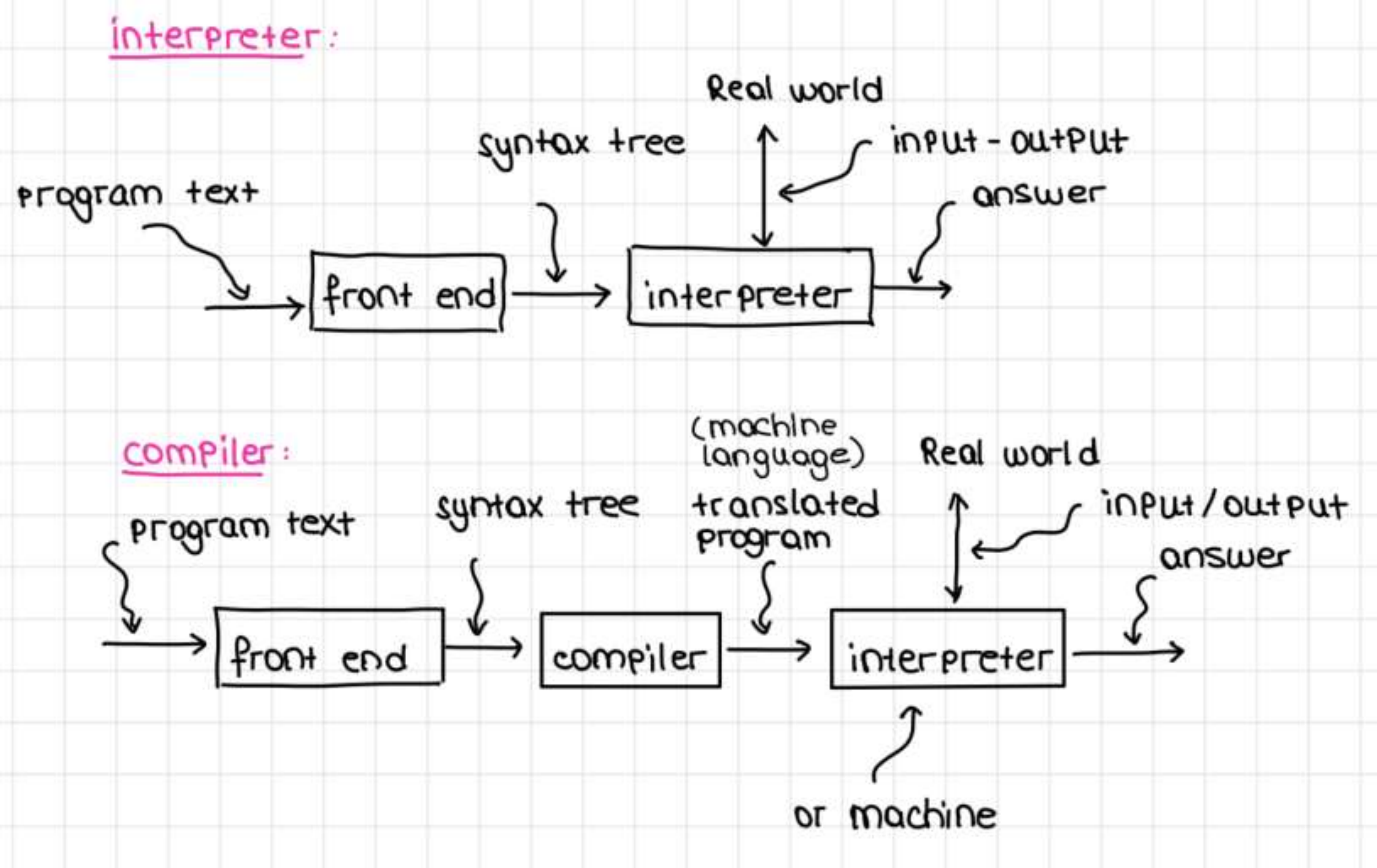
Announcements

1. Mid-semester evaluation
2. Lecture notes
3. The Quiz Let vs. Scheme

Lecture 10
Abstract Syntax, Representation, Interpretation

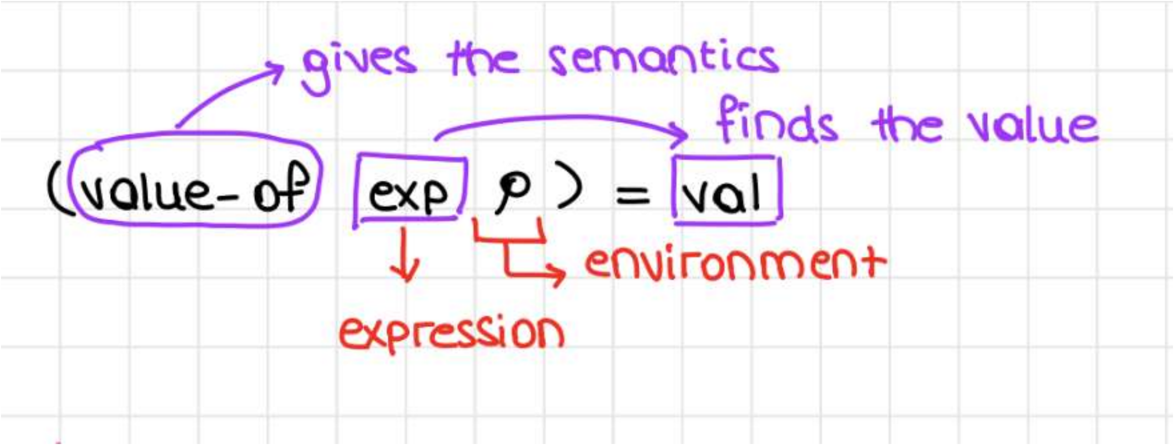
T. METIN SEZGIN

Interpreters and Compilers



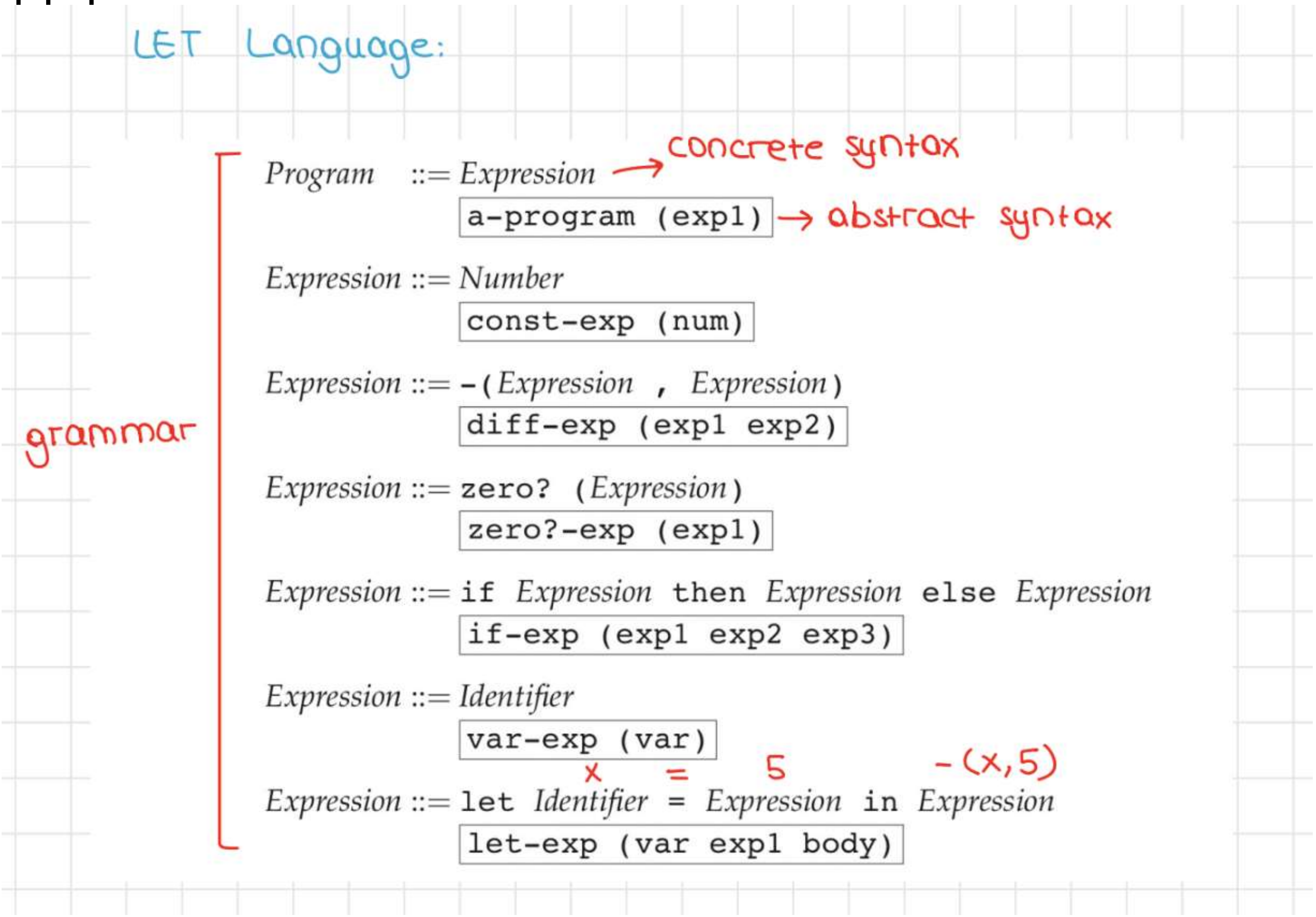
Credit: Ceren Tarim

Evaluation



Credit: Ceren Tarim

LET



LET Language:

