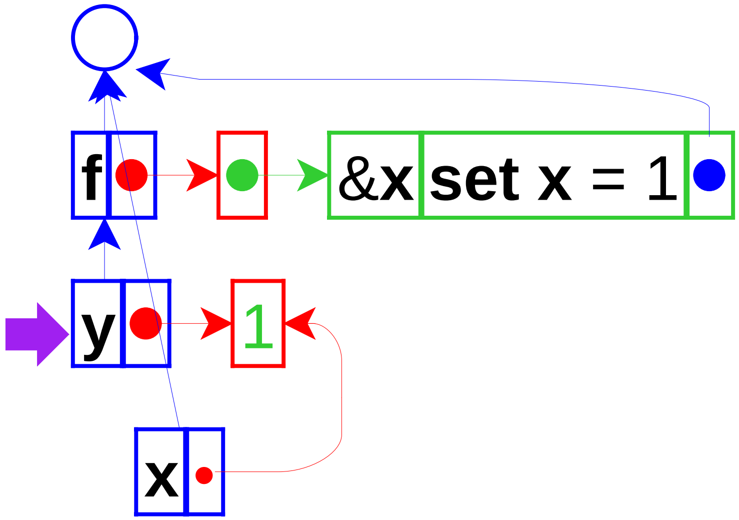
let f = **proc**(&**x**) **set x** = 1
  in let y = 0
    in { (**f y**);
      y }

Call-by-Reference



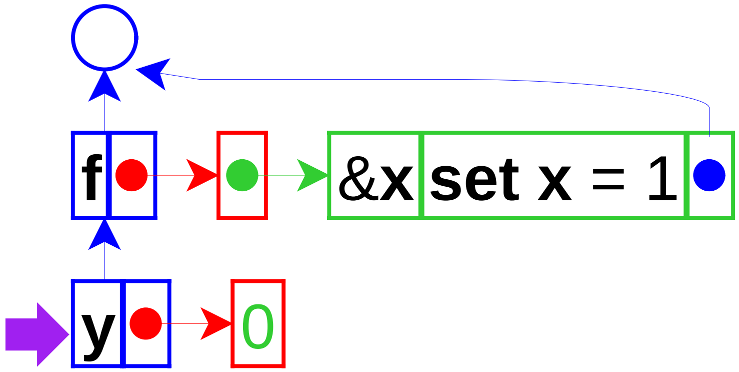
let f = **proc**(&**x**) **set x** = 1
  in let y = 0
    in { (**f y**);
      y }

Call-by-Reference



let f = **proc**(&**x**) **set x** = 1
  in let y = 0
    in { (**f y**);
      y }

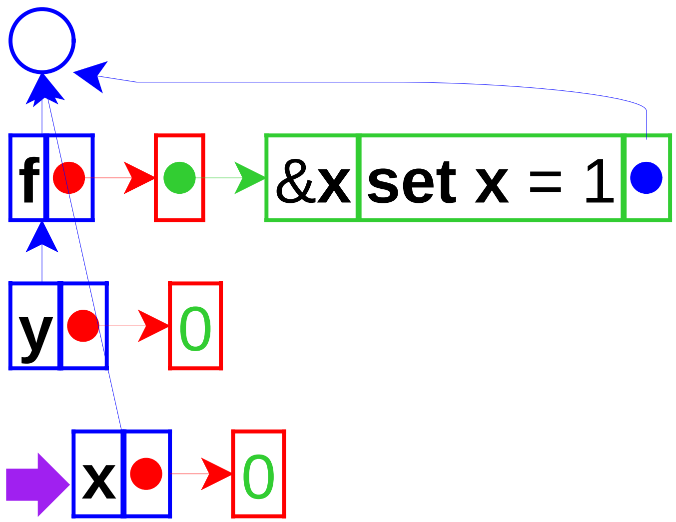
Call-by-Reference with Non-variables



If call-by-reference argument is not a variable...

let f = **proc**(&**x**) **set x** = 1
  in let y = 0
    in { (**f** 0);
      y }

Call-by-Reference with Non-variables



... create a location

let f = **proc**(&**x**) **set x** = 1
in let y = 0
in { (**f** 0);
y }

Interpreter Changes

- Add call-by-reference arguments (indicated by &)
- New **var** datatype, with **`cbv-var`** and **`cbr-var`** variants
- Create explicit locations for variables

```
location : expval -> location
location-val : location -> expval
location-set! : location expval -> void
```

- Change variable lookup to de-reference locations
- Change **set** to work on locations
- Add **`eval-fun-rands`** and change **`apply-proc`**

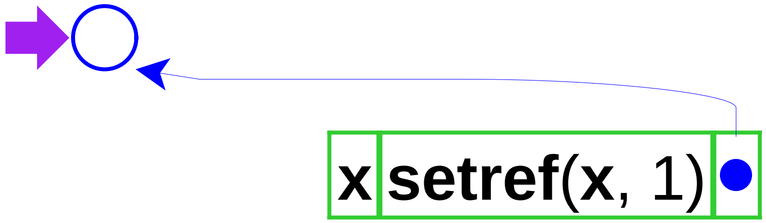
& versus * in C++

```
void f(int* x) {
  *x = 1;
}

int main() {
  int y = 0;
  f(&y);
  return y;
}
```

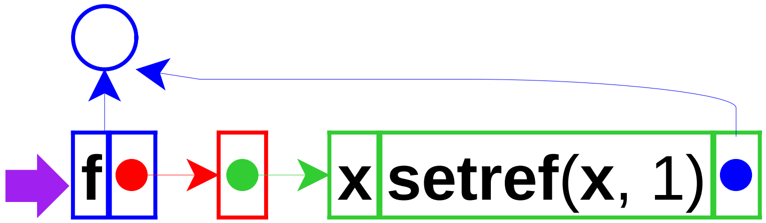
- This is back to ***call-by-value***, but with a reference as a value
- To study this form of call, we can add explicit references to our language, too

