Announcements

1. Mid-semester evaluation
2. Lecture notes

Announcements

1. Mid-semester evaluation
2. Lecture notes

About compilation:

(steps): Analyzer → Scanning → Parsing (generate: e.g AST) → translator

Simplified by:
- lexical Analyzer (lex): Give me the Grammar for the language.
- Parser generator (yacc) yet Another compiler compiler!

Farrin Sofian

Announcements

1. Mid-semester evaluation
2. Lecture notes

type 1 Lc exp

type 2 Lc exp

type 3 Lc exp

```
(define-datatype lc-exp lc-exp?
  (var-exp
    (var identifier?))
  (lambda-exp
    (bound-var identifier?)
    (body lc-exp?))
  (app-exp
    (rator lc-exp?)
    (rand lc-exp?)))
```

Lc-exp ::= *Identifier* ← Concrete

`var-exp (var)` ← Abstract

::= (lambda (*Identifier*) *Lc-exp*)

`lambda-exp (bound-var body)`

::= (*Lc-exp* *Lc-exp*)

`app-exp (rator rand)`

our constructors

Farrin Sofian

Announcements

1. Mid-semester evaluation
2. Lecture notes

```
(define (REDUCE) op init lst)
 (if (null? lst)
  init
  (op (car lst)
   (reduce op init (cdr lst))))
        ↓
(define (add-up lst )
  (reduce + 0 lst))

listeyi azaltarak geri dönüş
                    uyguluyor
```

Taha Yasin Erel

Lecture 10
Abstract Syntax, Representation, Interpretation

T. METIN SEZGIN

Nuggets of the lecture

- Syntax is all about structure
- Semantics is all about meaning
- We can use abstract syntax to represent programs as trees
- Parsing takes a program builds a syntax tree
- Unparsing converts abstract tree to a text file
- Big picture of compilers and interpreters

Human vs. the computer

- Lambda calculus

$$\begin{aligned} \mathit{LcExp} &::= \mathit{Identifier} \\ &::= \texttt{(lambda (}\mathit{Identifier}\texttt{)}\ \mathit{LcExp}\texttt{)} \\ &::= \texttt{(}\mathit{LcExp}\ \mathit{LcExp}\texttt{)} \end{aligned}$$

- Alternative syntax

$$\begin{aligned} \mathit{Lc\text{-}exp} &::= \mathit{Identifier} \\ &::= \texttt{proc}\ \mathit{Identifier}\ \texttt{=>}\ \mathit{Lc\text{-}exp} \\ &::= \mathit{Lc\text{-}exp}\,\texttt{(}\mathit{Lc\text{-}exp}\texttt{)} \end{aligned}$$

- The computer

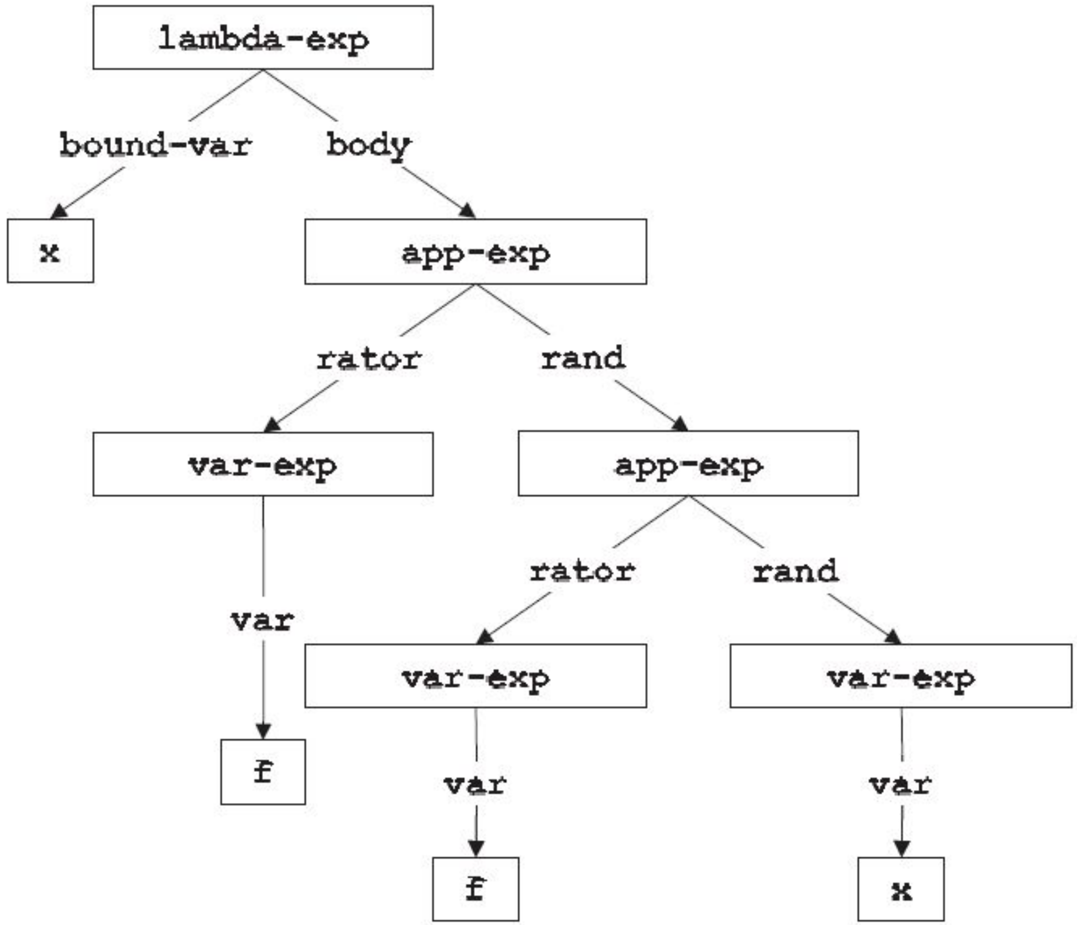
```
(define-datatype lc-exp lc-exp?
  (var-exp
    (var identifier?))
  (lambda-exp
    (bound-var identifier?)
    (body lc-exp?))
  (app-exp
    (rator lc-exp?)
    (rand lc-exp?)))
```

