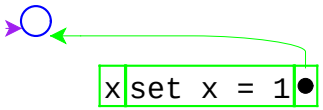
What is the result of this program?

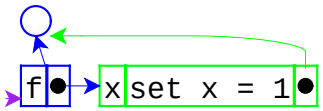
```
let f = proc(x) set x = 1
 in let y = 0
     in { (f y);
          y }
```

Is it 0 or 1?

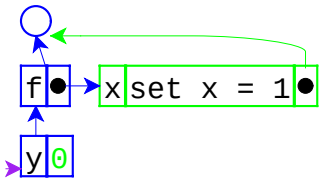
```
let f = proc(x) set x = 1
 in let y = 0
     in { (f y);
          y }
```

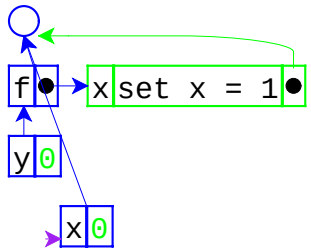

```
let f = proc(x) set x = 1
 in let y = 0
     in { (f y);
          y }
```

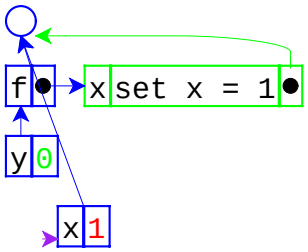

```
let f = proc(x) set x = 1
 in let y = 0
     in { (f y);
          y }
```



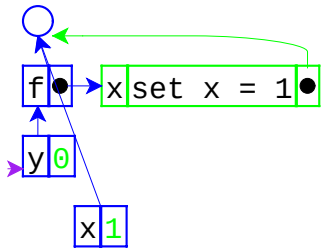

```
let f = proc(x) set x = 1
 in let y = 0
     in { (f y);
          y }
```


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


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


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

So the answer is 0.

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

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

The result above is 0, too.


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

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

But the result above is 1.

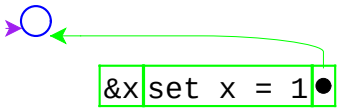
```
void f(int& x) {
  x = 1;
}

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

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

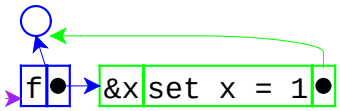
The previous example showed **call-by-value**.


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

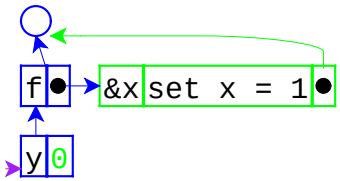
Adding call-by-reference parameters to our language.



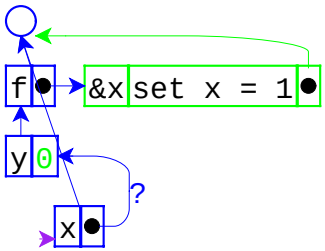

```
let f = proc(&x) set x = 1
 in let y = 0
     in { (f y);
          y }
```


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

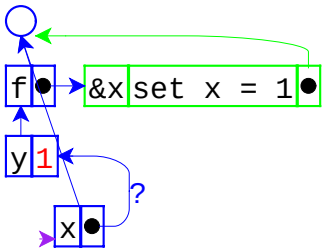




```
let f = proc(&x) set x = 1
 in let y = 0
     in { (f y);
          y }
```

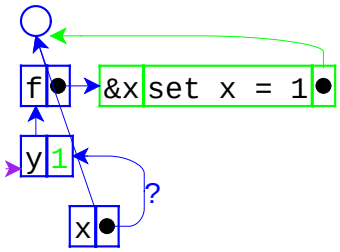




```
let f = proc(&x) set x = 1
 in let y = 0
     in { (f y);
          y }
```

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 }
```


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


What changes in the interpreter?

Same as before:

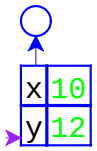
- **Expressed values:** Number + Proc
- **Denoted values:** Ref(Expressed Value)

Same as before:

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

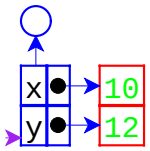
The difference is that application doesn't always create a new location for a new variable binding.

=> Separate location creation from environment extension



The old way

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

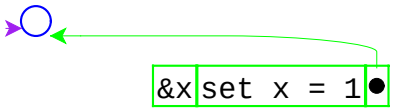



The new way

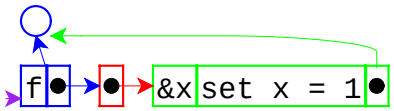
```
let x = 10
    y = 12
 in +(x,y)
```


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

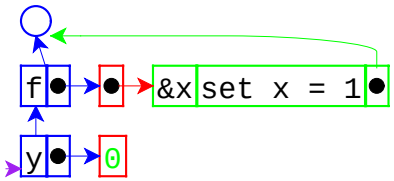
Do the previous evaluation the new way...



