```
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)?

Manual laziness:

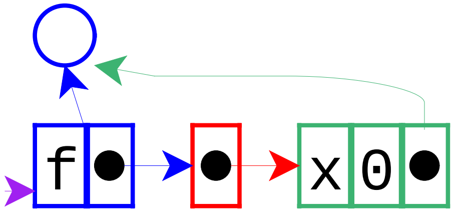
```
let f = proc(xthunk)0
 in (f proc()+(1,+(2,+(3,+(4,+(5,6))))))

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

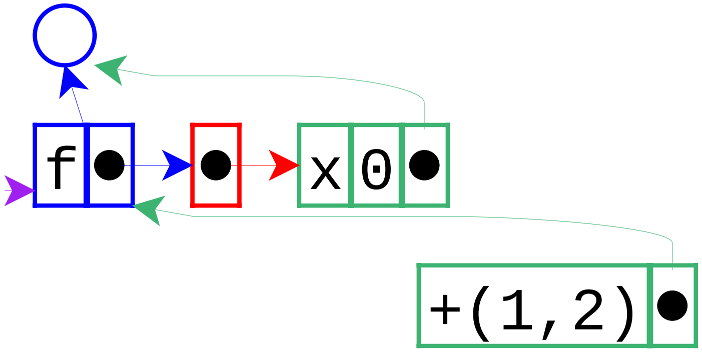
By using `proc` to delay evaluation, we can avoid unnecessary computation.

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

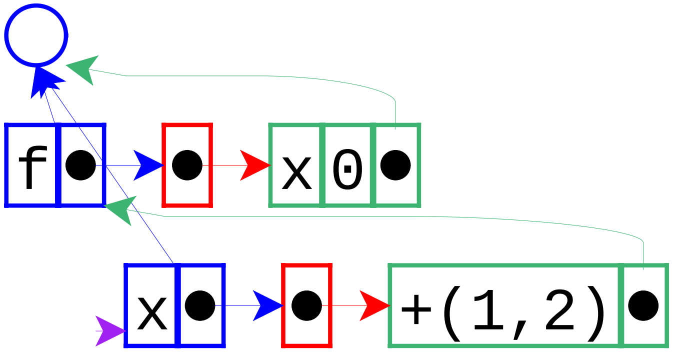
```
let f = proc(x)0
 in (f +(1,2))
```



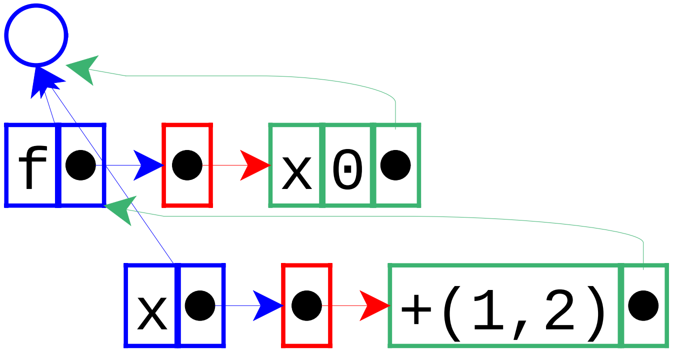

```
let f = proc(x)0
 in (f +(1,2))
```




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

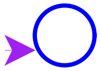



```
let f = proc(x)0
 in (f +(1,2))
```

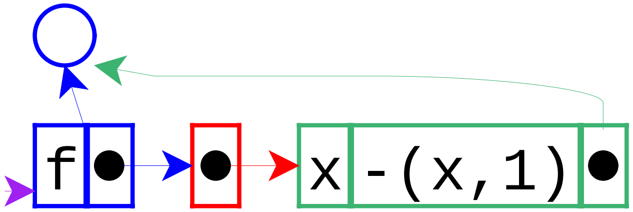



```
let f = proc(x)0
 in (f +(1,2))
```

