

Lexical Addressing

INTERPRETATION

Review



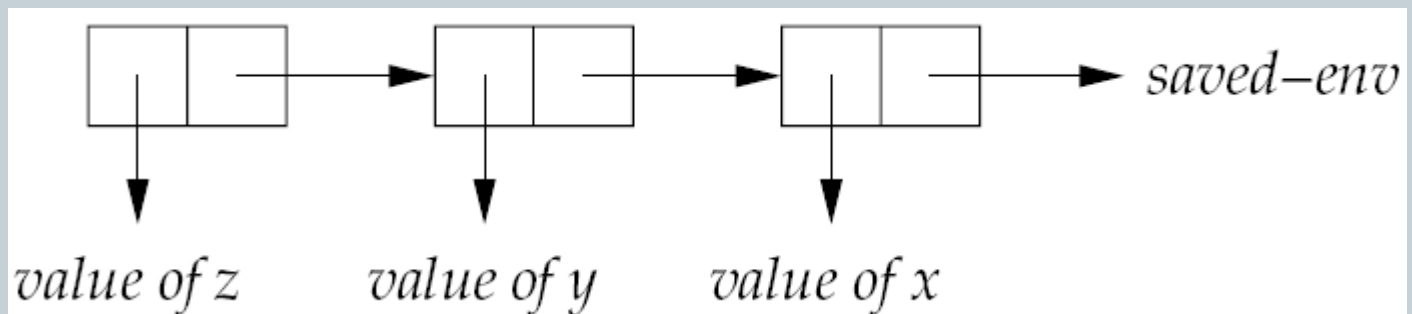
T. METIN SEZGIN

New environment interface



nameless-environment

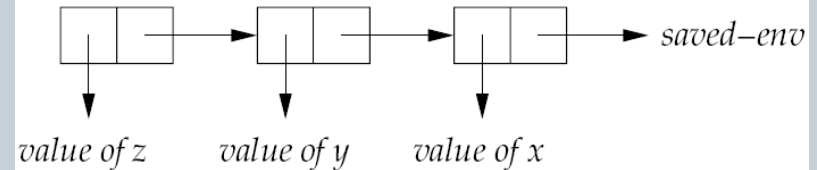
nameless-environment? : $SchemeVal \rightarrow Bool$
empty-nameless-env : $() \rightarrow Nameless-env$
extend-nameless-env : $Expval \times Nameless-env \rightarrow Nameless-env$
apply-nameless-env : $Nameless-env \times Lexaddr \rightarrow DenVal$



New environment interface



nameless-environment? : $SchemeVal \rightarrow Bool$
empty-nameless-env : $() \rightarrow Nameless-env$
extend-nameless-env : $Expval \times Nameless-env \rightarrow Nameless-env$
apply-nameless-env : $Nameless-env \times Lexaddr \rightarrow DenVal$



```
nameless-environment? : SchemeVal → Bool
(define nameless-environment?
  (lambda (x)
    ((list-of expval?) x)))
```

```
empty-nameless-env : () → Nameless-env
(define empty-nameless-env
  (lambda ()
    ' ()))
```

```
extend-nameless-env : ExpVal × Nameless-env → Nameless-env
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons val nameless-env)))
```

```
apply-nameless-env : Nameless-env × Lexaddr → ExpVal
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))
```

Procedure specification and implementation



```
(apply-procedure (procedure body  $\rho$ ) val)  
= (value-of body (extend-nameless-env val  $\rho$ ))
```

```
procedure : Nameless-exp  $\times$  Nameless-env  $\rightarrow$  Proc  
(define-datatype proc proc?  
  (procedure  
    (body expression?)  
    (saved-nameless-env nameless-environment?)))
```

```
apply-procedure : Proc  $\times$  ExpVal  $\rightarrow$  ExpVal  
(define apply-procedure  
  (lambda (proc1 val)  
    (cases proc proc1  
      (procedure (body saved-nameless-env)  
        (value-of body  
          (extend-nameless-env val saved-nameless-env))))))
```