grammar

Program ::= *Expression* → concrete syntax

`a-program (exp1)` → abstract syntax

Expression ::= *Number*

`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`

`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`

`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*

`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*

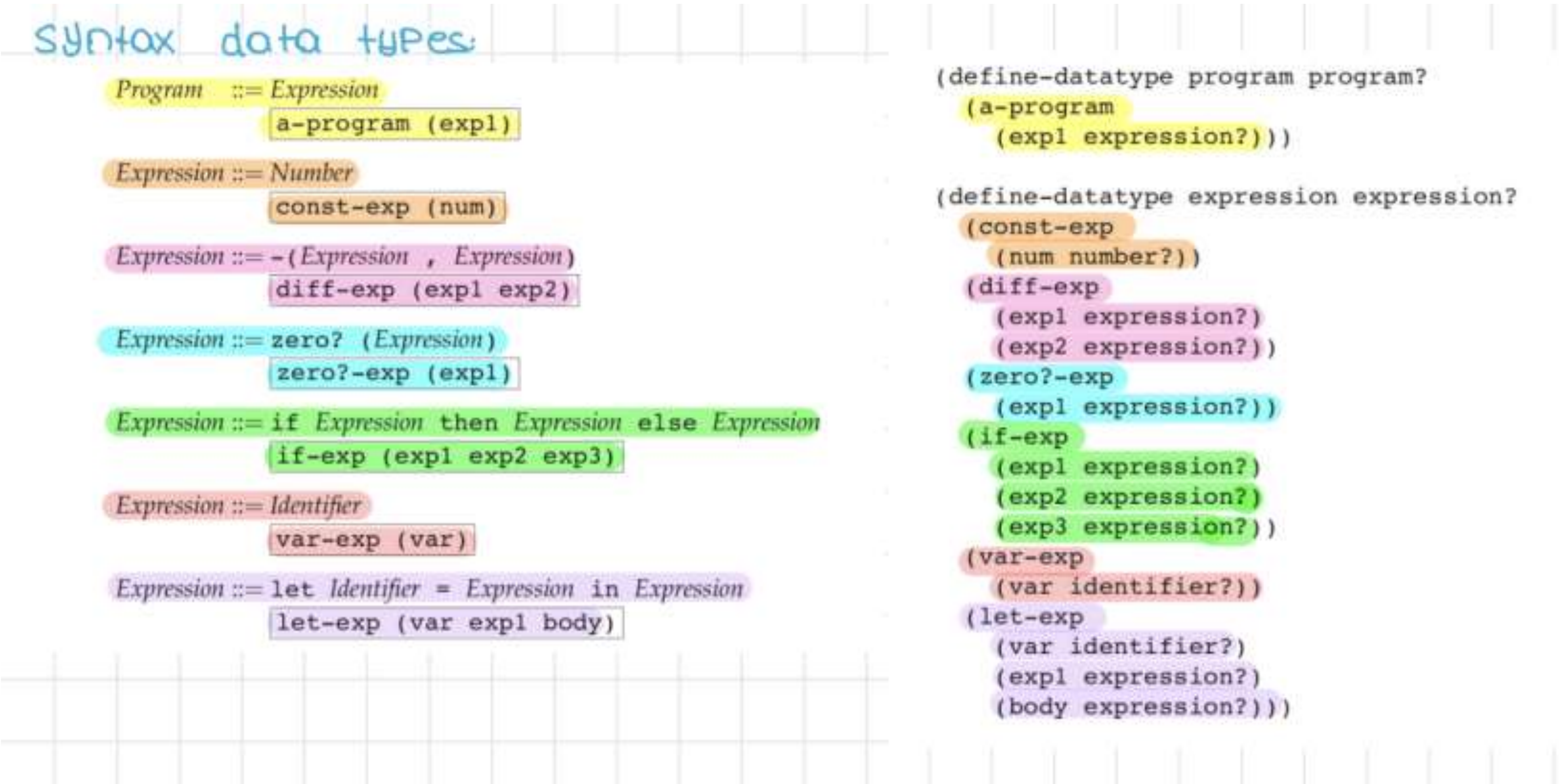
`var-exp (var)`

x = 5 -(x,5)

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*

`let-exp (var exp1 body)`

Credit: Ceren Tarim



syntax data types:

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero?` `(`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

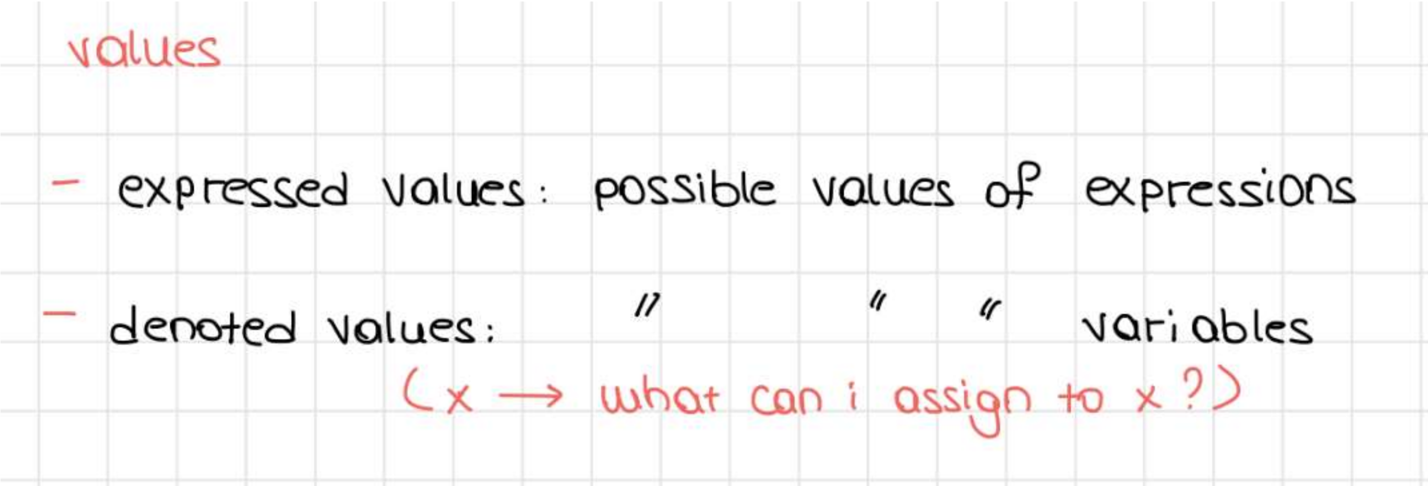
Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))