Call-by-Value with References



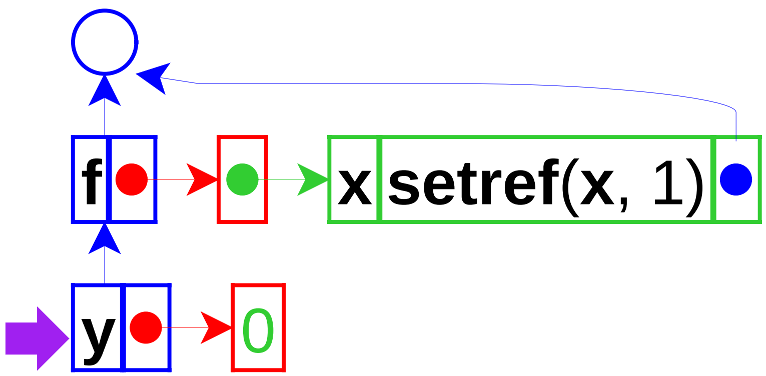

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y  }
```

Call-by-Value with References



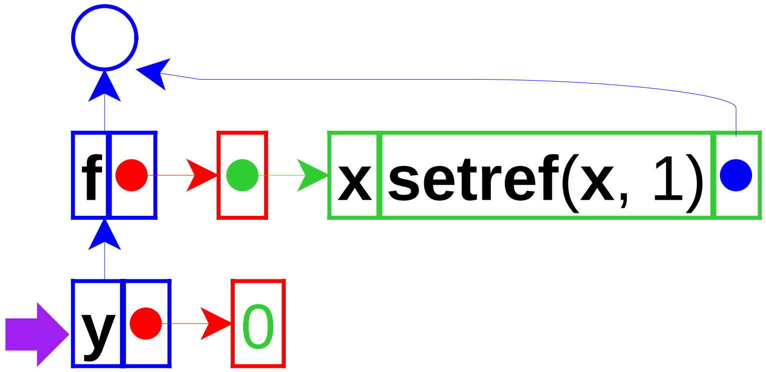
let f = **proc**(**x**) **setref**(**x**, 1)
 in let y = 0
    in { (**f ref**(**y**));
        y }

Call-by-Value with References



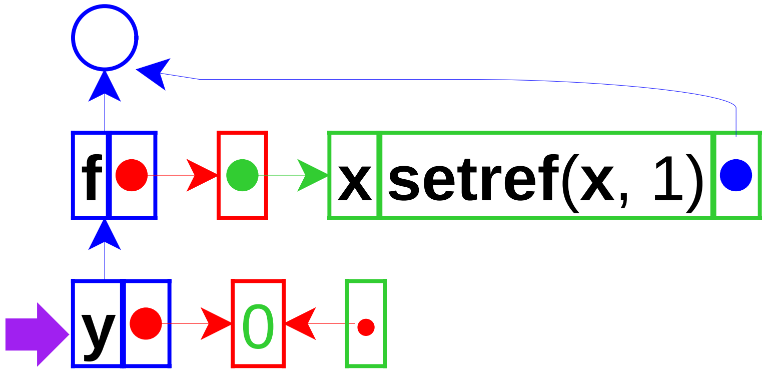
let f = **proc**(**x**) **setref**(**x**, 1)
 in let y = 0
    in { **(f ref(y));**
       y }

Call-by-Value with References



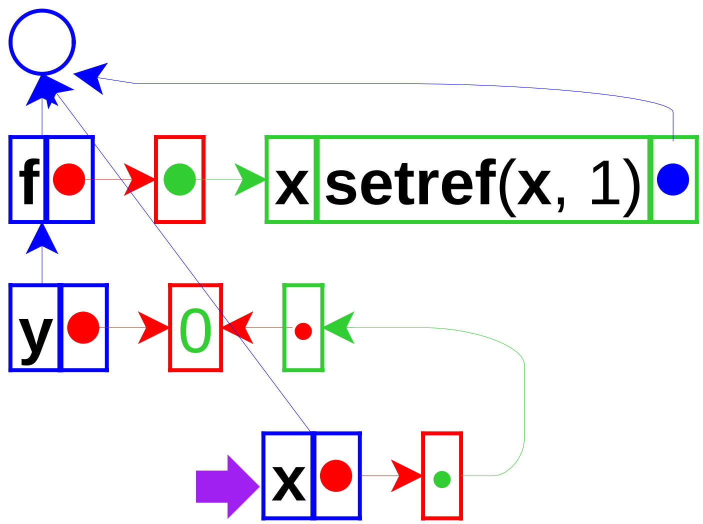

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y  }
```

Call-by-Value with References




```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y  }
```

Call-by-Value with References



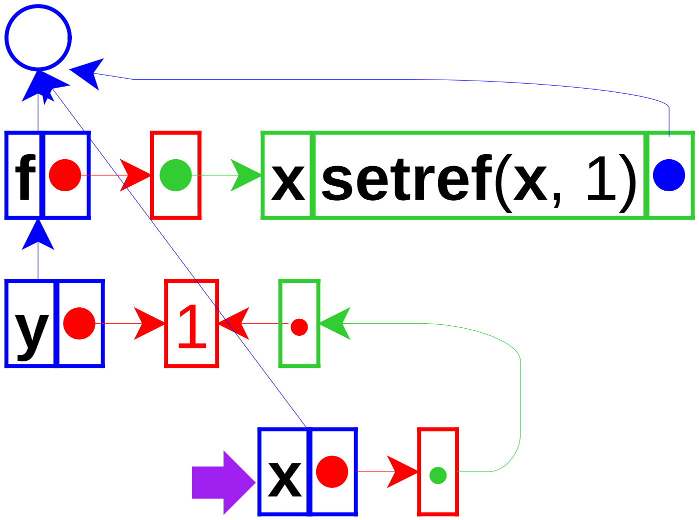
let f = **proc**(**x**) **setref(x**, 1)

in let y = 0

in { (**f ref**(**y**));