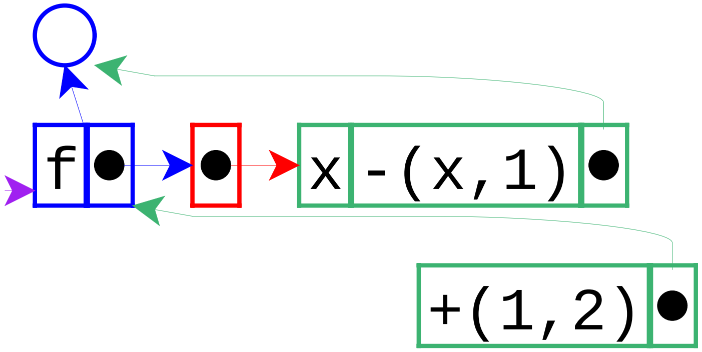
The result is 0.



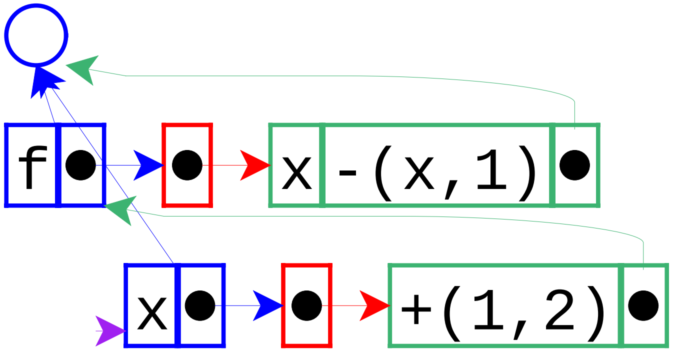
```
let f = proc(x)-(x,1)
 in (f +(1,2))
```



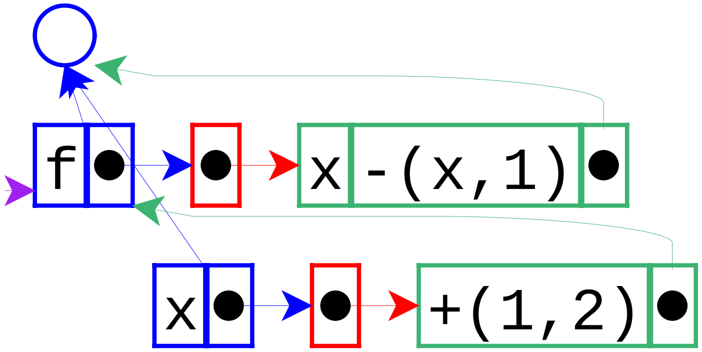

```
let f = proc(x)-(x,1)
 in (f +(1,2))
```




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

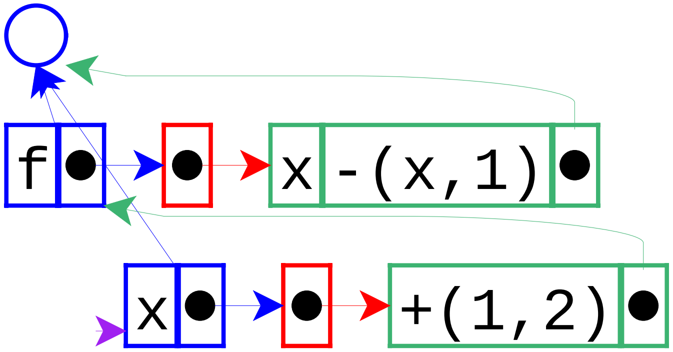



```
let f = proc(x)-(x,1)
 in (f +(1,2))
```



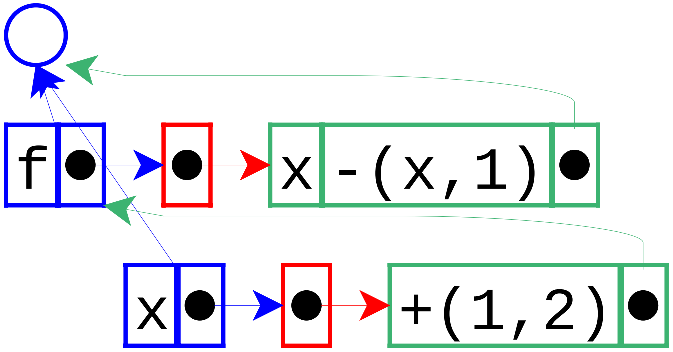

```
let f = proc(x)-(x,1)
 in (f +(1,2))
```

Force evaluation of thunk.