$$\begin{aligned} \mathit{Lc\text{-}exp} &::= \mathit{Identifier} \\ &\quad \boxed{\texttt{var-exp (var)}} \\ &::= \texttt{(lambda (}\mathit{Identifier}\texttt{)}\ \mathit{Lc\text{-}exp}\texttt{)} \\ &\quad \boxed{\texttt{lambda-exp (bound-var body)}} \\ &::= \texttt{(}\mathit{Lc\text{-}exp}\ \mathit{Lc\text{-}exp}\texttt{)} \\ &\quad \boxed{\texttt{app-exp (rator rand)}} \end{aligned}$$

Nugget

We can use abstract syntax to represent programs as trees

A specific example



Abstract syntax tree for `(lambda (x) (f (f x)))`

Nugget

Parsing takes a program builds a syntax tree

Parsing expressions

parse-expression : $SchemeVal \to LcExp$

```
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
         (lambda-exp
           (car (cadr datum))
           (parse-expression (caddr datum)))
         (app-exp
           (parse-expression (car datum))
           (parse-expression (cadr datum)))))
      (else (report-invalid-concrete-syntax datum)))))
```

Nugget

Unparsing goes in the reverse direction

"Unparsing"

unparse-lc-exp : *LcExp* $\rightarrow$ *SchemeVal*

```
(define unparse-lc-exp
  (lambda (exp)
    (cases lc-exp exp
      (var-exp (var) var)
      (lambda-exp (bound-var body)
        (list 'lambda (list bound-var)
          (unparse-lc-exp body)))
      (app-exp (rator rand)
        (list
```

The next few weeks

- Expressions
- Binding of variables
- Scoping of variables
- Environment
- Interpreters

Nugget

Semantics is all about evaluating programs, finding their “value”

