Assigning to a Variable

What is the result of this program?

```plaintext
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 }
Assigning to a Variable

\[
\text{let } f = \text{proc(x) set x = 1} \\
\text{in let y = 0} \\
\text{in \{ (f y); y \}}
\]
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 }
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 }
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++

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

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

The result above is 0, too
Variables in C++

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

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

But the result above is 1
Variables in C++

```cpp
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

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

\[
\begin{align*}
\text{in } & \text{ let } y = 0 \\
\text{in } & \text{ let } \{ (f \ y); \ y \ \} 
\end{align*}
\]
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 }
Assignment and Call-by-Reference

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

\[
\text{let } f = \text{proc}(\&x) \ \text{set } x = 1 \\
\text{in let } y = 0 \\
\text{in } \{ \ (f \ y); \\
\text{y \ } \}
\]
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 }
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

```haskell
let x = 10
    y = 12
in +(x,y)
```
Assignment and Call-by-Reference

The new way

```plaintext
let x = 10
    y = 12
in +(x,y)
```
Call-by-Reference

Do the previous evaluation the new way...

```plaintext
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.

\[
\text{let } f = \text{proc}(\&x) \ \text{set } x = 1 \\
\quad \text{in let } y = 0 \\
\quad \quad \text{in } \{ (f \ y); y \} 
\]
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  }
If call-by-reference argument is not a variable...

\[
\text{let } f = \text{proc}(&x) \text{ set } x = 1 \\
\text{in let } y = 0 \\
\text{in } \{ (f 0); \text{ set } x = 1; \ y \} 
\]
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++

```cpp
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

\[
\text{let } f = \proc(x) \setref(x, 1) \\
\text{in let } y = 0 \\
\text{in } \{ (f \ref(y)); y \} 
\]
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 }
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 }
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

\[
\text{let } f = \text{proc}(x) 0 \\
\text{in } (f + (1, + (2, + (3, + (4, + (5, 6)))))))
\]

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:

```plaintext
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

\[
\text{let } f = \text{proc}(x)0 \\
\text{in } (f + (1,2))
\]
Evaluation with Lazy Arguments

\begin{align*}
\text{let } f &= \text{proc}(x) 0 \\
\text{in } (f + (1,2))
\end{align*}
Evaluation with Lazy Arguments

Application creates a new kind of green box, with two slots: a \textit{thunk}

\begin{verbatim}
let f = proc(x)0
    in (f +(1,2))
\end{verbatim}
let $f = \text{proc}(x)0$

in $(f + (1,2))$
The result is 0

\[
\text{let } f = \text{proc}(x)0 \\
\text{in } (f + (1,2))
\]
let f = proc(x)-(x,1) in (f +(1,2))
let f = proc(x)-(x,1) in (f +(1,2))
let \( f = \text{proc}(x)-(x,1) \)
\[ \text{in } (f + (1,2)) \]
Evaluation with Lazy Arguments

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

lookup of x...
Evaluation with Lazy Arguments

\[
\text{let } f = \text{proc}(x) - (x, 1) \\
\text{in } (f + (1, 2))
\]

... forces evaluation of the thunk
Evaluation with Lazy Arguments

let \( f = \text{proc}(x) - (x, 1) \)
\text{in} (f + (1, 2))

so 3 is the value of \( x \)
Evaluation with Lazy Arguments

Let $f = \text{proc}(x) - (x, 1)$ in $(f + (1, 2))$

The result is 2
Evaluation with Lazy Arguments

Lazy expression that needs its environment...

```plaintext
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))
Evaluation with Lazy Arguments

```
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))
Evaluation with Lazy Arguments

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

Evaluation of x forces the thunk...
Evaluation with Lazy Arguments

Let's define a function and trigger its evaluation with the `thunk`'s environment, not the current one.

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

```
Evaluation with Lazy Arguments

让 f = proc(x) - (x, 1)
在 let y = 7
在 (f + (1, y))
```
let f = proc(x) - (x, 1) 
    in let y = 7 
        in (f + (1, y))
What if the right-hand side for \( y \) is an expression, instead of a value?

\[
\text{let } f = \text{proc}(x) - (x, 1) \\
\text{in let } y = +(3, 4) \\
\text{in } (f + (1, y))
\]
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))

Added thunk for the value of y
Evaluation with Lazy Arguments

Another thunk for the argument of \( f \)

```plaintext
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

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

and so on (to get 7)
Implementing Lazy Evaluation

Interpreter changes:

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

(Implement in DrScheme)
The lazy strategy we just implemented is **call-by-name**

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

```plaintext
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))
let f = proc(x)-(x,x)
in (f +(1,2))
let f = proc(x)-(x,x) in (f +(1,2))
Evaluation with Lazy Arguments

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

lookup of x...
Evaluation with Lazy Arguments

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

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

lookup of x again gets 3
Evaluation with Lazy Arguments

\[ f(x, x) \]

(The result is 0)

\[
\text{let } f = \text{proc}(&x) \text{ set } x = 1 \\
\quad \text{in } \text{let } y = 0 \\
\quad \quad \text{in } \{ (f y); y \}
\]
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 = \textit{lazy} evaluation
- Call-by-value = \textit{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
Why don't more languages provide lazy evaluation?

• Disadvantage: evaluation order is not obvious

```plaintext
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 \texttt{set} form

• Such languages are called \textit{purely functional}
  
  ○ Note: call-by-reference is meaningless in a purely functional language

• A language with \texttt{set} can be called \textit{imperative}
Laziness and Eagerness

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

\[
\text{let } f = \text{proc}(x)0 \\
\text{in } (f \ [\text{loop forever}])
\]

• 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