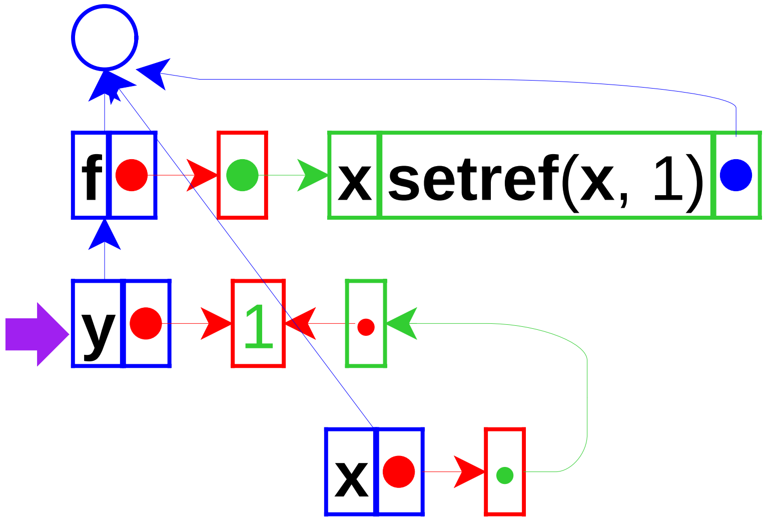
y }

Call-by-Value with References



let f = **proc**(**x**) **setref(x**, 1)
in let y = 0
in { (**f ref**(**y**));
y }

Call-by-Value with References



let f = **proc**(**x**) **setref**(**x**, 1)
 in let y = 0
    in { (**f ref**(**y**));
        y }

Interpreter Changes for References

Revised language:

- Expressed vals: Number + Proc + Ref(Expressed Val)
- Denoted vals: Ref(Expressed Val)

Interpreter changes:

- Add reference values
- Add **`ref`** form and **`setref`** primitive

Lazy Evaluation of Function Arguments

let f = **proc**(**x**)0
in (**f** +(1,+(2,+(3,+(4,+(5,6))))))

The computed 21 is never used.

What if we were *lazy* about computing function arguments (in case they aren't used)?

Lazy Evaluation of Function Arguments

One way to laziness:

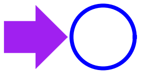
let f = **proc**(**xthunk**)0
in (**f proc**()+(1,+(2,+(3,+(4,+(5,6))))))

let f = **proc**(**xthunk**)-((**xthunk**), 7)
in (**f proc**()+(1,+(2,+(3,+(4,+(5,6))))))

By using **proc** to delay evaluation, we can avoid unnecessary computation.

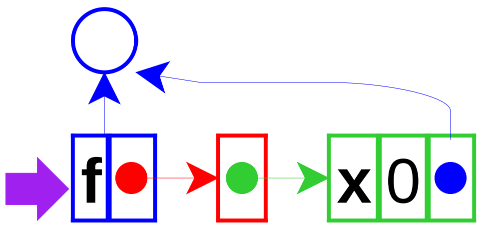
How about making the language compute function arguments lazily in *all* applications?

Evaluation with Lazy Arguments



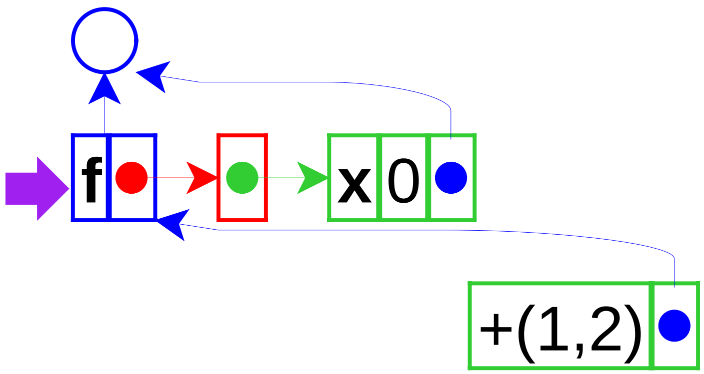
let f = **proc**(**x**)0
  in (**f** +(1,2))

Evaluation with Lazy Arguments



let f = **proc**(**x**)0
  in (f +(1,2))

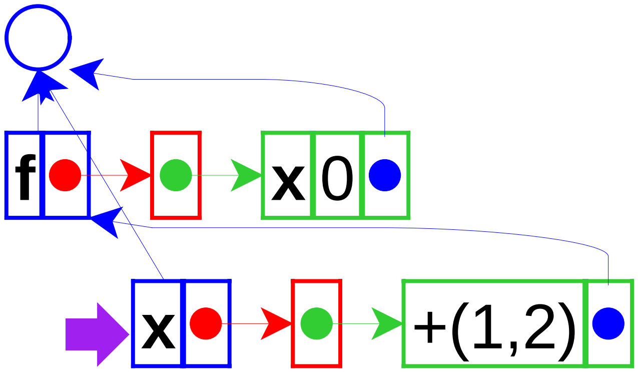
Evaluation with Lazy Arguments



Application creates a new kind of green box, with two slots: a ***thunk***

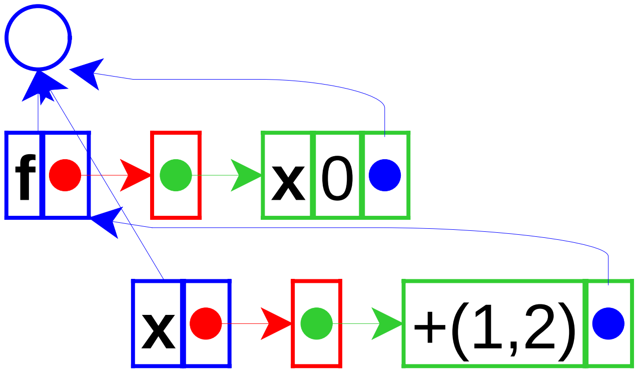
let f = **proc**(**x**)0
  in (**f** +(1,2))

