Lexical Addressing

INTERPRETATION

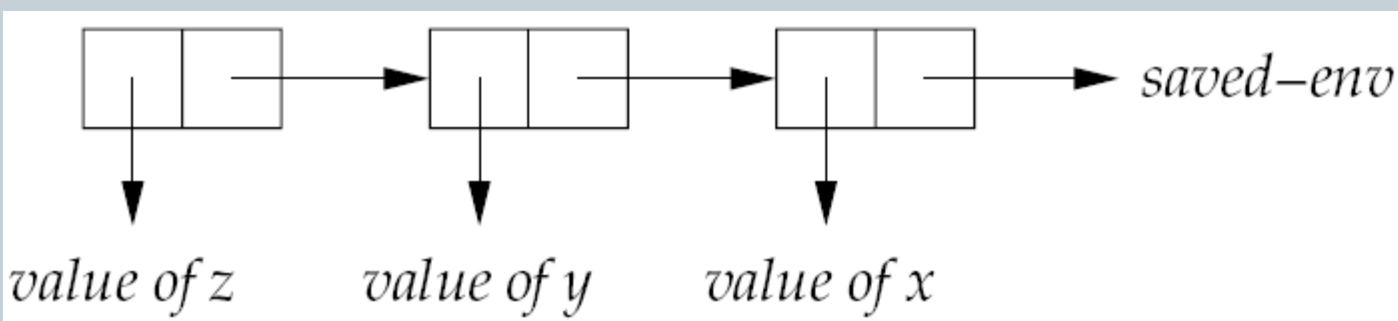
Review

T. METIN SEZGIN

New environment interface

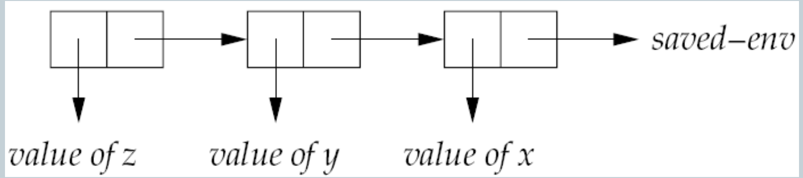
`nameless-environment`

nameless-environment?	: $\mathit{SchemeVal} \to \mathit{Bool}$
empty-nameless-env	: $() \to \mathit{Nameless\text{-}env}$
extend-nameless-env	: $\mathit{Expval} \times \mathit{Nameless\text{-}env} \to \mathit{Nameless\text{-}env}$
apply-nameless-env	: $\mathit{Nameless\text{-}env} \times \mathit{Lexaddr} \to \mathit{DenVal}$



New environment interface

nameless-environment?	: *SchemeVal* → *Bool*
empty-nameless-env	: () → *Nameless-env*
extend-nameless-env	: *Expval* × *Nameless-env* → *Nameless-env*
apply-nameless-env	: *Nameless-env* × *Lexaddr* → *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* ρ) *val*)
= (value-of *body* (extend-nameless-env *val* ρ))

procedure : *Nameless-exp* $\times$ *Nameless-env* $\to$ *Proc*

```
(define-datatype proc proc?
  (procedure
    (body expression?)
    (saved-nameless-env nameless-environment?)))
```

apply-procedure : *Proc* $\times$ *ExpVal* $\to$ *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)))))
```

Nameless Interpreter

run = String → ExpVal