(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var identifier?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?)))
```

Credit: Ceren Tarim

Values

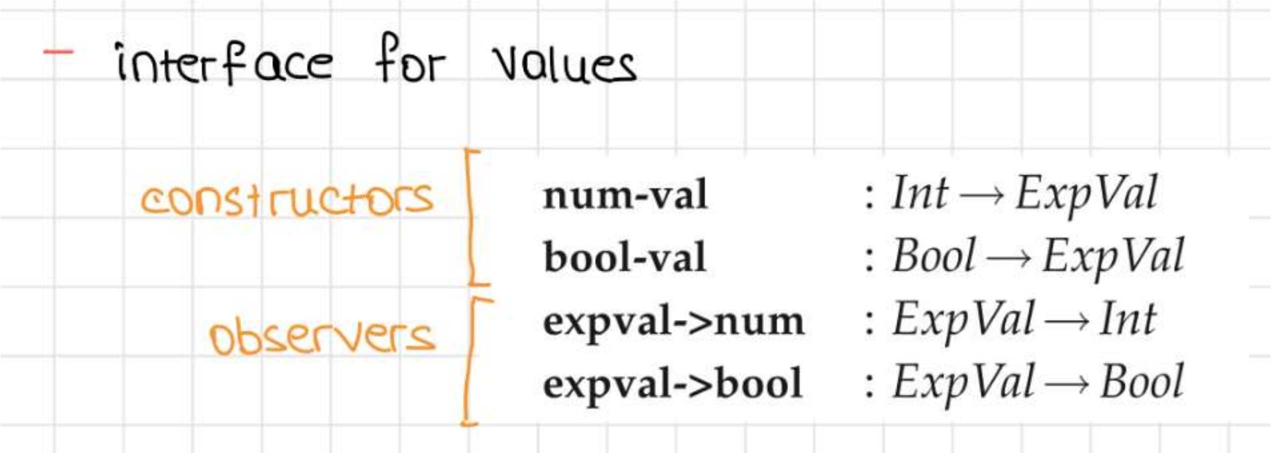


values

- expressed values: possible values of expressions
- denoted values: " " " variables

(x → what can i assign to x ?)

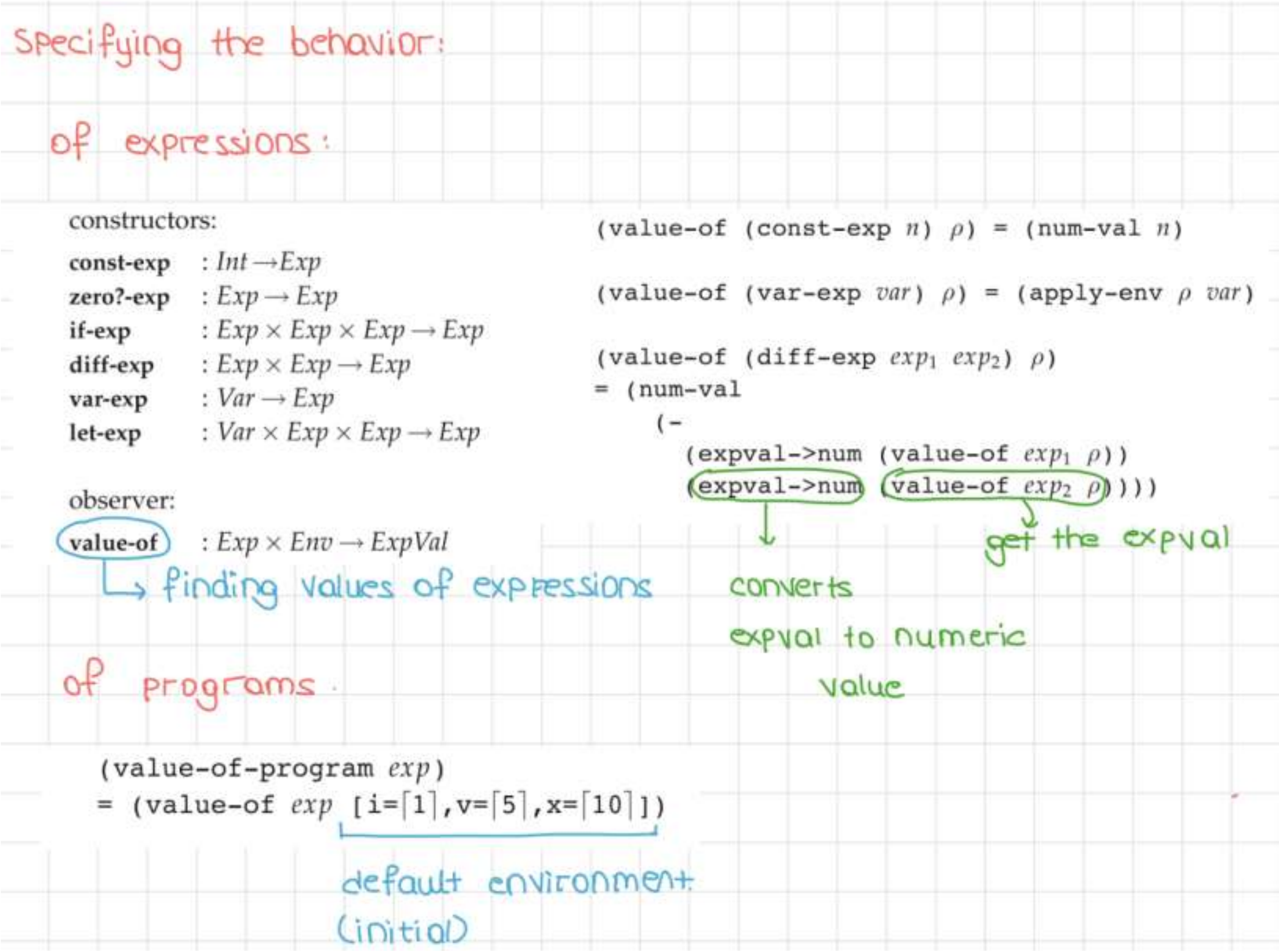
Credit: Ceren Tarim



- interface for values

constructors	**num-val**	: $Int \rightarrow ExpVal$
	bool-val	: $Bool \rightarrow ExpVal$
observers	**expval->num**	: $ExpVal \rightarrow Int$
	expval->bool	: $ExpVal \rightarrow Bool$

Credit: Ceren Tarim



Specifying the behavior:

of expressions:

constructors:

const-exp	: $Int \to Exp$
zero?-exp	: $Exp \to Exp$
if-exp	: $Exp \times Exp \times Exp \to Exp$
diff-exp	: $Exp \times Exp \to Exp$
var-exp	: $Var \to Exp$
let-exp	: $Var \times Exp \times Exp \to Exp$

observer:

value-of	: $Exp \times Env \to ExpVal$

↳ finding values of expressions

```
(value-of (const-exp n) ρ) = (num-val n)

(value-of (var-exp var) ρ) = (apply-env ρ var)

(value-of (diff-exp exp1 exp2) ρ)
= (num-val
    (-
      (expval->num (value-of exp1 ρ))
      (expval->num (value-of exp2 ρ))))
```

converts expval to numeric value

get the expval

of programs.

