

Announcements



1. Most of the code used in the class at the EOPL web site: <https://eopl3.com>

Lecture 9

Representation Strategies for Data Types



T. METIN SEZGIN

Lecture notes: Eda Guven



we can use unary representation

scheme number representation

big num rep.

} all will work w/

all we need to change

is zero?

successor

predecessor

environment (it allows us to store variable value pairs)

we can do representation 2 ways → data structure rep.
↳ procedural rep.

The general form of `define-datatype`



the general form of `define-datatype`

```
(define-datatype type-name type-pred-name  
  { (var-name { (field-name predicate)* } ) }
```

ex:

```
LcExp ::= Identifier  
       ::= (lambda (Identifier) LcExp)  
       ::= (LcExp LcExp)
```

```
(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?)))
```

Example uses of define-datatype

ex:

$S\text{-list} ::= (\{S\text{-exp}\}^*)$
 $S\text{-exp} ::= \text{Symbol} \mid S\text{-list}$



$S\text{-list} ::= ()$
 $::= (S\text{-exp} . S\text{-exp})$
 $::= \text{symbol} \mid S\text{-list}$

```
(define-datatype s-list s-list?  
  (empty-s-list)  
  (non-empty-s-list  
    (first s-exp?)  
    (rest s-list?)))
```

```
(define-datatype s-exp s-exp?  
  (symbol-s-exp  
    (sym symbol?))  
  (s-list-s-exp  
    (slst s-list?)))
```

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

```
LcExp ::= Identifier  
      ::= (lambda (Identifier) LcExp)  
      ::= (LcExp LcExp)
```

- Alternative syntax

```
Lc-exp ::= Identifier  
       ::= proc Identifier => Lc-exp  
       ::= Lc-exp (Lc-exp)
```

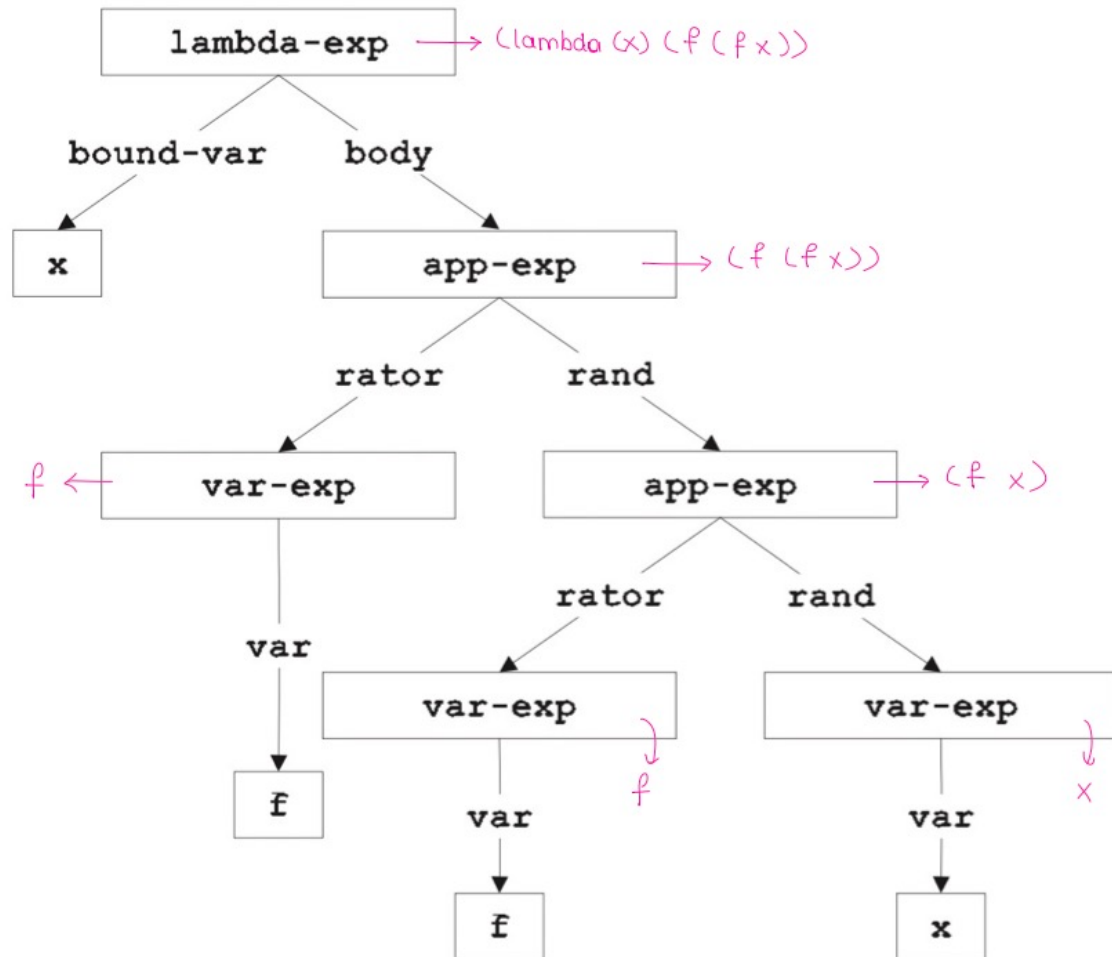
- The computer

```
(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?)))
```

```
Lc-exp ::= Identifier → human syntax  
       var-exp (var) → abstract syntax  
       ::= (lambda (Identifier) Lc-exp)  
       lambda-exp (bound-var body)  
       ::= (Lc-exp Lc-exp)  
       app-exp (rator rand)
```