```
(define run
  (lambda (string)
    (value-of-program
      (translation-of-program
        (scan&parse string)))))
```

LET

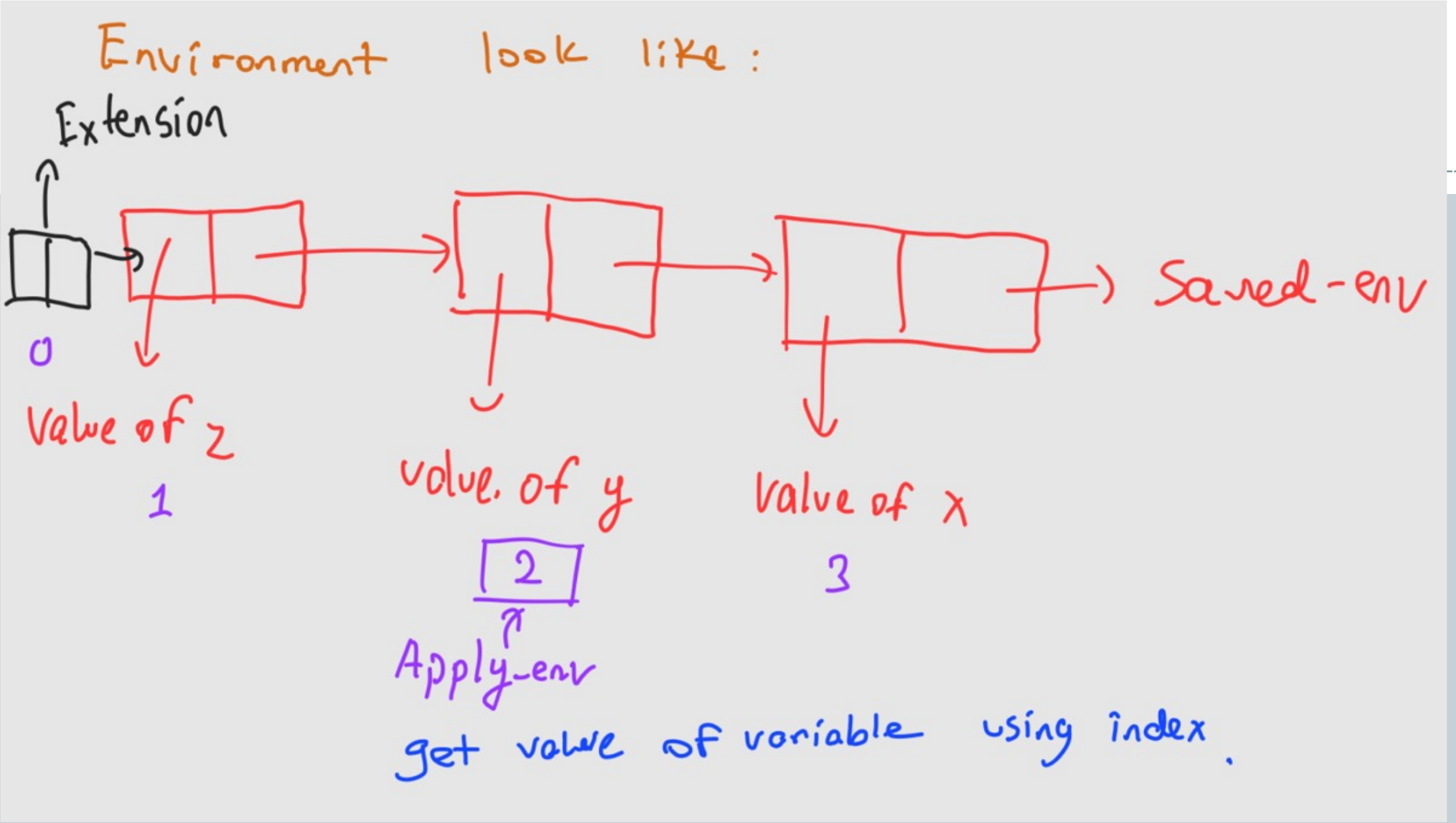
nameless LET

produces its AST and then feeds it to translator

Ayten Dilara Yavuz

the treslotor is a one time job. Don't runs the code or concern the output of code pieces just read and convert it into nameless let,

Ahmet Yesevi



Ahmet Yesevi

translation-of function:

Evaluate body with extended static environment