```
(value-of-program exp)
= (value-of exp [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉])
```

default environment (initial)

Credit: Ceren Tarim

Lecture 11
Let

T. METIN SEZGIN

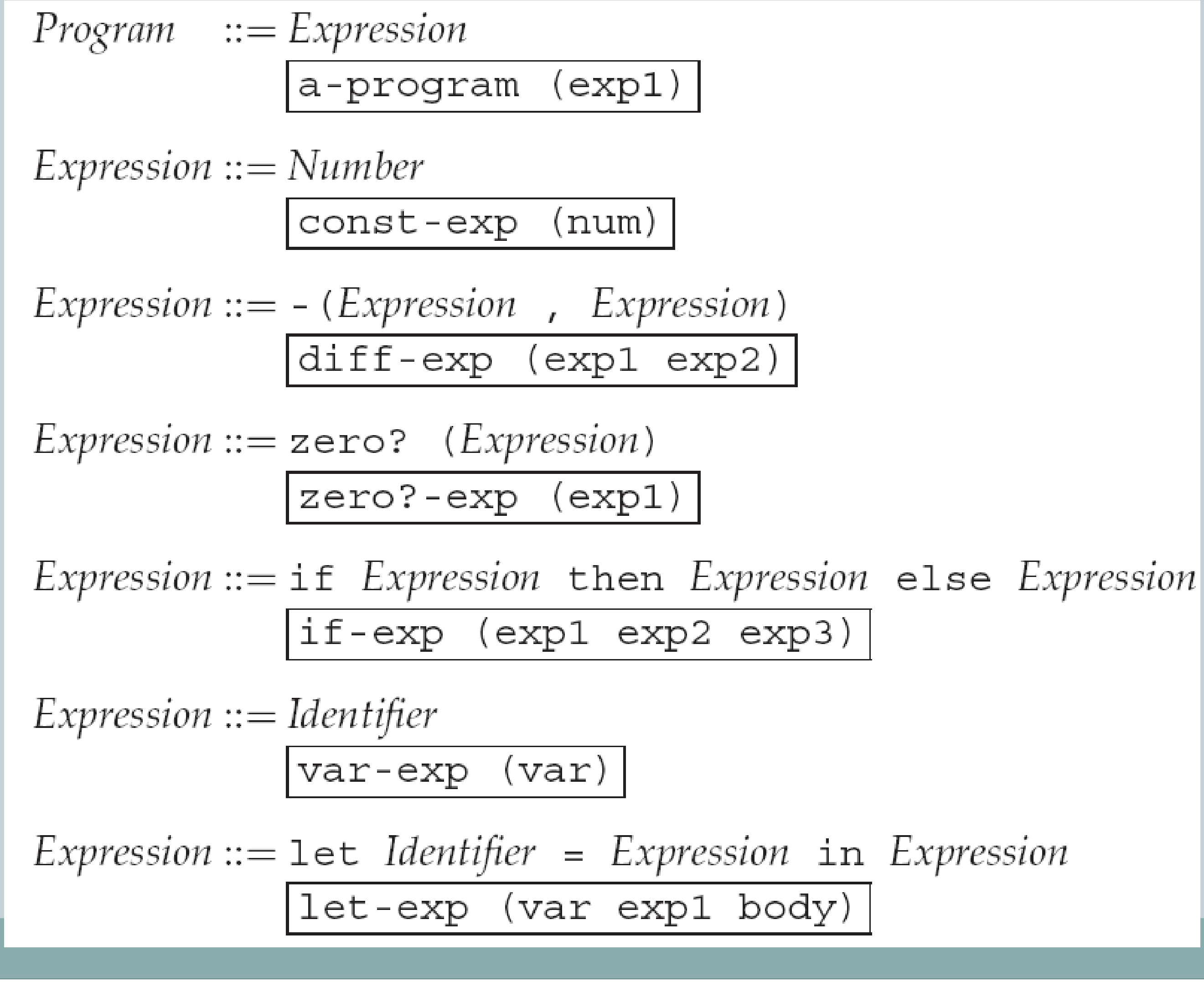
Nuggets of the lecture

- Let is a simple but expressive language
- Steps of inventing a language
- Values
- We specify the meaning of expressions first

Nugget

Let is a simple but expressive language

LET: our pet language



Program ::= *Expression*

`a-program (exp1)`

Expression ::= *Number*

`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`

`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`

`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*

`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*

`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*

`let-exp (var exp1 body)`

An example program

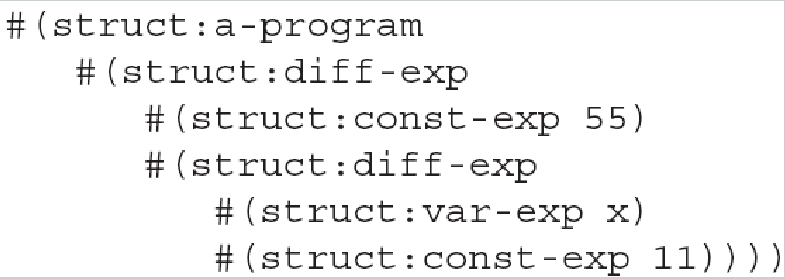
- Input

