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

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:

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

$$\Downarrow$$

S-list ::= ()
::= (S-exp . S-exp)
::= symbol | S-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?)))
```

Fatma Ceren Tarim

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?)))
```

$\mathit{Lc\text{-}exp} ::= \mathit{Identifier}$ → human syntax

$\boxed{\texttt{var-exp (var)}}$ → abstract syntax

$::= \texttt{(lambda (}\mathit{Identifier}\texttt{)}\ \mathit{Lc\text{-}exp}\texttt{)}$

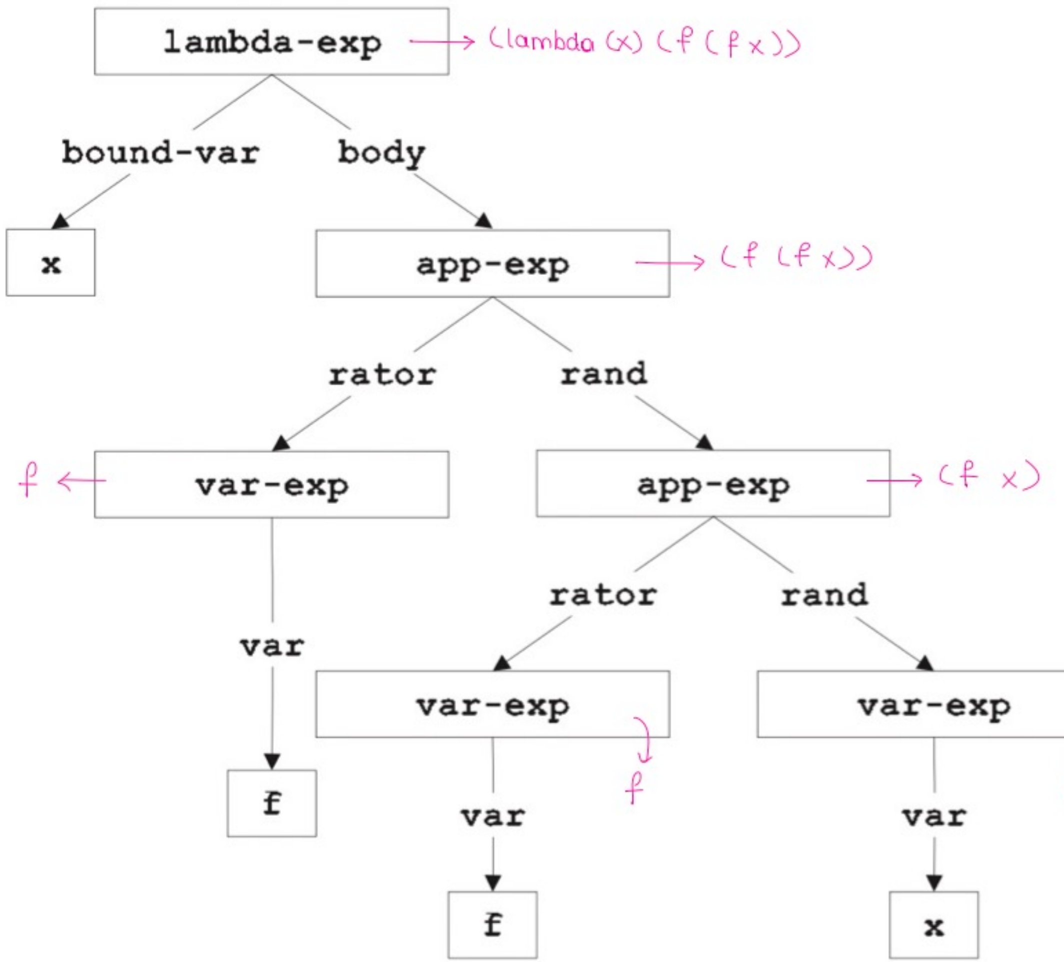
$\boxed{\texttt{lambda-exp (bound-var body)}}$

$::= \texttt{(}\mathit{Lc\text{-}exp}\ \mathit{Lc\text{-}exp}\texttt{)}$

$\boxed{\texttt{app-exp (rator rand)}}$

Abstract Syntax Tree

abstract syntax tree: (lambda (x) (f (f x)))