Interpreter for the new language



```
value-of : Nameless-exp × Nameless-env → ExpVal
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (const-exp (num)    ...as before...)
      (diff-exp (exp1 exp2) ...as before...)
      (zero?-exp (exp1)    ...as before...)
      (if-exp (exp1 exp2 exp3) ...as before...)
      (call-exp (rator rand) ...as before...)

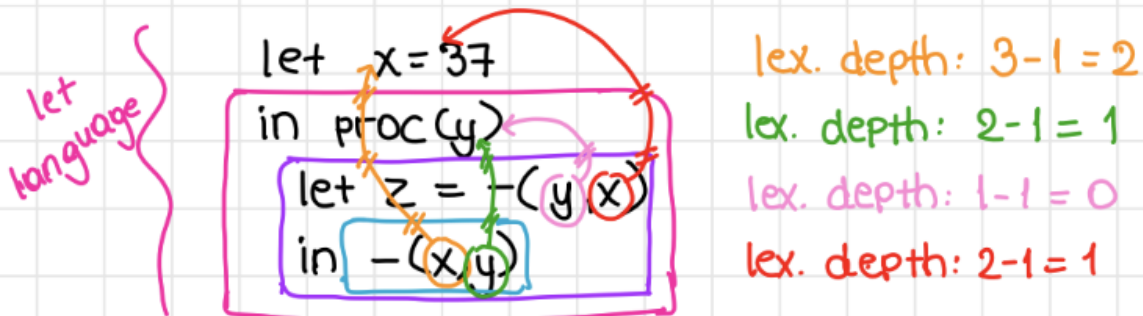
      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp))))))
```

★ Arguments to procedures always found at the expected places.



Ceren Tarim

```

(lambda (x)
  ((lambda (a)
     (x a)))
  x))

```

```

(nameless-lambda
  ((nameless-lambda
    (#1 #0))
   #0))

```

! replace var names with their lexical depth.

!

translating: $\text{exp } x \text{ Serv} \rightarrow \text{nameless-exp}$

interpreter: $\text{nameless-exp } x \text{ nameless-env} \rightarrow \text{expval}$

Ceren Tarim

the translator:

$Expression ::= \%lexref \text{ number}$ address

```
nameless-var-exp (num)
```

~~Expression ::= %let Expression in Expression~~

```
nameless-let-exp (exp1 body)
```

$$\textit{Expression} ::= \% \textit{lexproc} \textit{Expression}$$

```
nameless-proc-exp (body)
```

! there is no var

Ceren Tarım

translation-of : $Exp \times Senv \rightarrow Nameless-exp$

```
(define translation-of
  (lambda (exp senv)
    (cases expression exp
      (const-exp (num) (const-exp num))
      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))
      (zero?-exp (exp1)
        (zero?-exp
          (translation-of exp1 senv)))
      (if-exp (exp1 exp2 exp3)
        (if-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)
          (translation-of exp3 senv)))
      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))
      (let-exp (var exp1 body)
        (nameless-let-exp
          (translation-of exp1 senv)
          (translation-of body
            (extend-senv var senv))))
      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))
      (call-exp (rator rand)
        (call-exp
          (translation-of rator senv)
          (translation-of rand senv)))
      (else
        (report-invalid-source-expression exp))))
```

grab the address

throw "var"

append it
to the env.
using new var

extended ver.
of the env.

Ceren Tarim

```

run : String → ExpVal
(define run
  (lambda (string)
    (value-of-program
      (translation-of-program
        (scan&parse string))))))

```

new environment interface: (interpreter)

```

nameless-environment? : SchemeVal → Bool
(define nameless-environment?
  (lambda (x)
    ((list-of expval?) x)))

```

```

empty-nameless-env : () → Nameless-env
(define empty-nameless-env
  (lambda ()
    '()))

```

```

extend-nameless-env : ExpVal × Nameless-env → Nameless-env
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons val nameless-env)))

```

```

apply-nameless-env : Nameless-env × Lexaddr → ExpVal
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))

```

pre-defined

Ceren Tarim

interpreter for the new language:

value-of : *Nameless-exp* \times *Nameless-env* \rightarrow *ExpVal*

```
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp
      (const-exp (num) ...as before...)
      (diff-exp (exp1 exp2) ...as before...)
      (zero?-exp (exp1) ...as before...)
      (if-exp (exp1 exp2 exp3) ...as before...)
      (call-exp (rator rand) ...as before...)

      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp))))))
```