```
"-(55, -(x,11))"
```

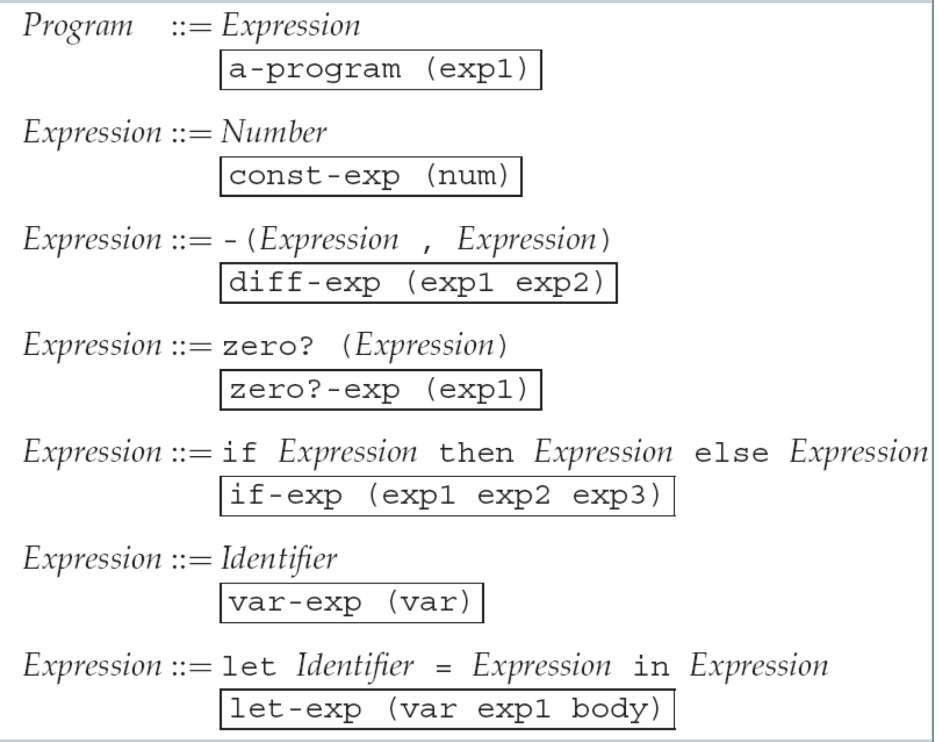
- Scanning & parsing