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

With 3 as the value of x.

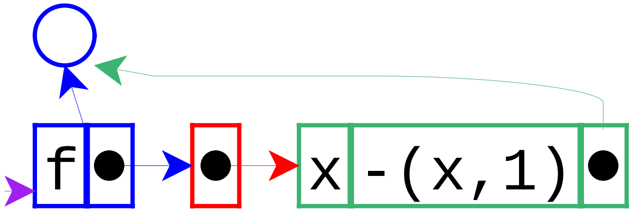



```
let f = proc(x)-(x,1)
 in (f +(1,2))
```

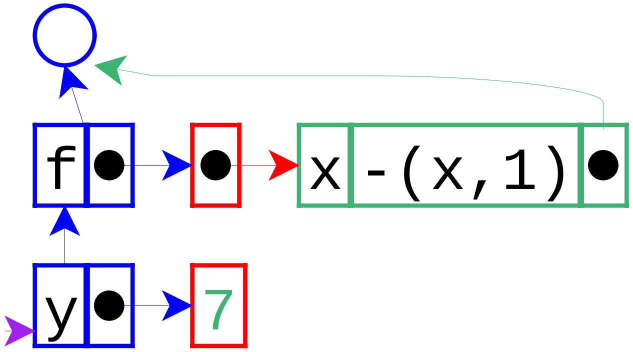
The result is 2.

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

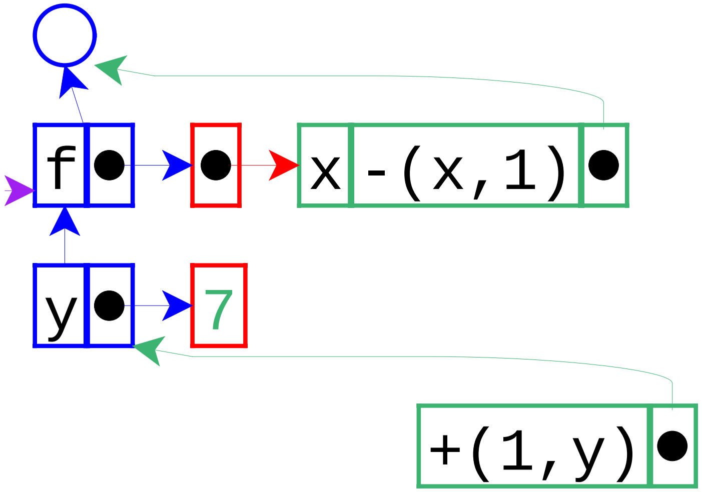
Lazy expression that needs its environment



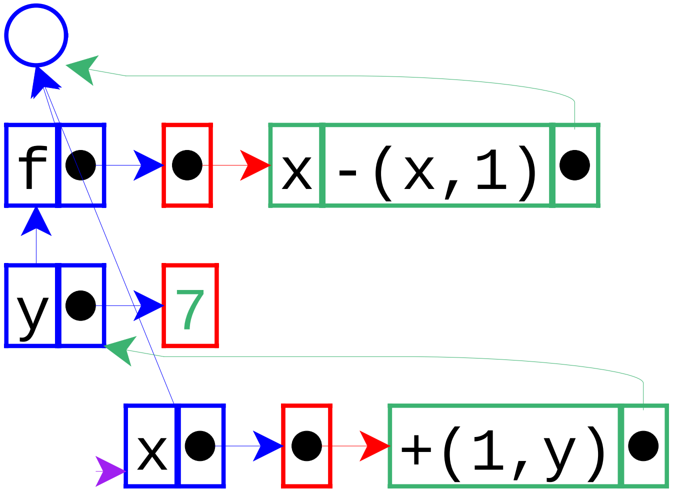

```
let f = proc(x)-(x,1)
 in let y = 7
     in (f +(1,y))
```