Evaluation with Lazy Arguments



let f = **proc**(**x**)0
in (**f** +(1,2))

Evaluation with Lazy Arguments



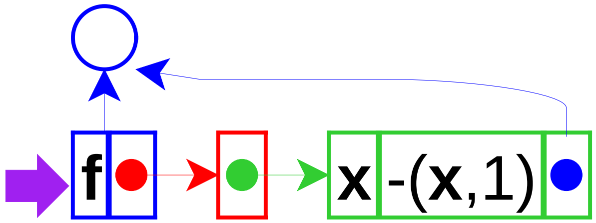
The result is 0

let f = **proc**(**x**)0
in (**f** +(1,2))

Evaluation with Lazy Arguments

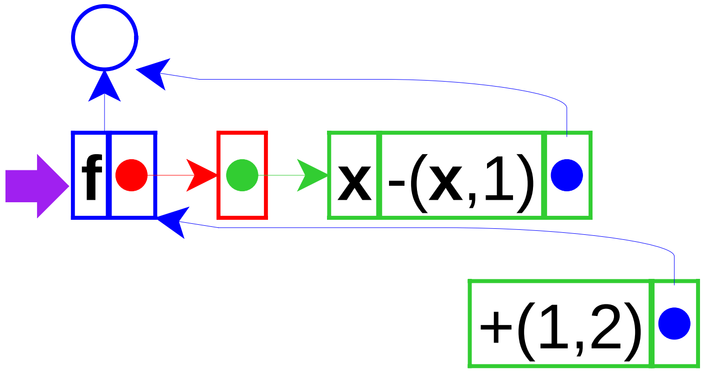
let f = **proc**(**x**)-(**x**,1)
  in (**f** +(1,2))

Evaluation with Lazy Arguments



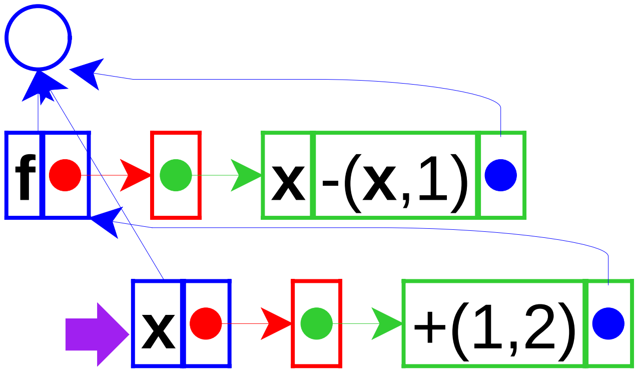
let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

Evaluation with Lazy Arguments



let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

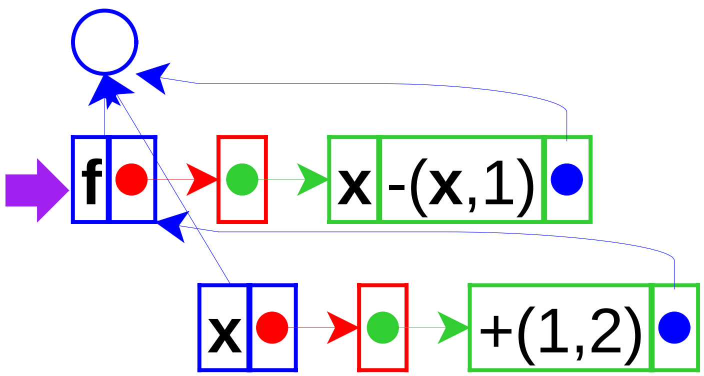
Evaluation with Lazy Arguments



lookup of **x**...