```
(scan&parse "-(55, -(x,11))")
```

- The AST



```
#(struct:a-program
   #(struct:diff-exp
      #(struct:const-exp 55)
      #(struct:diff-exp
         #(struct:var-exp x)
         #(struct:const-exp 11))))
```



Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero?` `(`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Nugget

Steps of inventing a language

Components of the language

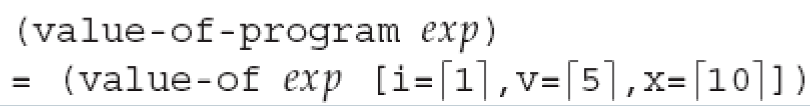
- Syntax and datatypes
- Values
- Environment
- Behavior specification
- Behavior implementation
  - Scanning
  - Parsing
  - Evaluation

Nugget

We specify the meaning of expressions first

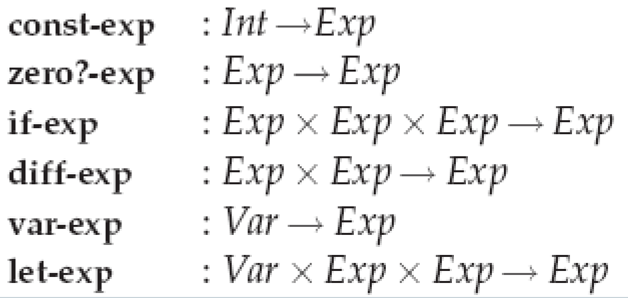
Specifying the behavior

- Programs

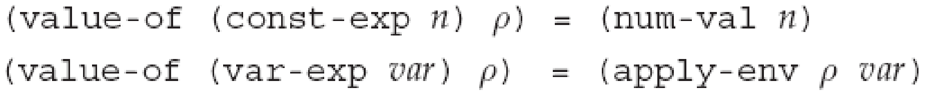


(value-of-program *exp*)
= (value-of *exp* [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉])

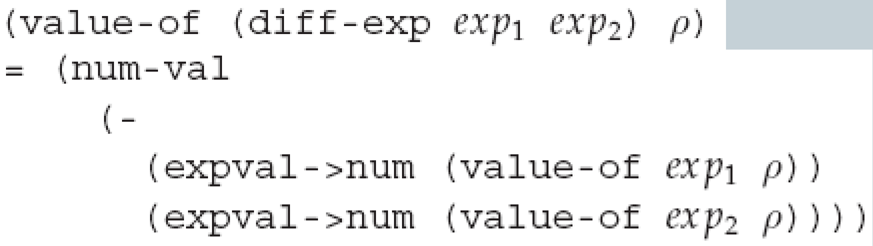
- Expressions
  - Constructors



const-exp	$: Int \rightarrow Exp$
zero?-exp	$: Exp \rightarrow Exp$
if-exp	$: Exp \times Exp \times Exp \rightarrow Exp$
diff-exp	$: Exp \times Exp \rightarrow Exp$
var-exp	$: Var \rightarrow Exp$
let-exp	$: Var \times Exp \times Exp \rightarrow Exp$

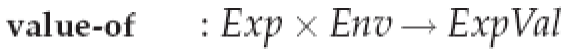


(value-of (const-exp n) ρ) = (num-val n)
(value-of (var-exp var) ρ) = (apply-env ρ var)



(value-of (diff-exp exp_1 exp_2) ρ)
= (num-val
    (-
      (expval->num (value-of exp_1 ρ))
      (expval->num (value-of exp_2 ρ))))

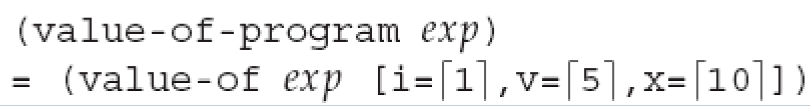
  - Observer



value-of $: Exp \times Env \rightarrow ExpVal$

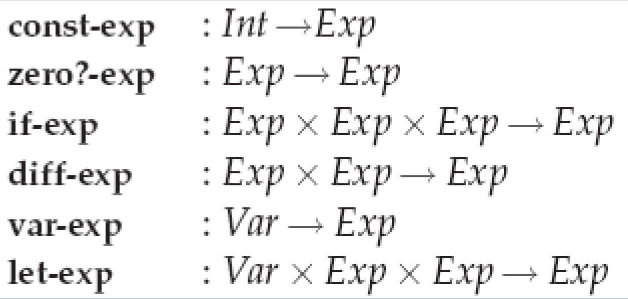