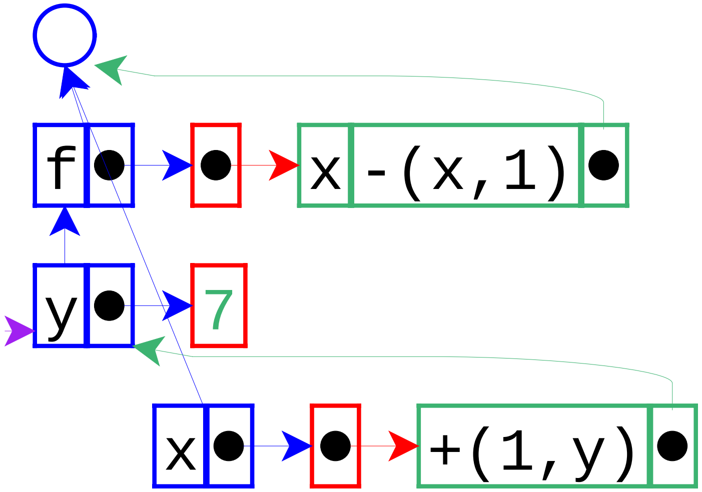

```
let f = proc(x)-(x,1)
 in let y = 7
     in (f +(1,y))
```



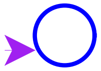

```
let f = proc(x)-(x,1)
 in let y = 7
     in (f +(1,y))
```




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

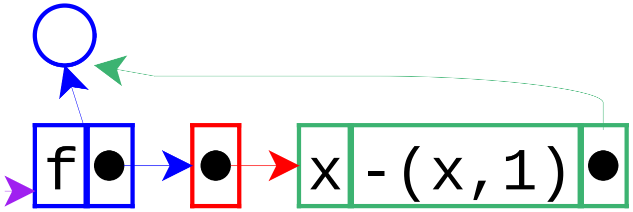



```
let f = proc(x)-(x,1)
 in let y = 7
     in (f +(1,y))
```

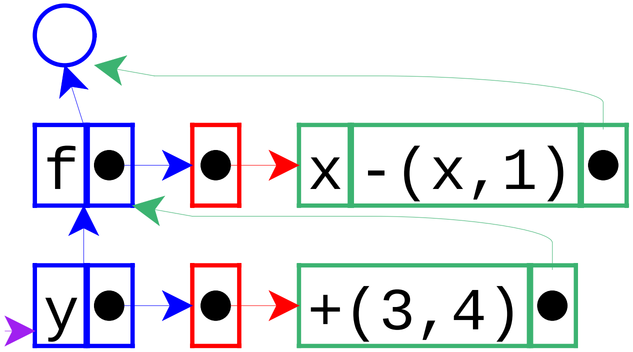


```
let f = proc(x)-(x,1)
 in let y = +(3,4)
     in (f +(1,y))
```

Change binding of y to an expression.

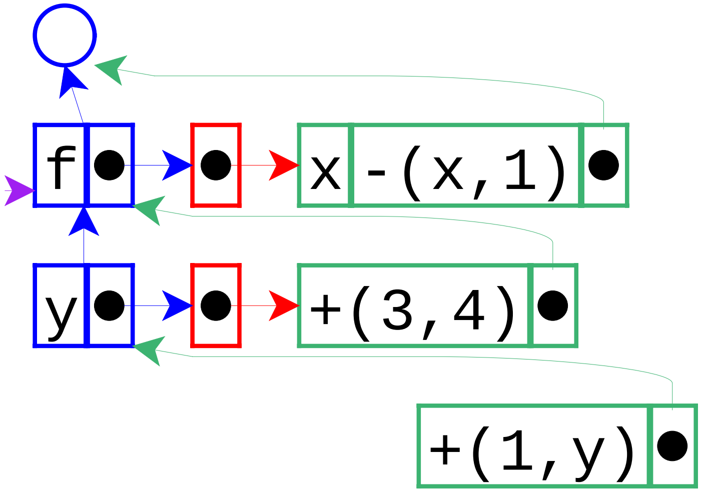



```
let f = proc(x)-(x,1)
 in let y = +(3,4)
     in (f +(1,y))
```