Parsing and Unparsing

parsing: text file → syntax tree

unparsing: syntax tree → text file

LcExp ::= *Identifier*
 ::= (lambda (*Identifier*) *LcExp*)
 ::= (*LcExp* *LcExp*)

parse-expression : *SchemeVal* → *LcExp*

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

unparse-lc-exp : *LcExp* → *SchemeVal*

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

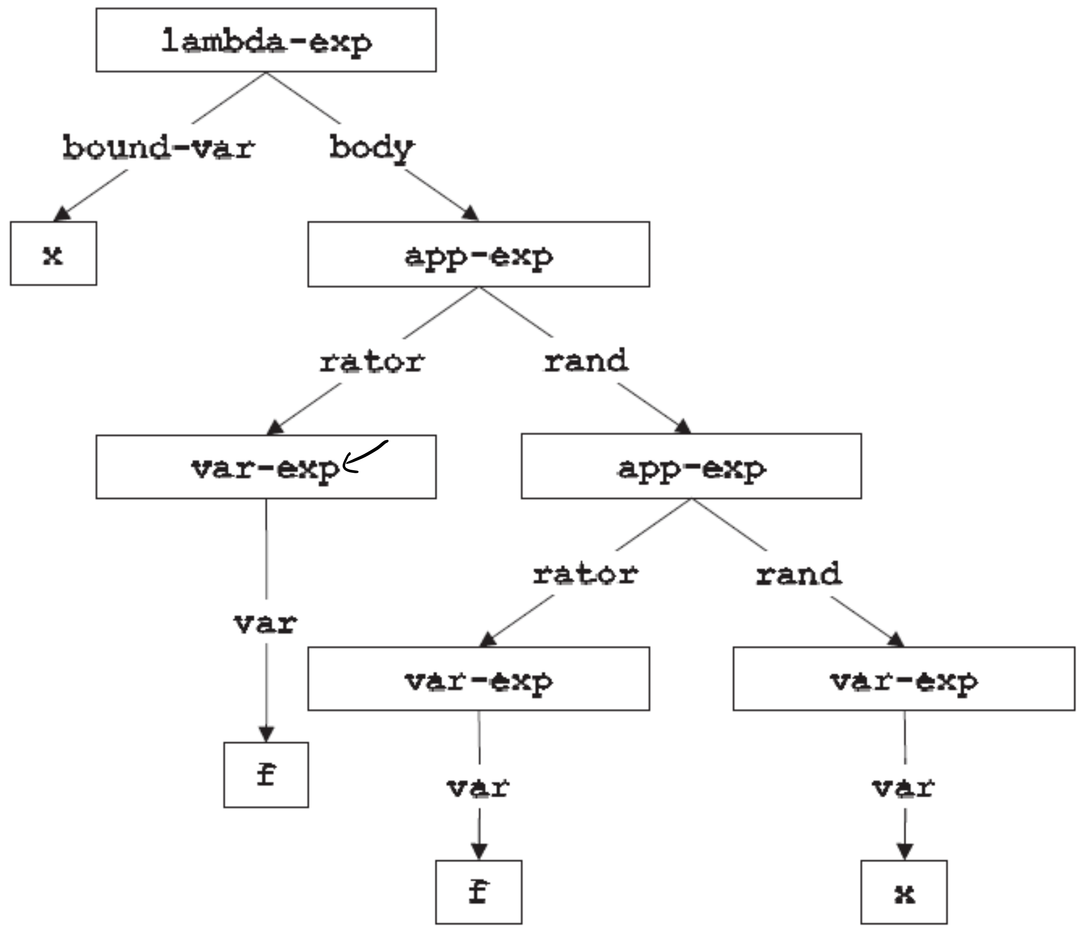
Lecture 10
Abstract Syntax, Representation, Interpretation

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* → *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"