let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

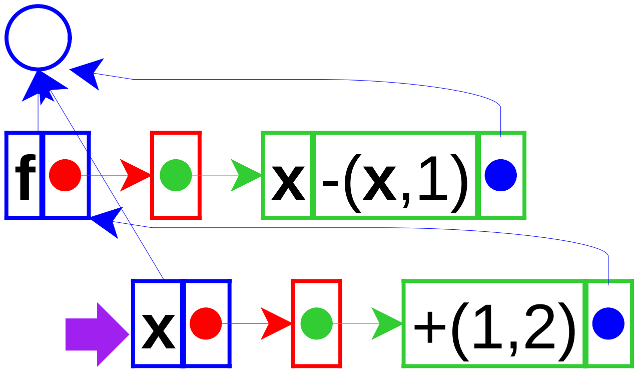
Evaluation with Lazy Arguments



... forces evaluation of the thunk

let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

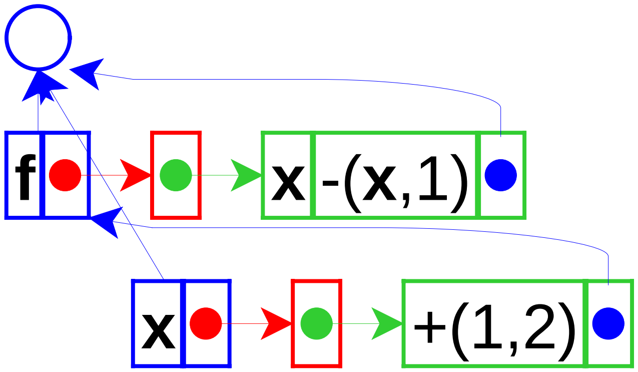
Evaluation with Lazy Arguments



so 3 is the value of **x**

let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

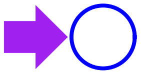
Evaluation with Lazy Arguments



The result is 2

let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

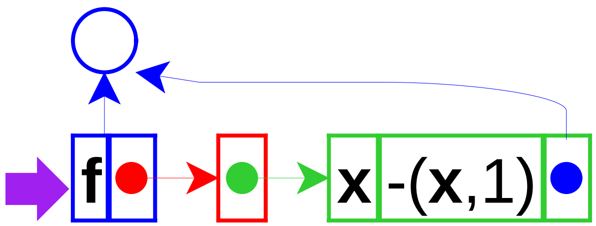
Evaluation with Lazy Arguments



Lazy expression that needs its environment...

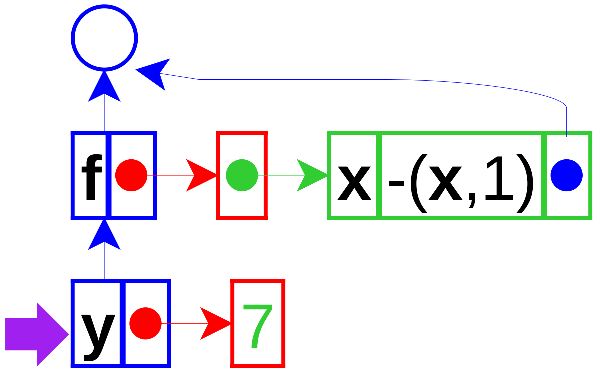
let f = **proc**(**x**)-(**x**,1)
  in let y = 7
    in (**f** +(1,**y**))

Evaluation with Lazy Arguments



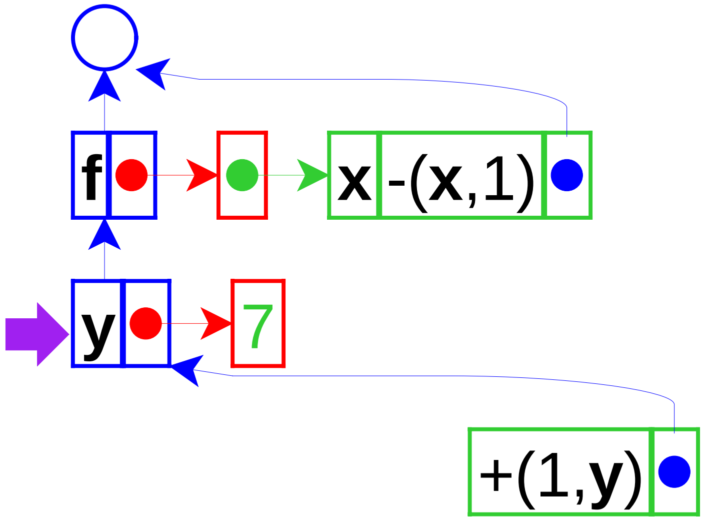
let f = **proc**(**x**)-(**x**,1)
in let y = 7
in (**f** +(1,**y**))

Evaluation with Lazy Arguments



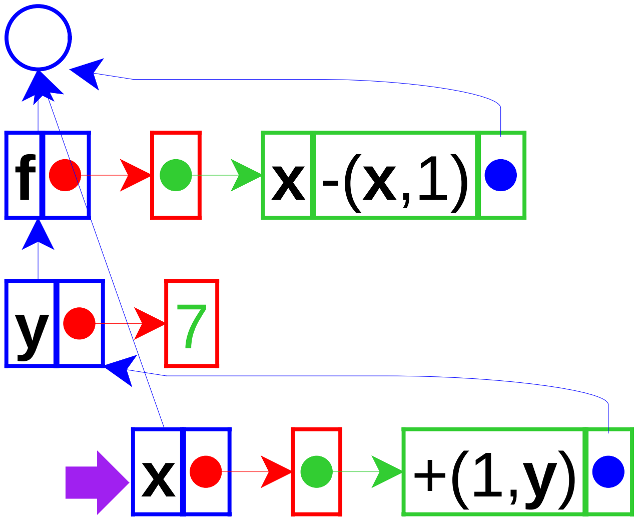
let f = **proc**(**x**)-(**x**,1)
in let y = 7
in (f +(1,**y**))

Evaluation with Lazy Arguments



let f = **proc**(**x**)-(**x**,1)
in let y = 7
in (**f** +(1,**y**))

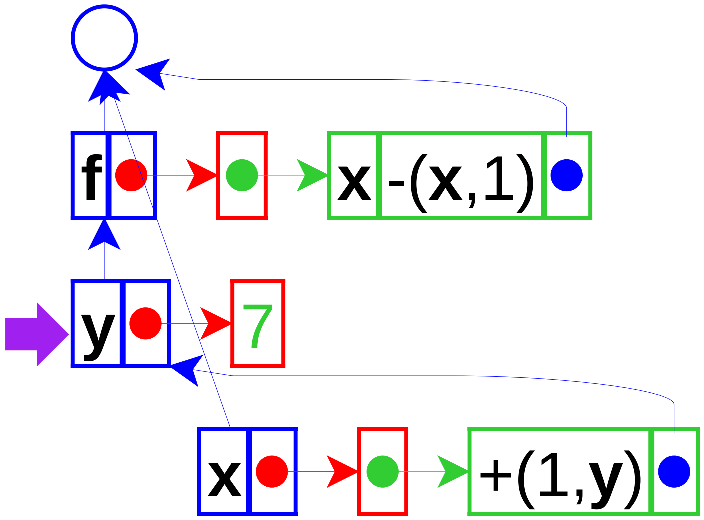
Evaluation with Lazy Arguments



Evaluation of **x** forces the thunk...