Ceren Tarim

State – Effects



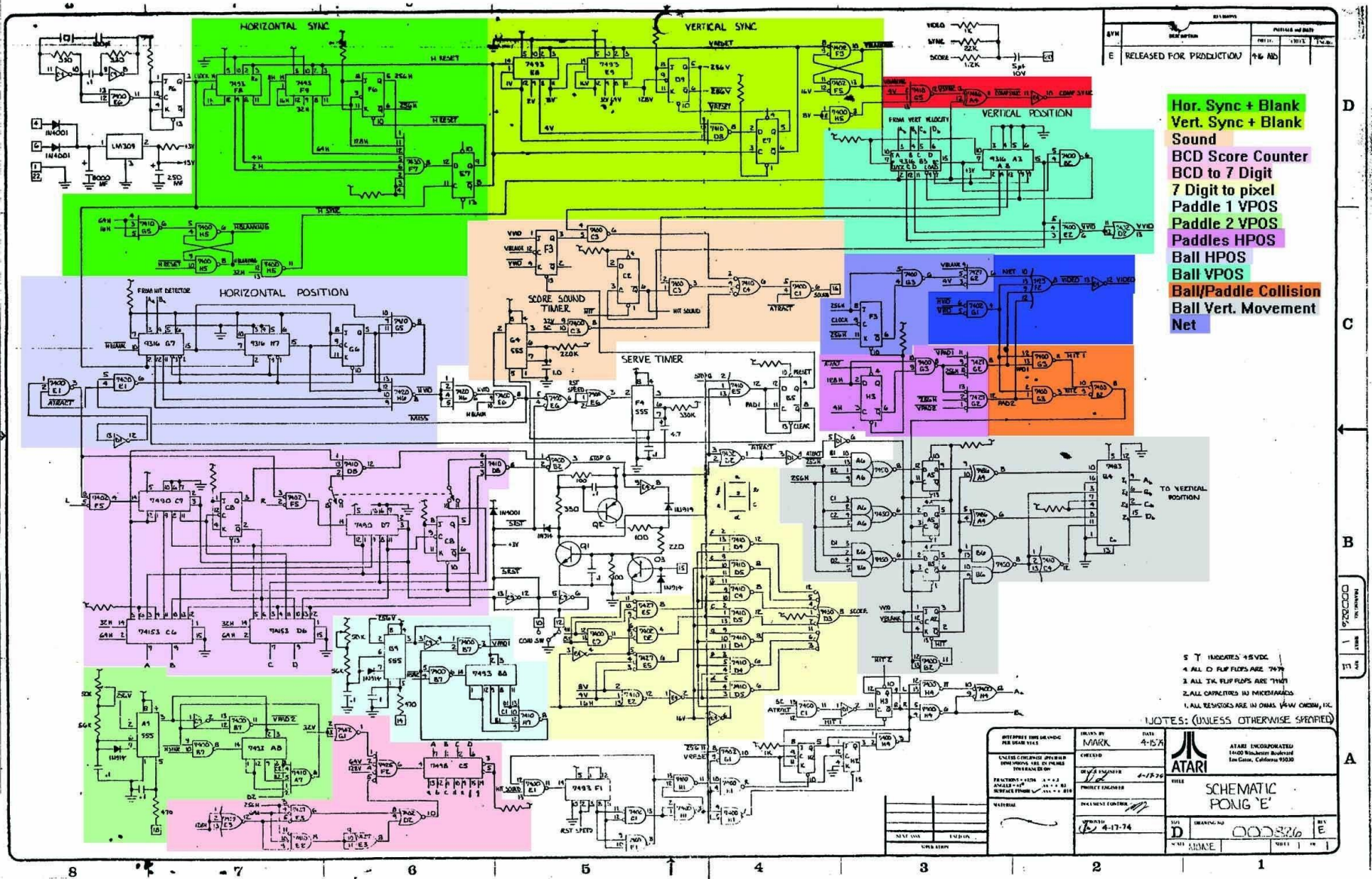
T. METIN SEZGIN

Nuggets



- Life before programming languages was hard
- No magic in building interpreters
- Memory model makes language more expressive