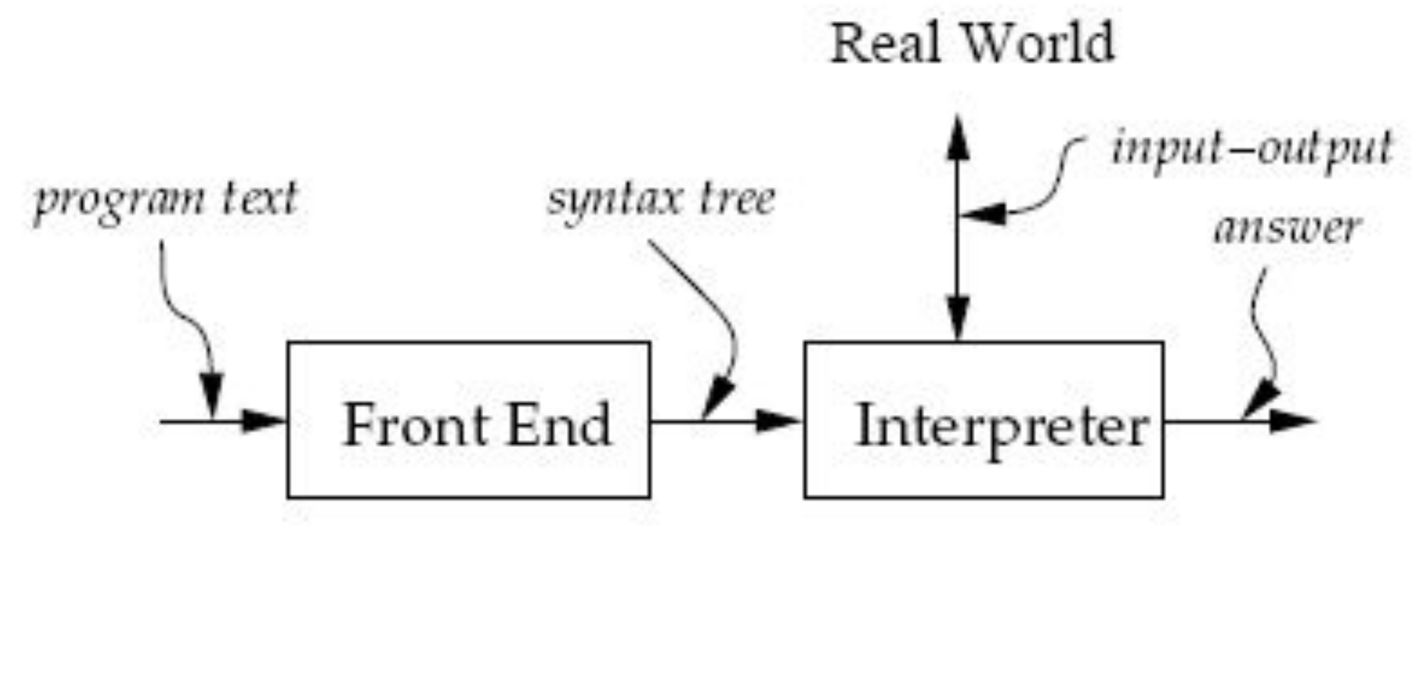
Notation

- Assertions for specification

$$(\texttt{value-of}\ \ exp\ \ \rho) = val$$

- Use rules from earlier chapters and specifications to compute values

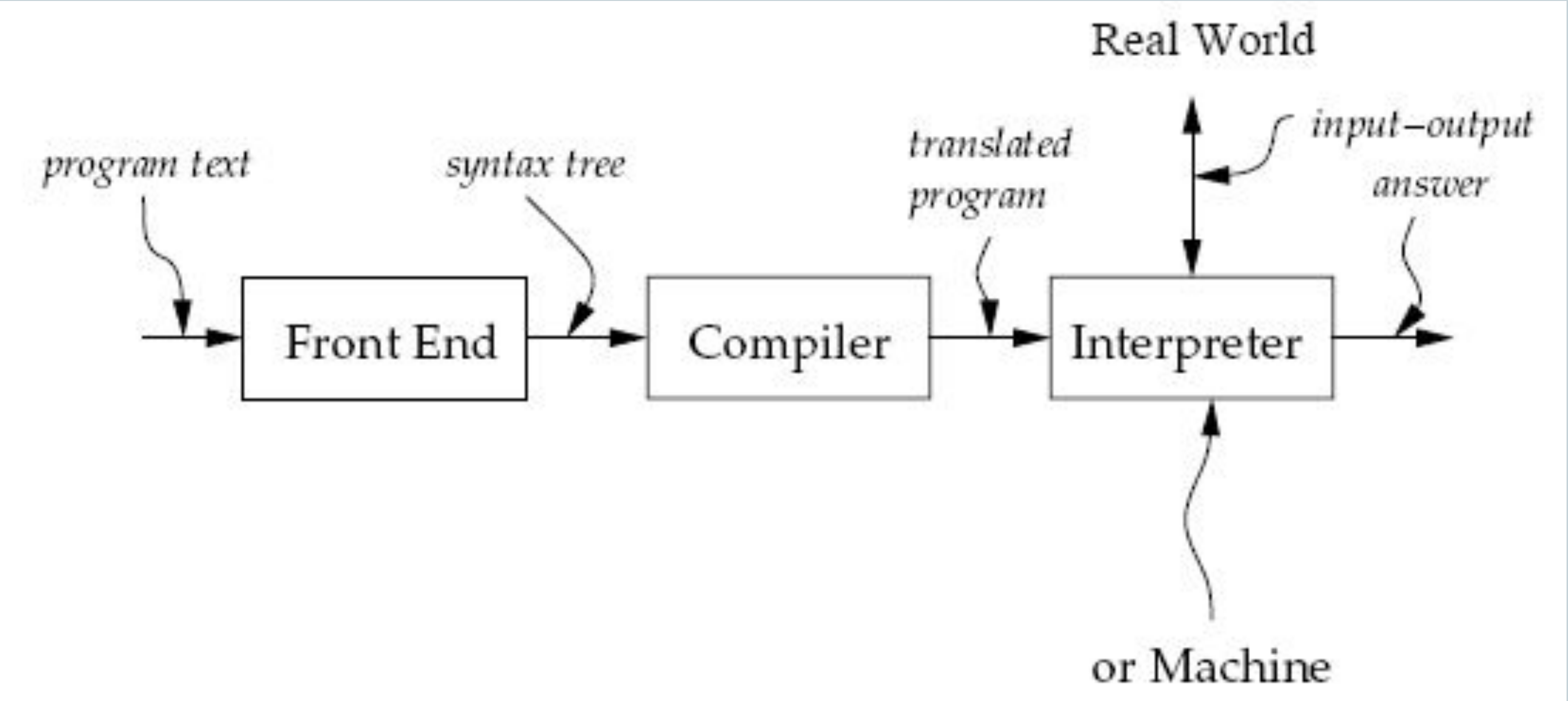
The big picture – interpreter



Execution via interpreter

Source language (defined language), implementation language (defining language), target language,

The big picture – compiler



Execution via Compiler

Source language (defined language), implementation language (defining language), target language, bytecode, virtual machine

About compilation

- Compilation
  - Analyzer
    - Scanning (lexical scanning)
      - Generates
        - Lexemes
        - Lexical items
        - Tokens
    - Parsing
      - Generates
        - AST
        - Syntactic structure
        - Grammatical structure
  - Translator
- All this work simplified
  - Lexical analyzers (lex)
  - Parser generators (yacc)
  - Use scheme 😊

```
int main()
{
        printf("hello, world");
        return 0;
}
```

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

Syntax data types

Program ::= *Expression*
$\boxed{\texttt{a-program (exp1)}}$

Expression ::= *Number*
$\boxed{\texttt{const-exp (num)}}$

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

Expression ::= `zero?` (*Expression*)
$\boxed{\texttt{zero?-exp (exp1)}}$

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
$\boxed{\texttt{if-exp (exp1 exp2 exp3)}}$

Expression ::= *Identifier*
$\boxed{\texttt{var-exp (var)}}$

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
$\boxed{\texttt{let-exp (var exp1 body)}}$

```
(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?)))
```

Nugget

Values

Values

- Set of values manipulated by the program
  - Expressed values
    - Possible values of expressions
  - Denoted values
    - Possible values of variables