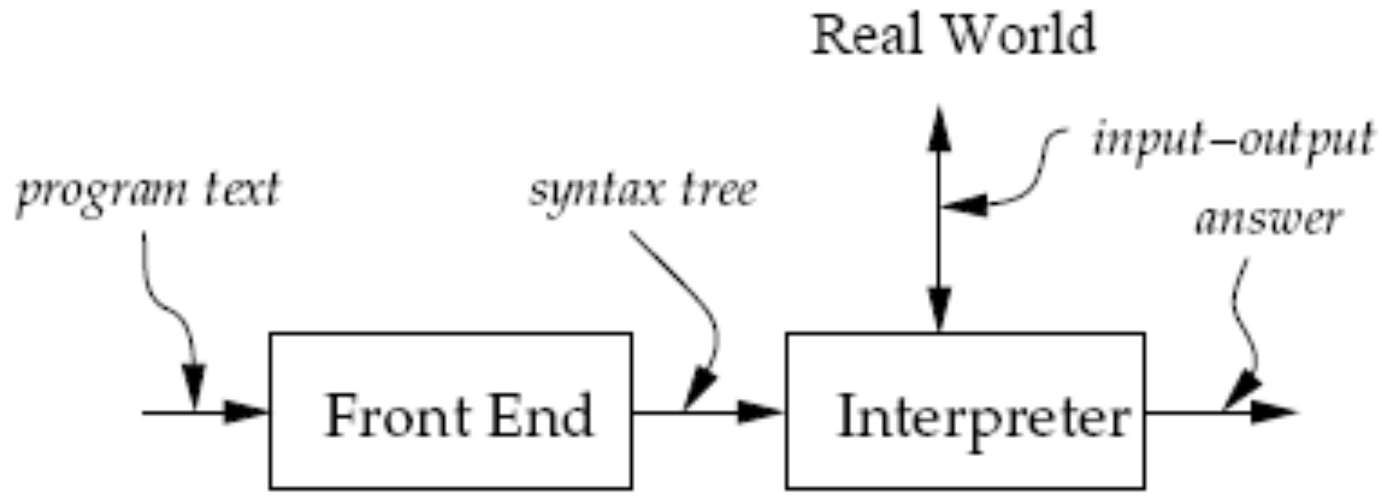
Notation

- Assertions for specification

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

- Use rules from earlier chapters and specifications to compute values

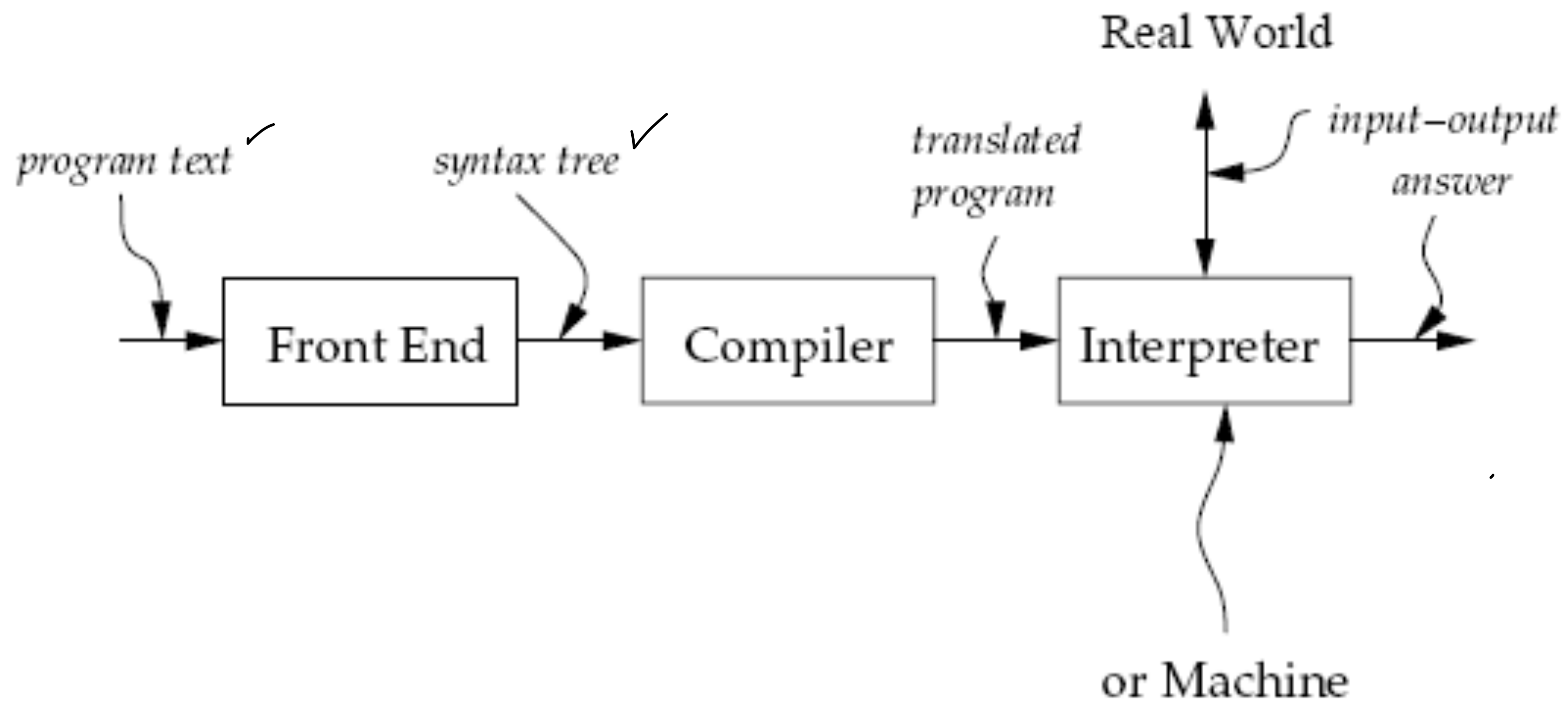
The big picture – interpreter



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 ☺

```c
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*
$\boxed{\texttt{a-program (exp1)}}$

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*
$\boxed{\texttt{let-exp (var exp1 body)}}$