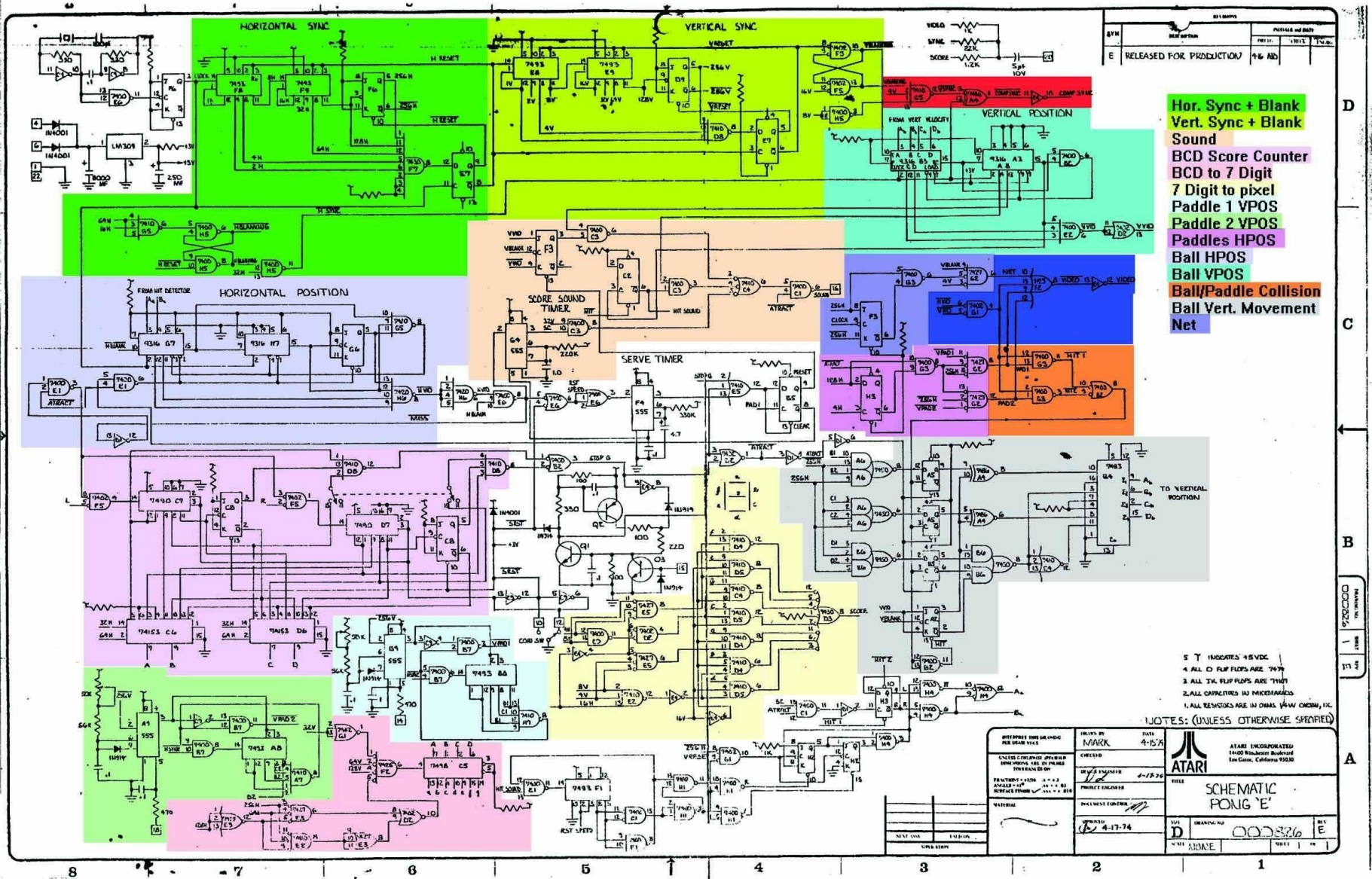
Life before programming languages



Life before programming languages



Life before programming languages



Nugget



- No magic in building interpreters

[A minimal C compiler](#)

[Conway's Game of Life](#)