$$ExpVal = Int + Bool$$
$$DenVal = Int + Bool$$

- Interface for values
  - Constructors
  - Observers

num-val	$: Int \to ExpVal$
bool-val	$: Bool \to ExpVal$
expval->num	$: ExpVal \to Int$
expval->bool	$: ExpVal \to Bool$

Environments

- Same model of environment from before

  - ρ ranges over environments.
  - [] denotes the empty environment.
  - $[var = val]\rho$ denotes `(extend-env` var val ρ`)`.
  - $[var_1 = val_1, var_2 = val_2]\rho$ abbreviates $[var_1 = val_1]([var_2 = val_2]\rho)$, etc.
  - $[var_1 = val_1, var_2 = val_2, \ldots]$ denotes the environment in which the value of var_1 is val_1, etc.

- Use

```
[x=3]
 [y=7]
  [u=5]ρ
```

to abbreviate

```
(extend-env 'x 3
  (extend-env 'y 7
    (extend-env 'u 5 ρ)))
```

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	$: Int \to Exp$
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	$: Var \times Exp \times Exp \to Exp$

```
(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))))
```

  - Observer

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

Specifying the behavior

- Programs

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

- 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	$: Var \times Exp \times Exp \to Exp$

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

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

  - Observer

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

Specifying the behavior

- Programs

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

- 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	$: Var \times Exp \times Exp \to Exp$

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

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

  - Observer

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

Behavior implementation

what we envision

Let $\rho =$ `[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⌉
```