let f = **proc**(**x**)-(**x**,1)
in let y = 7
in (**f** +(1,**y**))

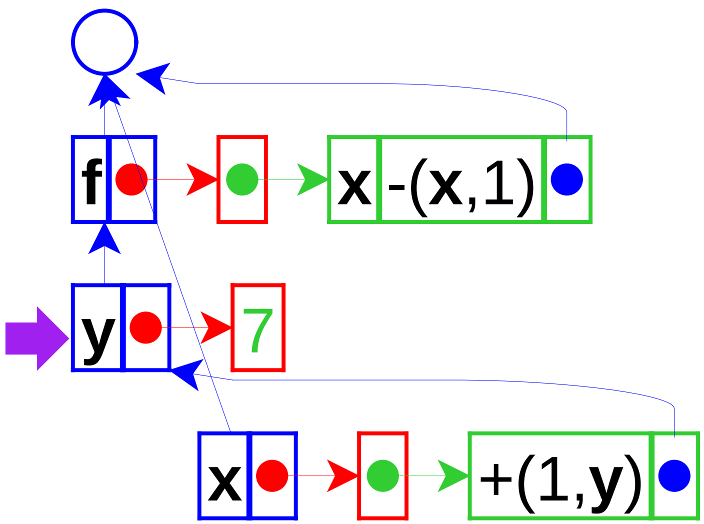
Evaluation with Lazy Arguments



Triggering evaluation with the *thunk*'s environment, not the current one

let f = **proc**(**x**)-(**x**,1)
in let y = 7
in (**f** +(1,**y**))

Evaluation with Lazy Arguments



(The result will be 7)

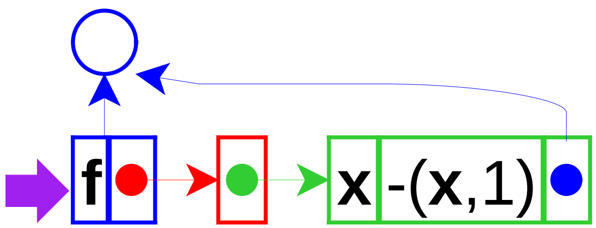
let f = **proc**(**x**)-(**x**,1)
  in let y = 7
    in (**f** +(1,**y**))

Evaluation with Lazy Arguments

What if the right-hand side for **y** is an expression, instead of a value?

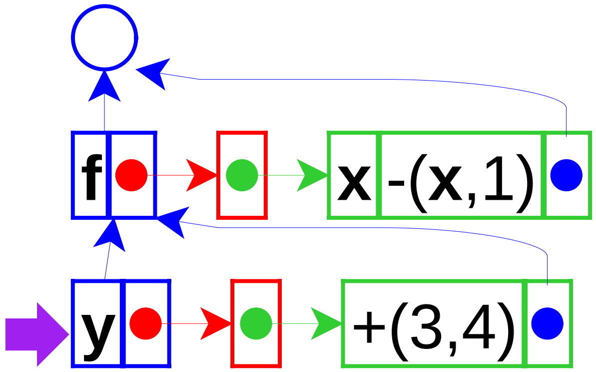
let f = **proc**(**x**)-(**x**,1)
  in let y = +(3,4)
    in (**f** +(1,**y**))

Evaluation with Lazy Arguments



let f = **proc**(**x**)-(**x**,1)
  in let y = +(3,4)
    in (**f** +(1,**y**))

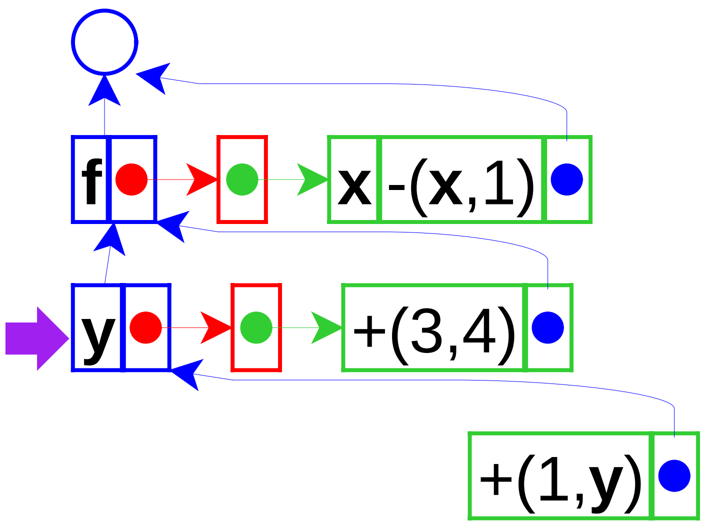
Evaluation with Lazy Arguments



Added thunk for the value of **y**

let f = **proc**(**x**)-(**x**,1)
in let y = +(3,4)
in (**f** +(1,**y**))

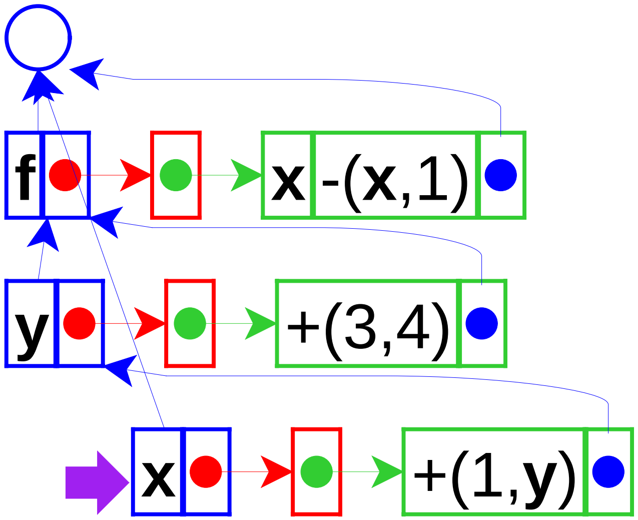
Evaluation with Lazy Arguments



Another thunk for the argument of **f**

let f = **proc**(**x**)-(**x**,1)
in let y = +(3,4)
in (**f** +(1,**y**))