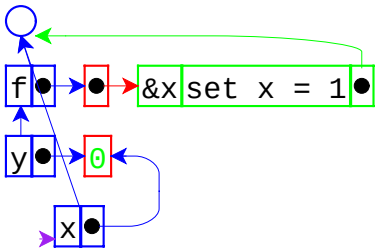

```
let f = proc(&x) set x = 1
 in let y = 0
     in { (f y);
          y }
```


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

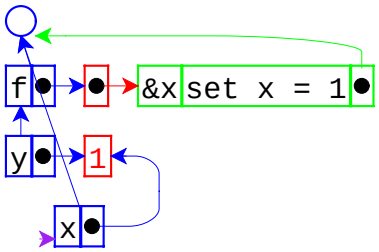




```
let f = proc(&x) set x = 1
 in let y = 0
     in { (f y);
          y }
```

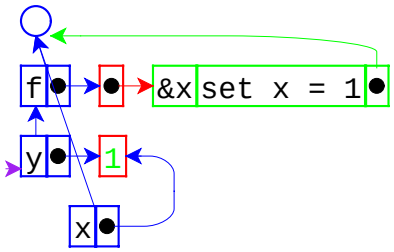




```
let f = proc(&x) set x = 1
 in let y = 0
     in { (f y);
          y }
```

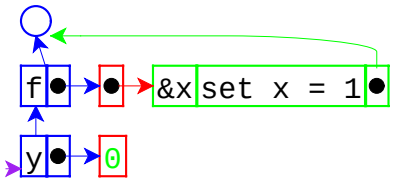
This time, the new environment frame points to a location box, which is consistent with other frames.




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

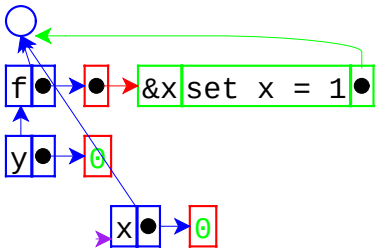




```
let f = proc(&x) set x = 1
 in let y = 0
     in { (f y);
          y                                    }
```


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

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




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

... always create a location.

Interpreter changes (starting with pre-`letrec` version):

- Add call-by-reference arguments (indicated by &).
  - New `var` type, with `cbv-var` and `cbr-var`
- Create explicit locations for variables.
  - `location`, `location-val`, `location-set!`
- Change variable lookup to deference locations.
- Change `set` to work on locations.
- Change `eval-rands` and `apply-proc`.
  - `make-var-location` helper proc

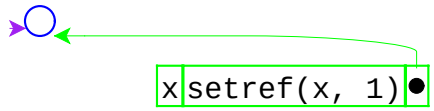

```
void f(int* x) {
  *x = 1;
}

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



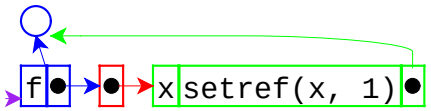
```
void f(int* x) {
  *x = 1;
}

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

This is back to **call-by-value**, but with a reference as a value.

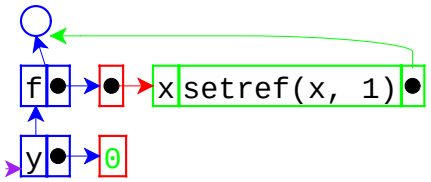
To study this form of call, we can add explicit references to our language, too.



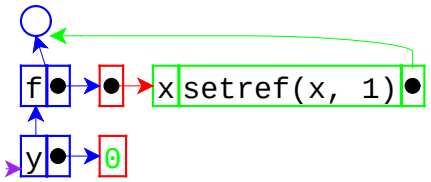

```
let f = proc(x) setref(x, 1)
 in let y = 0
     in { (f ref(y));
          y }
```



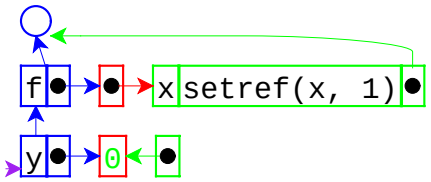

```
let f = proc(x) setref(x, 1)
 in let y = 0
     in { (f ref(y));
          y }
```

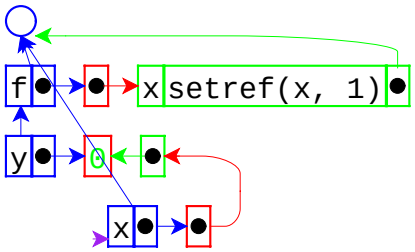

```
let f = proc(x) setref(x, 1)
 in let y = 0
     in { (f ref(y));
          y }
```

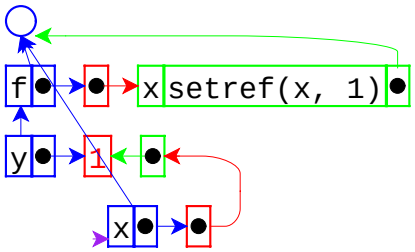

```
let f = proc(x) setref(x, 1)
 in let y = 0
     in { (f ref(y));
          y }
```

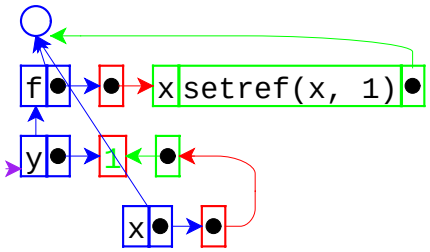

```
let f = proc(x) setref(x, 1)
 in let y = 0
     in { (f ref(y));
          y }
```




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


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


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


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.