Abstract Syntax Tree

abstract syntax tree: $(\text{lambda } (x) (f (f x)))$



Parsing and Unparsing

parsing: text file \rightarrow syntax tree

unparsing: syntax tree \rightarrow text file

```
LcExp ::= Identifier
       ::= (lambda (Identifier) LcExp)
       ::= (LcExp LcExp)
```

parse-expression : *SchemeVal* \rightarrow *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))))
```

Handwritten annotations:
- \rightarrow identifier (pointing to `(car (cadr datum))`)
- \rightarrow LcExp (pointing to `(parse-expression (caddr datum))`)
- \rightarrow LcExp (pointing to `(parse-expression (car datum))`)
- \rightarrow LcExp (pointing to `(parse-expression (cadr datum))`)

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
          (unparse-lc-exp rator)
          (unparse-lc-exp rand))))))
```

Handwritten annotations:
- \rightarrow identifier (pointing to `bound-var`)
- \rightarrow LcExp (pointing to `body`)
- \rightarrow LcExp (pointing to `rator`)
- \rightarrow LcExp (pointing to `rand`)

Lecture 10

Abstract Syntax, Representation, Interpretation



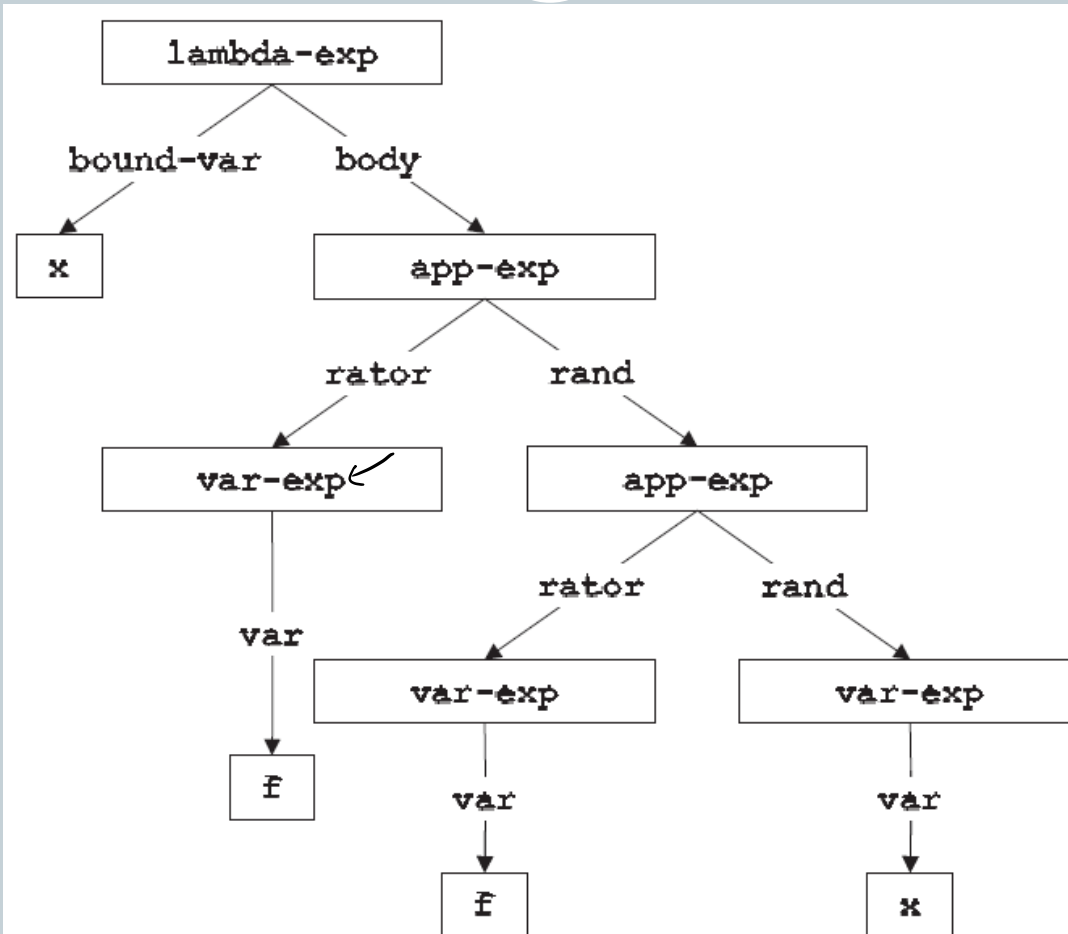
T. METIN SEZGIN

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* \rightarrow *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