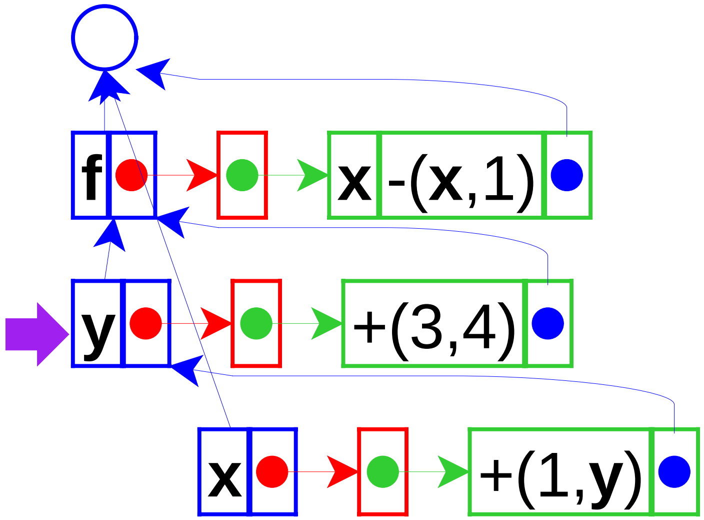
Evaluation with Lazy Arguments



Evaluation of **x** forces a thunk...

let f = **proc**(**x**)-(**x**,1)
in let y = +(3,4)
in (**f** +(1,**y**))

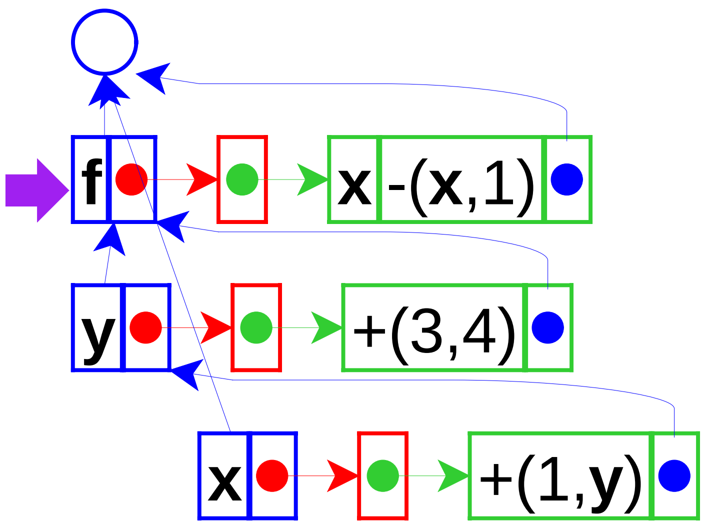
Evaluation with Lazy Arguments



which, in turn, forces another thunk...

let f = **proc**(**x**)-(**x**,1)
in let y = +(3,4)
in (**f** +(1,**y**))

Evaluation with Lazy Arguments



and so on (to get 7)

let f = **proc**(**x**)-(**x**,1)
in let y = +(3,4)
in (**f** +(1,**y**))

Implementing Lazy Evaluation

Interpreter changes:

- Change **`eval-fun-rands`** to create thunks
- Change variable lookup to force thunk evaluation

(Implement in DrScheme)

Call-by-Name and Call-by-Need

The lazy strategy we just implemented is ***call-by-name***

- Advantage: unneeded arguments are not computed
- Disadvantage: needed arguments may be computed many times

let f = **proc**(**x**)+(**x**,+(**x**,**x**))
in (**f** +(1,+(2,+(3,+(4,+(5,6))))))

Best of both worlds: ***call-by-need***

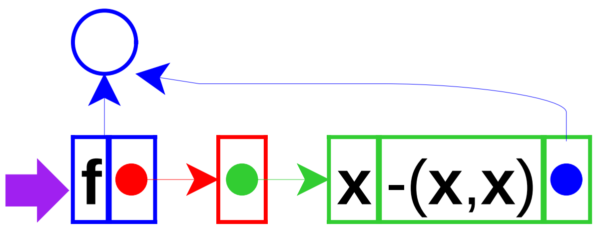
- Evaluates each lazy expression once, then remembers the result

Evaluation with Lazy Arguments

Start as before...

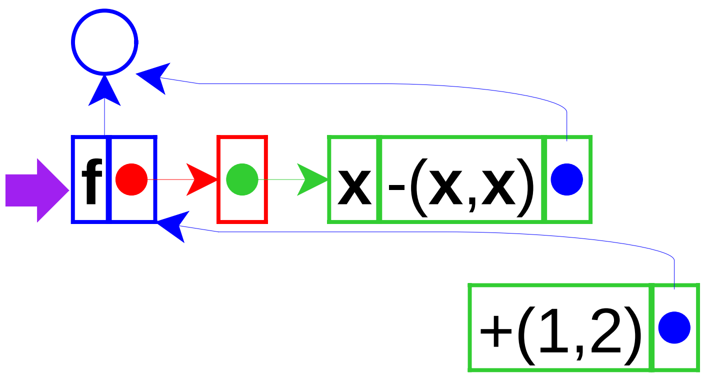
let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