[An interpreter in Conway's Game of Life](#)

Languages considered so far



- LET
- PROC
- LETREC
- EXPLICIT-REFS (EREF)

Computational Effects



- So far we have considered
 - Expressions generating values
 - Everything local
 - No notion of global state
 - No global storage
- We want to be able to
 - Read memory locations
 - Print values in the memory
 - Write to the memory
 - Have global variables
 - Share values across separate computations
- We need
 - A model for memory
 - ✦ Access memory locations
 - ✦ Modify memory contents

New concepts



- **Storable values**
 - What sorts of things can we store?
- **Memory stores**
 - Where do we store things?
- **Memory references (pointers)**
 - How do we access the stores?

The new design



- Denotable and Expressed values

$$\begin{aligned} \textit{ExpVal} &= \textit{Int} + \textit{Bool} + \textit{Proc} + \textit{Ref}(\textit{ExpVal}) \\ \textit{DenVal} &= \textit{ExpVal} \end{aligned}$$

- Three new operations
 - `newref`
 - `deref`
 - `setref`

Example: references help us share variables



```
let x = newref(0)
in letrec even(dummy)
    = if zero?(deref(x))
      then 1
      else begin
          setref(x, -(deref(x), 1));
          (odd 888)
        end
    odd(dummy)
    = if zero?(deref(x))
      then 0
      else begin
          setref(x, -(deref(x), 1));
          (even 888)
        end
    in begin setref(x, 13); (odd 888) end
```

Example: references help us create hidden state

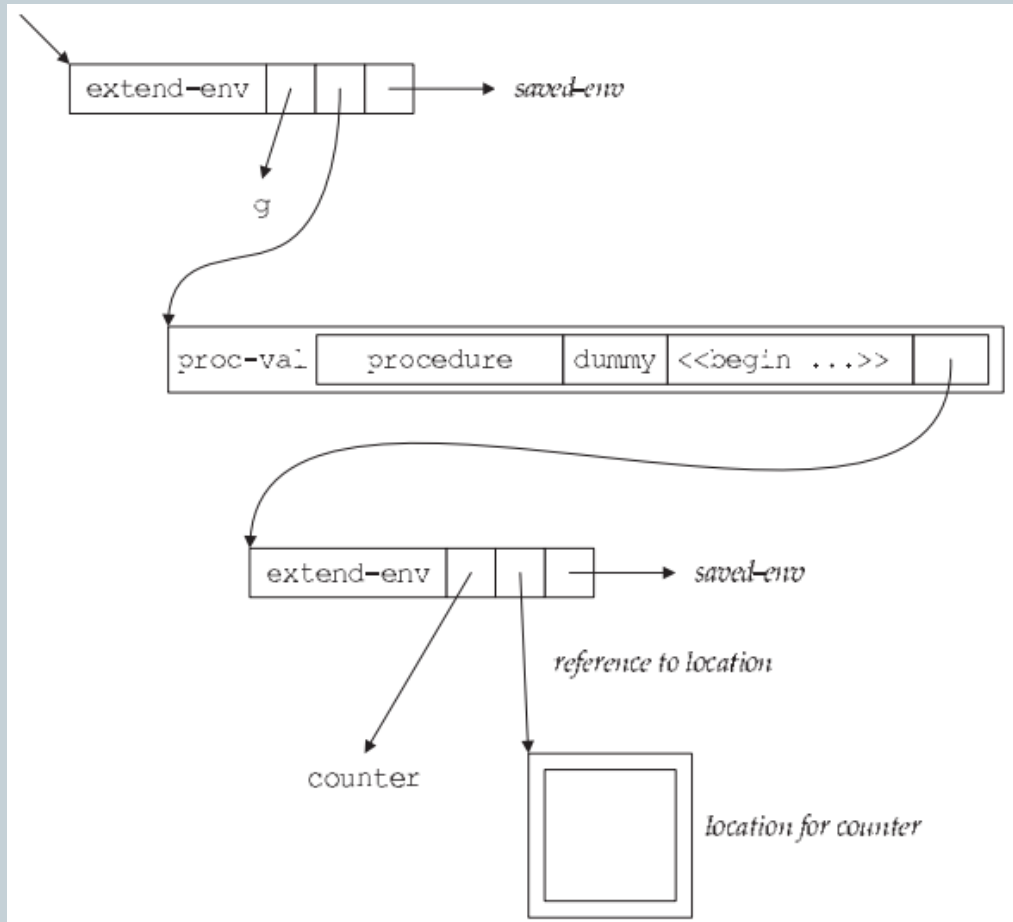


```
let g = let counter = newref(0)
      in proc (dummy)
        begin
          setref(counter, -(deref(counter), -1));
          deref(counter)
        end
in let a = (g 11)
  in let b = (g 11)
    in -(a,b)
```