```
translation-of : Exp × Senv → 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)))))
```

Pass translation to sub-expressions

By using recursion we can make translation even in 1 slide long code :)

Birkan Celik

⇒ Nameless Let is not designed for user usage but computers.
⇒ Expressed and denoted values are same as LET for user perspective.
⇒ Nameless version is more efficient to use since it uses variable addresses directly for look up.

Eren Berke Demirbas

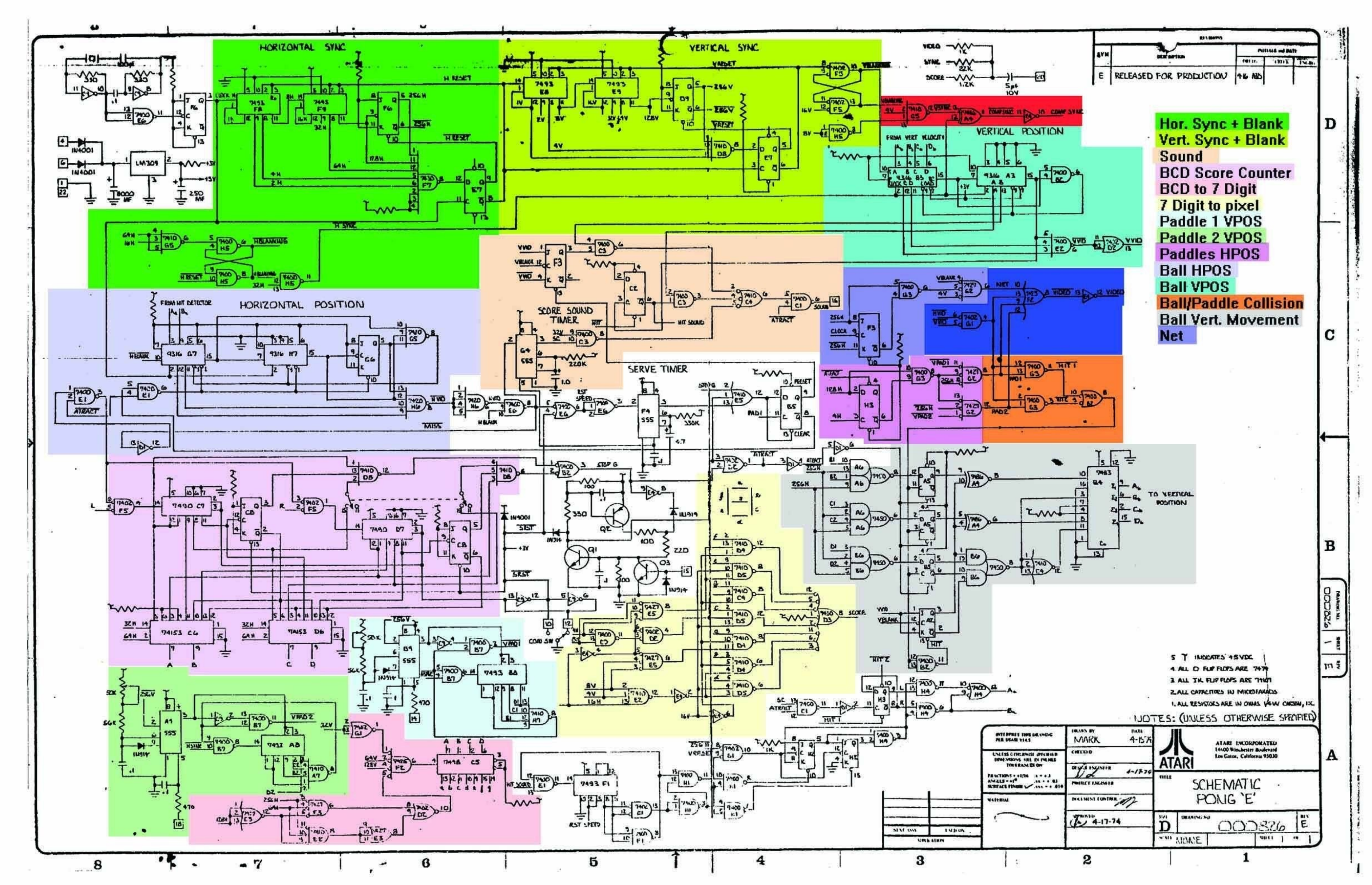
State – Effects

T. METIN SEZGIN

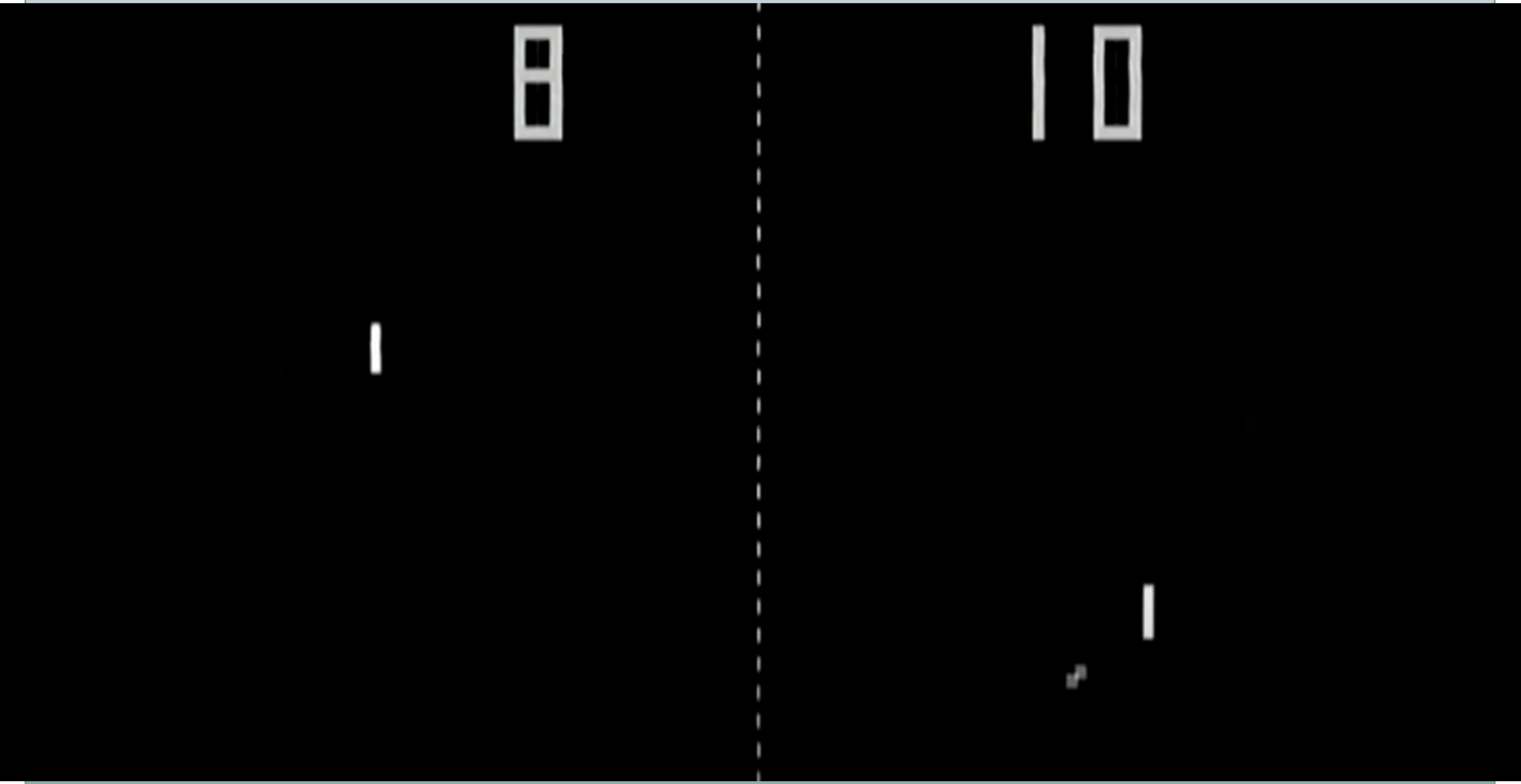
Nuggets

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

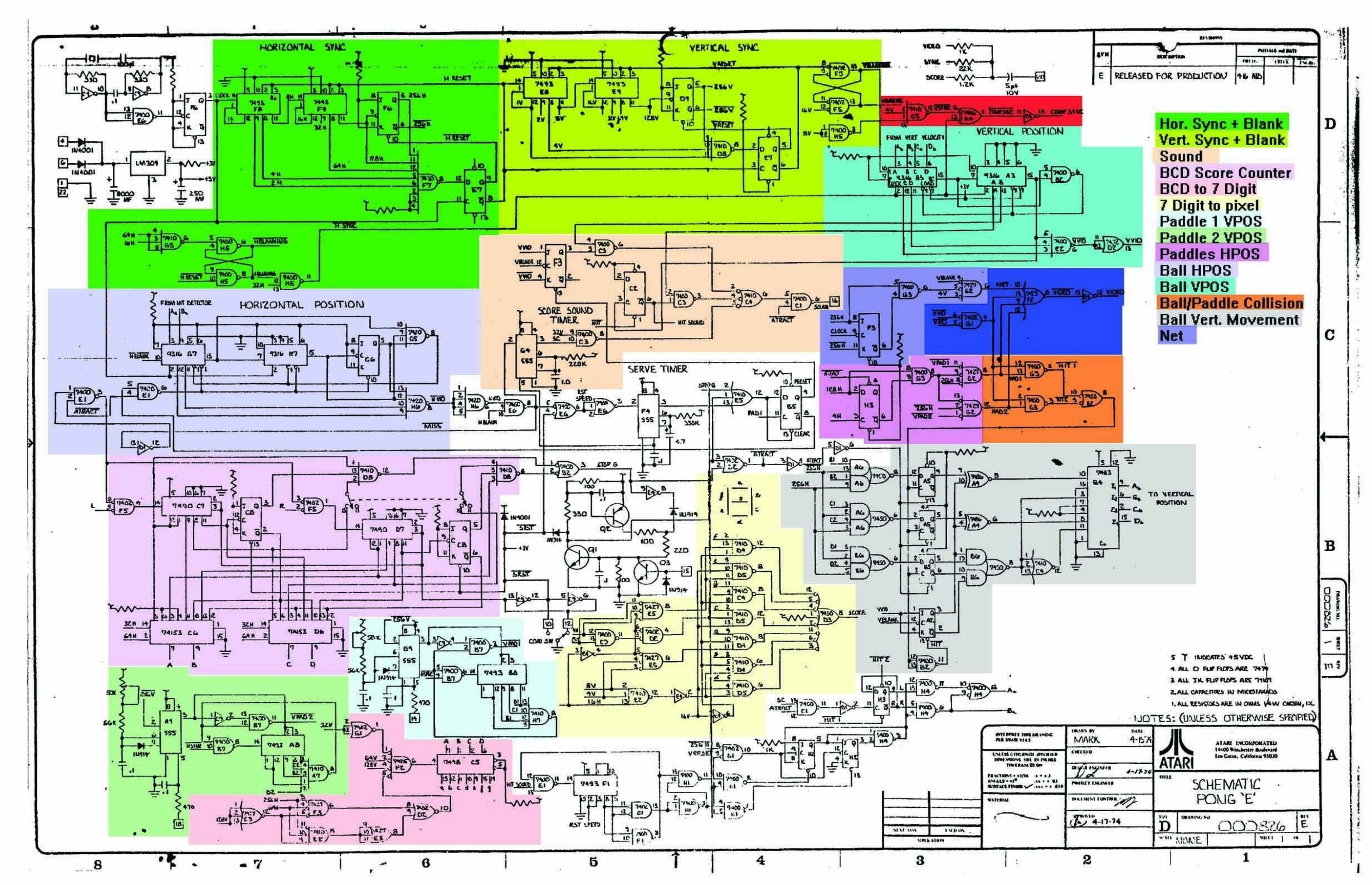
Life before programming languages



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

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

- 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
```