Specifying the behavior

- Programs

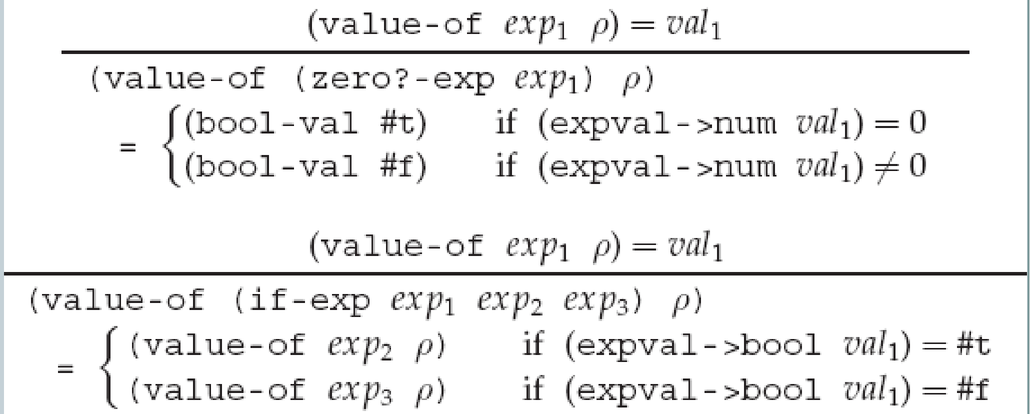


$$
\begin{array}{l}
(\texttt{value-of-program}\ exp) \\
= (\texttt{value-of}\ exp\ \texttt{[i=}\lceil\texttt{1}\rceil\texttt{,v=}\lceil\texttt{5}\rceil\texttt{,x=}\lceil\texttt{10}\rceil\texttt{]})
\end{array}
$$

- Expressions
  - Constructors



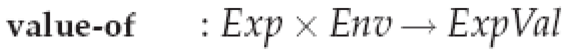
const-exp	$: Int \rightarrow Exp$
zero?-exp	$: Exp \rightarrow Exp$
if-exp	$: Exp \times Exp \times Exp \rightarrow Exp$
diff-exp	$: Exp \times Exp \rightarrow Exp$
var-exp	$: Var \rightarrow Exp$
let-exp	$: Var \times Exp \times Exp \rightarrow Exp$



$$
\frac{(\texttt{value-of}\ exp_1\ \rho) = val_1}{\begin{array}{l}(\texttt{value-of (zero?-exp}\ exp_1\texttt{)}\ \rho) \\ = \begin{cases}(\texttt{bool-val \#t}) & \text{if } (\texttt{expval->num}\ val_1) = 0 \\ (\texttt{bool-val \#f}) & \text{if } (\texttt{expval->num}\ val_1) \neq 0\end{cases}\end{array}}
$$

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

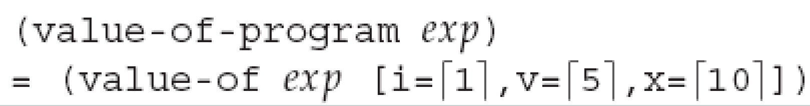
  - Observer



value-of $: Exp \times Env \rightarrow ExpVal$

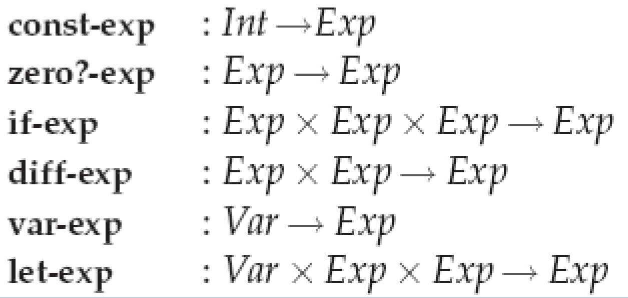
Specifying the behavior

- Programs

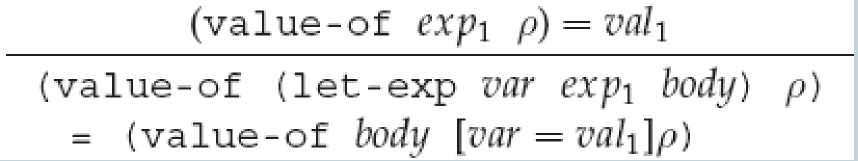


$$\begin{aligned}&\texttt{(value-of-program } exp\texttt{)}\\ &= \texttt{(value-of } exp\ \texttt{[i=}\lceil\texttt{1}\rceil\texttt{,v=}\lceil\texttt{5}\rceil\texttt{,x=}\lceil\texttt{10}\rceil\texttt{])}\end{aligned}$$

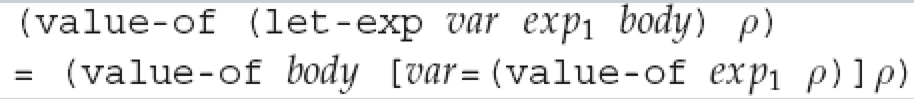
- Expressions
  - Constructors



const-exp	$: Int \rightarrow Exp$
zero?-exp	$: Exp \rightarrow Exp$