The entire expression evaluates to -1

Behind the scenes...

```
let g = let counter = newref(0)
      in proc (dummy)
        begin
          setref(counter, -(deref(counter), -1));
          deref(counter)
        end
in let a = (g 11)
  in let b = (g 11)
    in -(a,b)
```



Example: reference to a reference



```
let x = newref(newref(0))
in begin
    setref(deref(x), 11);
    deref(deref(x))
end
```

What does this evaluate to?

EREF implementation



- What happens to the store?
- How do we represent/implement stores?
- Behavior specification
- Implementation

Nugget



**In order to add the memory
feature to the language, we
need a data structure**

Store passing specifications



- The new **value-of** $(\text{value-of } exp_1 \ \rho \ \sigma_0) = (val_1, \sigma_1)$

Nugget



**We also need to rewrite the
rules of evaluation to use the
memory**

Store passing specifications



- The new **value-of** $(\text{value-of } exp_1 \ \rho \ \sigma_0) = (val_1, \sigma_1)$
- Example $(\text{value-of } (\text{const-exp } n) \ \rho \ \sigma) = (n, \sigma)$
- More examples

$$\begin{array}{l} (\text{value-of } exp_1 \ \rho \ \sigma_0) = (val_1, \sigma_1) \\ (\text{value-of } exp_2 \ \rho \ \sigma_1) = (val_2, \sigma_2) \end{array}$$

$$(\text{value-of } (\text{diff-exp } exp_1 \ exp_2) \ \rho \ \sigma_0) = ([val_1] - [val_2], \sigma_2)$$

$$(\text{value-of } exp_1 \ \rho \ \sigma_0) = (val_1, \sigma_1)$$

$$\begin{aligned} & (\text{value-of } (\text{if-exp } exp_1 \ exp_2 \ exp_3) \ \rho \ \sigma_0) \\ &= \begin{cases} (\text{value-of } exp_2 \ \rho \ \sigma_1) & \text{if } (\text{expval} \rightarrow \text{bool } val_1) = \#t \\ (\text{value-of } exp_3 \ \rho \ \sigma_1) & \text{if } (\text{expval} \rightarrow \text{bool } val_1) = \#f \end{cases} \end{aligned}$$

Nugget



**We also need to write the rules
of evaluation for the new
expressions**

Grammar specification



- The new grammar

```
Expression ::= newref (Expression)
               newref-exp (exp1)

Expression ::= deref (Expression)
               deref-exp (exp1)

Expression ::= setref (Expression , Expression)
               setref-exp (exp1 exp2)
```

- Specification

$$\frac{(\text{value-of } \text{exp } \rho \sigma_0) = (val, \sigma_1) \quad l \notin \text{dom}(\sigma_1)}{(\text{value-of } (\text{newref-exp } \text{exp}) \rho \sigma_0) = ((\text{ref-val } l), [l=val] \sigma_1)}$$
$$\frac{(\text{value-of } \text{exp } \rho \sigma_0) = (l, \sigma_1)}{(\text{value-of } (\text{deref-exp } \text{exp}) \rho \sigma_0) = (\sigma_1(l), \sigma_1)}$$
$$\frac{\begin{array}{l} (\text{value-of } \text{exp}_1 \rho \sigma_0) = (l, \sigma_1) \\ (\text{value-of } \text{exp}_2 \rho \sigma_1) = (val, \sigma_2) \end{array}}{(\text{value-of } (\text{setref-exp } \text{exp}_1 \text{exp}_2) \rho \sigma_0) = ([23], [l=val] \sigma_2)}$$