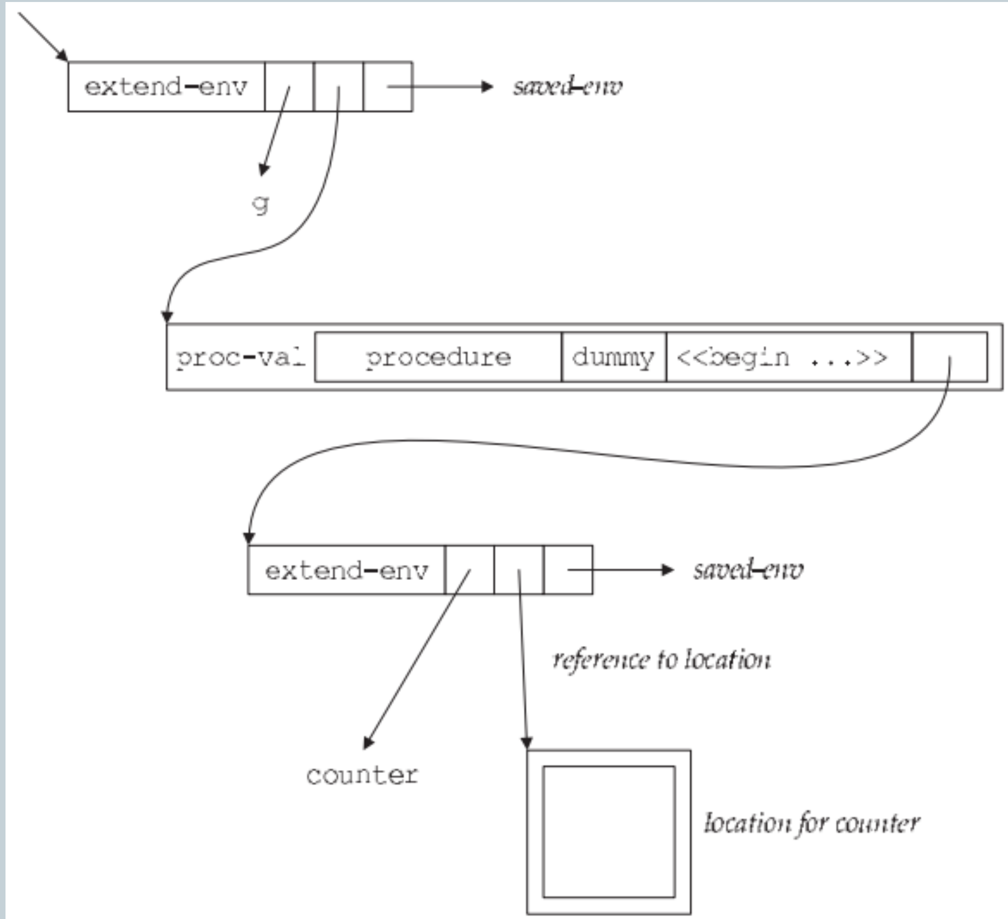
Example: references help us create hidden state