Evaluation with Lazy Arguments



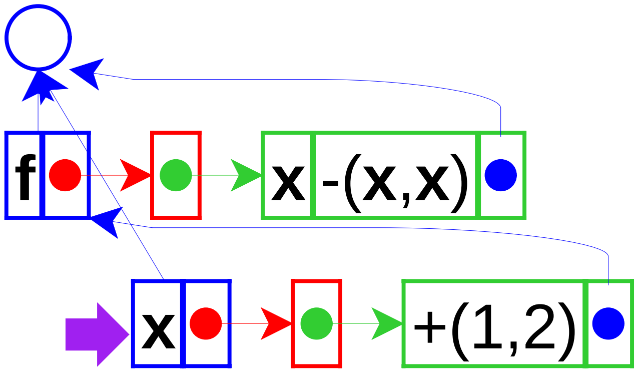
let f = **proc**(**x**)-(**x**,**x**)
  in (**f** +(1,2))

Evaluation with Lazy Arguments



let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

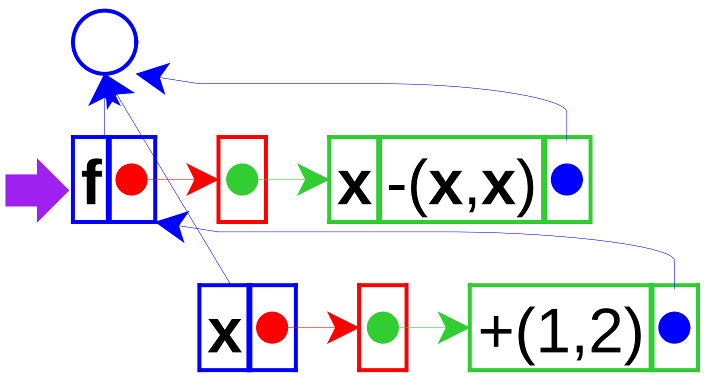
Evaluation with Lazy Arguments



lookup of **x**...

let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

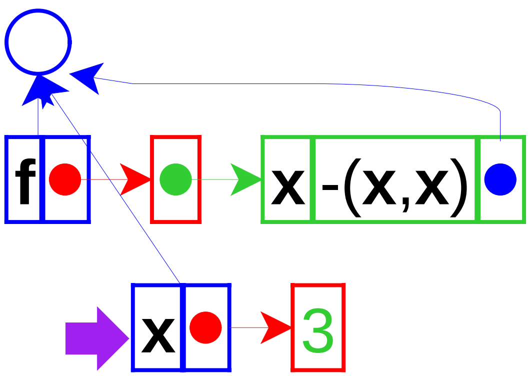
Evaluation with Lazy Arguments



... forces evaluation of the thunk to get 3

let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

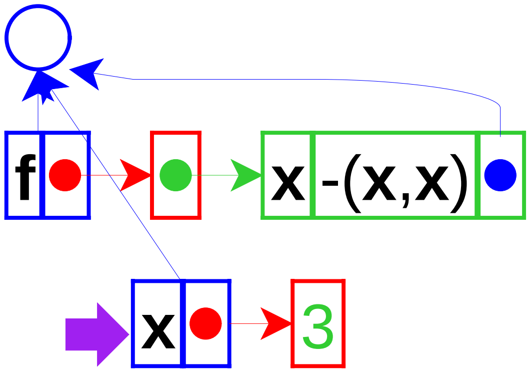
Evaluation with Lazy Arguments



so change **x** to 3 --- which is the essence of call-by-need

let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

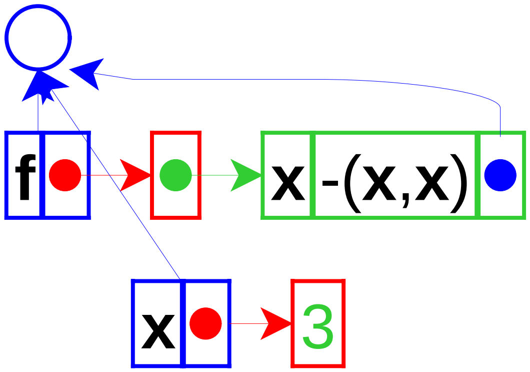
Evaluation with Lazy Arguments



lookup of **x** again gets 3

let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

Evaluation with Lazy Arguments



(The result is 0)

let f = **proc**(&**x**) **set x** = 1
in let y = 0
in { (**f y**);
y }

Implementing Call-by-Need

Interpreter changes:

- Change variable lookup to replace thunks in locations with their values

(Implement in DrScheme)

Calling Convention Terminology

- Call-by-name and call-by-need = ***lazy*** evaluation
- Call-by-value = ***eager*** evaluation

Call-by-reference can augment either...

... but the combination of reference and laziness is difficult to reason about

Popular Calling-Convention Choices

- Most languages are call-by-value
  - C, C++, Pascal, Scheme, Java, ML, Smalltalk...
- Some provide call-by-reference
  - C++, Pascal
- A few are call-by-need
  - Haskell
- Practically no languages are call-by-name

Popularity of Laziness

Why don't more languages provide lazy evaluation?

- Disadvantage: evaluation order is not obvious

```
let x = 0  f = ...
  in let y = set x=1
         z = set x=2
     in { (f y z) ; x }
```

Popularity of Laziness

Why do some languages provide lazy evaluation?

- Evaluation order does not matter if the language has no **set** form
- Such languages are called ***purely functional***
  - Note: call-by-reference is meaningless in a purely functional language
- A language with **set** can be called ***imperative***

Laziness and Eagerness

Even in a purely functional language, lazy and eager evaluation can produce different results

let f = **proc**(**x**)0
in (**f** *[loop forever]*)

- Eager answer: none
- Lazy answer: 0

Summary

- Call-by-reference
  - split location from environment representation
  - handle function arguments/variables specially
- Call-by-name
  - thunk all argument expressions
  - modify variable lookup to evaluate thunks
- Call-by-need
  - revise variable lookup to install computed thunk result