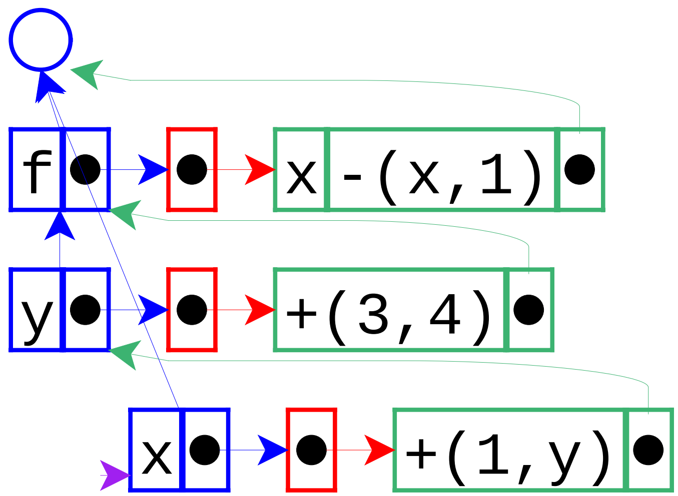



```
let f = proc(x)-(x,1)
 in let y = +(3,4)
     in (f +(1,y))
```

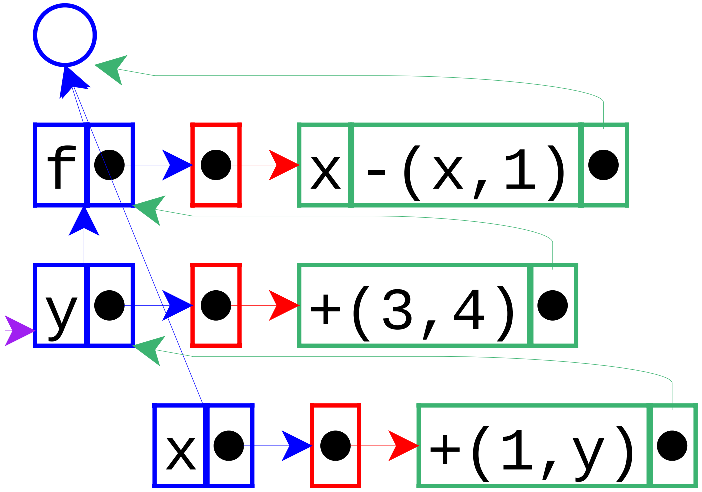
Added lazy binding for y.



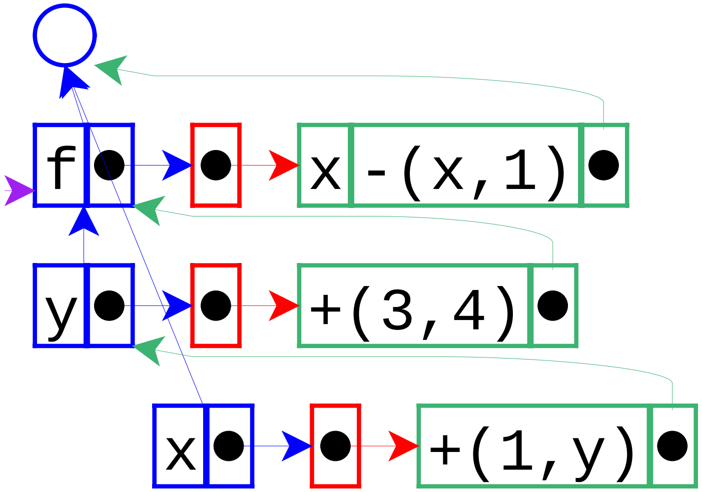

```
let f = proc(x)-(x,1)
 in let y = +(3,4)
     in (f +(1,y))
```




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




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




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

Interpreter changes:

- Change `eval-fun-rands` to create thunks.
- Change variable lookup to eval thunks.

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.

Interpreter changes:

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

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

Call-by-reference can augment either

- 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 none are call-by-name

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 }
```

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**.

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

```
let f = proc(x)0
 in (f <loop forever>)
```

- Eager answer: none
- Lazy answer: `0`