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

Nugget

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

Store passing specifications

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

- Example $(\texttt{value-of}\ (\texttt{const-exp}\ n)\ \rho\ \sigma) = (n, \sigma)$

- More examples

$$\frac{\begin{array}{c}(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1) \\ (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) = (val_2, \sigma_2)\end{array}}{(\texttt{value-of}\ (\texttt{diff-exp}\ exp_1\ exp_2)\ \rho\ \sigma_0) = (\lceil \lfloor val_1 \rfloor - \lfloor val_2 \rfloor \rceil, \sigma_2)}$$

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ \rho\ \sigma_0) \\ \quad = \begin{cases}(\texttt{value-of}\ exp_2\ \rho\ \sigma_1) & \text{if } (\texttt{expval->bool}\ val_1) = \texttt{\#t} \\ (\texttt{value-of}\ exp_3\ \rho\ \sigma_1) & \text{if } (\texttt{expval->bool}\ val_1) = \texttt{\#f}\end{cases}\end{array}}$$

Nugget

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

Grammar specification

- The new grammar

$Expression$ ::= `newref (`$Expression$`)`
$\boxed{\texttt{newref-exp (exp1)}}$

$Expression$ ::= `deref (`$Expression$`)`
$\boxed{\texttt{deref-exp (exp1)}}$

$Expression$ ::= `setref (`$Expression$ `,` $Expression$`)`
$\boxed{\texttt{setref-exp (exp1 exp2)}}$

- Specification

$$\frac{(\texttt{value-of}\ exp\ \rho\ \sigma_0) = (val, \sigma_1) \qquad l \notin \mathrm{dom}(\sigma_1)}{(\texttt{value-of}\ (\texttt{newref-exp}\ exp)\ \rho\ \sigma_0) = ((\texttt{ref-val}\ l), [l{=}val]\sigma_1)}$$

$$\frac{(\texttt{value-of}\ exp\ \rho\ \sigma_0) = (l, \sigma_1)}{(\texttt{value-of}\ (\texttt{deref-exp}\ exp)\ \rho\ \sigma_0) = (\sigma_1(l), \sigma_1)}$$

$$\frac{\begin{array}{c}(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (l, \sigma_1) \\ (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) = (val, \sigma_2)\end{array}}{(\texttt{value-of}\ (\texttt{setref-exp}\ exp_1\ exp_2)\ \rho\ \sigma_0) = (\lceil 23 \rceil, [l{=}val]\sigma_2)}$$