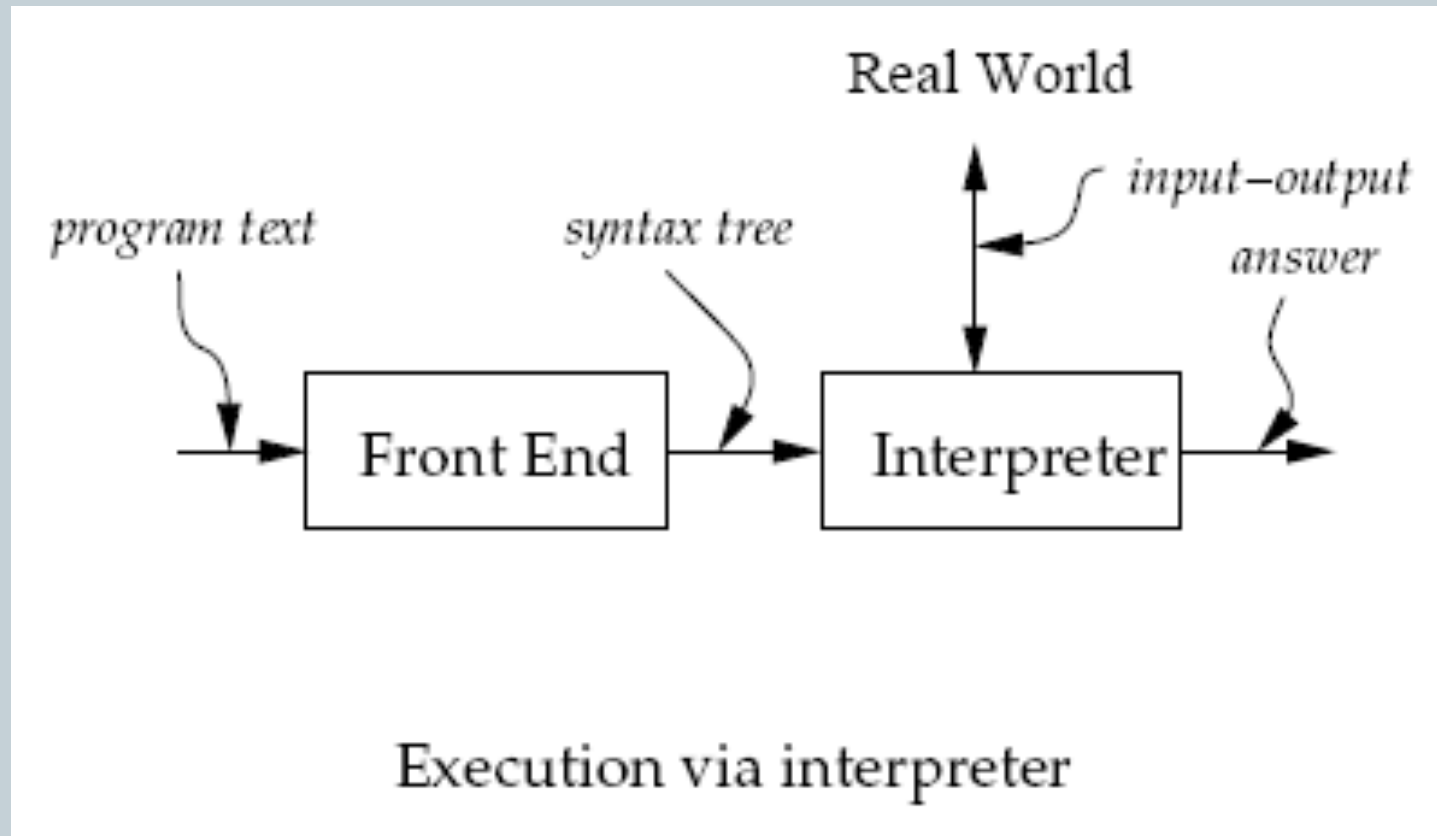


- Assertions for specification

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

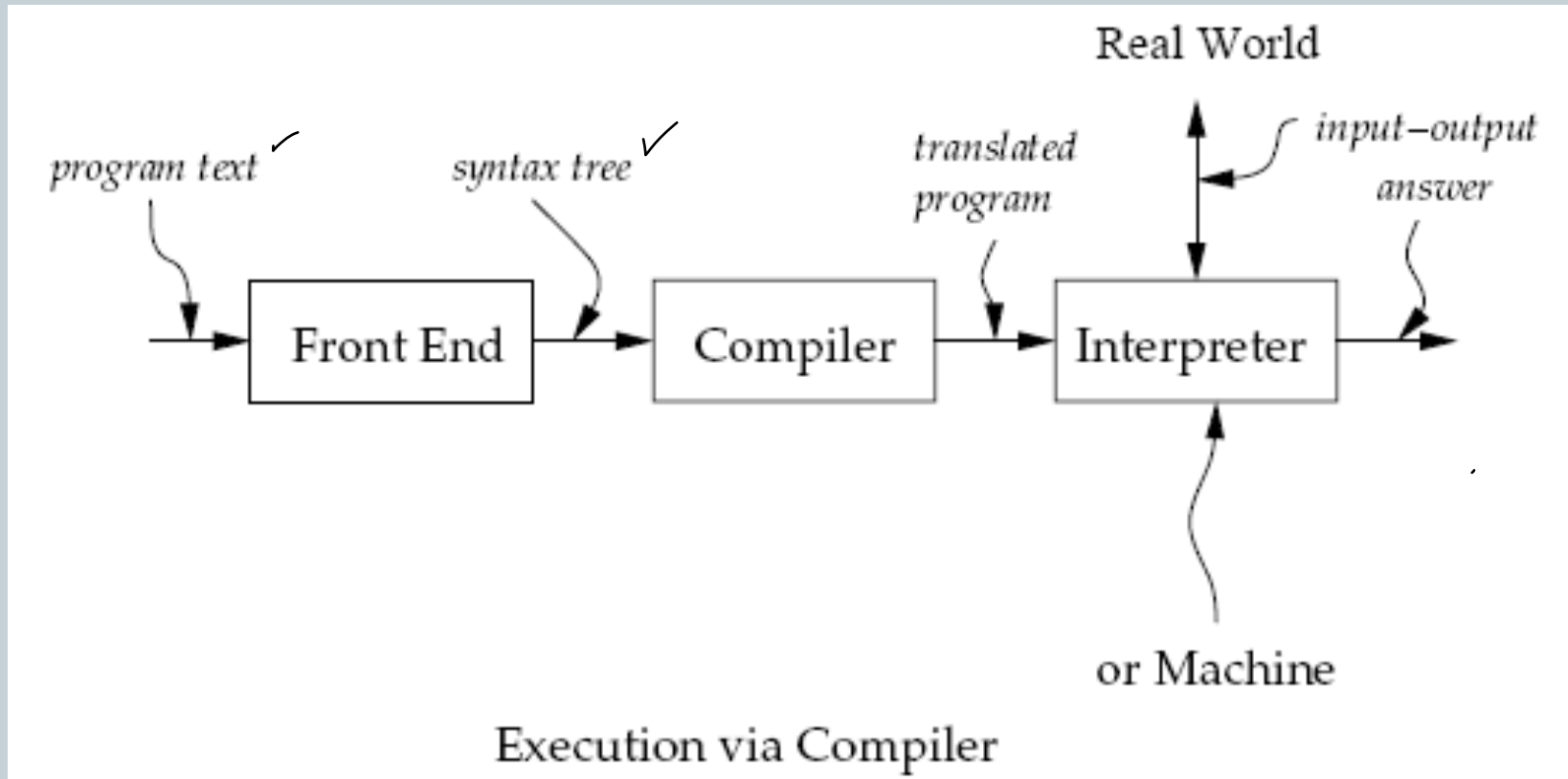
- Use rules from earlier chapters and specifications to compute values

The big picture – interpreter



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

The big picture – 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;  
}
```

Nugget



Evaluating programs, requires
understanding the expressions of
the 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)`