if-exp	$: Exp \times Exp \times Exp \rightarrow Exp$
diff-exp	$: Exp \times Exp \rightarrow Exp$
var-exp	$: Var \rightarrow Exp$
let-exp	$: Var \times Exp \times Exp \rightarrow Exp$

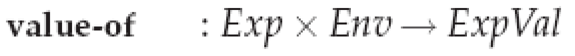


$$\frac{\texttt{(value-of } exp_1\ \rho\texttt{)} = val_1}{\begin{aligned}&\texttt{(value-of (let-exp } var\ exp_1\ body\texttt{)}\ \rho\texttt{)}\\ &\quad = \texttt{(value-of } body\ [var = val_1]\rho\texttt{)}\end{aligned}}$$



$$\begin{aligned}&\texttt{(value-of (let-exp } var\ exp_1\ body\texttt{)}\ \rho\texttt{)}\\ &= \texttt{(value-of } body\ \texttt{[}var\texttt{=(value-of } exp_1\ \rho\texttt{)]}\rho\texttt{)}\end{aligned}$$

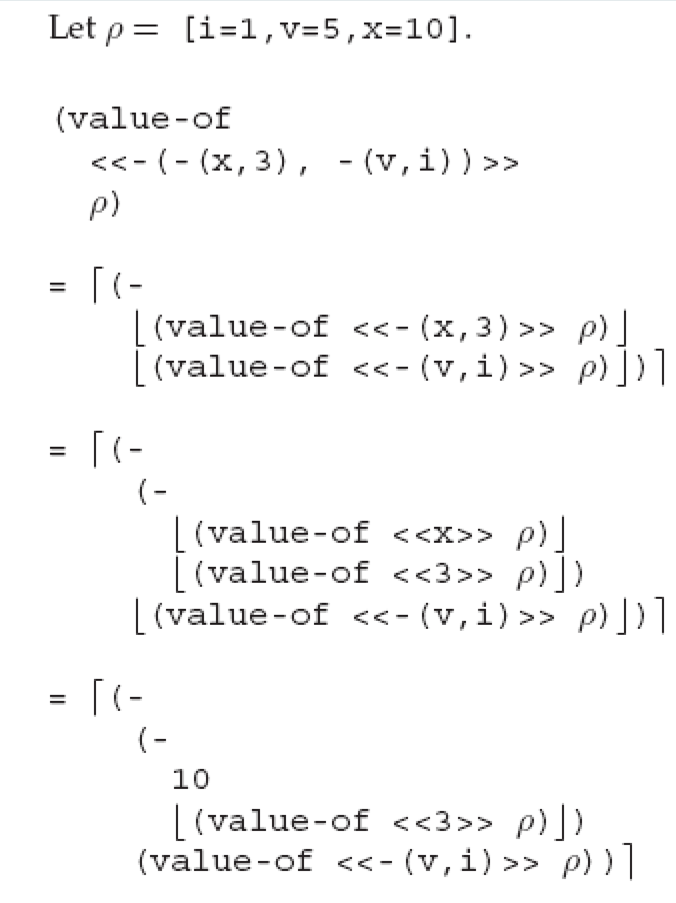
  - Observer



value-of	$: Exp \times Env \rightarrow ExpVal$

Behavior implementation

what we envision



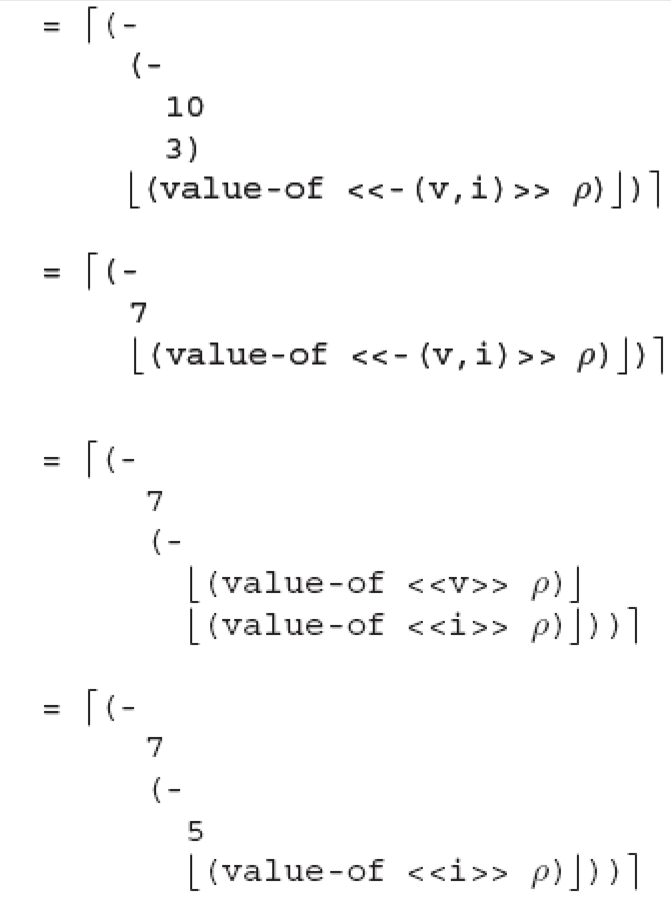
```
Let ρ = [i=1,v=5,x=10].

(value-of
  <<-(-(x,3), -(v,i))>>
  ρ)

= ⌈(-
    ⌊(value-of <<-(x,3)>> ρ)⌋
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      ⌊(value-of <<x>> ρ)⌋
      ⌊(value-of <<3>> ρ)⌋)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      10
      ⌊(value-of <<3>> ρ)⌋)
    (value-of <<-(v,i)>> ρ))⌉
```

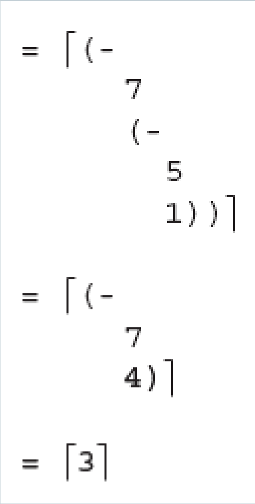


```
= ⌈(-
    (-
      10
      3)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    7
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
     7
     (-
       ⌊(value-of <<v>> ρ)⌋
       ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
     7
     (-
       5
       ⌊(value-of <<i>> ρ)⌋))⌉
```



```
= ⌈(-
     7
     (-
       5
       1))⌉

= ⌈(-
     7
     4)⌉

= ⌈3⌉
```