Announcements

1. We'll have 4 projects total
2. Let me know if you would like a 5th project setup

1

1

Lecture 10

Representation Strategies for Data Types

T. METIN SEZGIN

2

The Environment Interface

- Environment
  - Function that maps variables to values

$$\{(var_1, val_1), \ldots, (var_n, val_n)\}$$

- The interface

$$\begin{array}{ll} \texttt{(empty-env)} & = \lceil \emptyset \rceil \\ \texttt{(apply-env } \lceil f \rceil \ var\texttt{)} & = f(var) \\ \texttt{(extend-env } var \ v \ \lceil f \rceil\texttt{)} & = \lceil g \rceil, \\ & \text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases} \end{array}$$

3

Data Structure Representation

- The interface
  - Constructors
  - Observers

$$\begin{array}{ll} \texttt{(empty-env)} & = \lceil \emptyset \rceil \\ \texttt{(apply-env } \lceil f \rceil \ var\texttt{)} & = f(var) \\ \texttt{(extend-env } var \ v \ \lceil f \rceil\texttt{)} & = \lceil g \rceil, \\ & \text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases} \end{array}$$

- For example

```
(define e
  (extend-env 'd 6
    (extend-env 'y 8
      (extend-env 'x 7
        (extend-env 'y 14
          (empty-env))))))
```

$e(\mathrm{d}) = 6, e(\mathrm{x}) = 7, e(\mathrm{y}) = 8$

- The grammar

Env-exp ::= `(empty-env)`
::= `(extend-env` *Identifier Scheme-value Env-exp*`)`

4

Implementation

```
Env = (empty-env) | (extend-env Var SchemeVal Env)
Var = Sym

empty-env : () → Env
(define empty-env
  (lambda () (list 'empty-env)))

extend-env : Var × SchemeVal × Env → Env
(define extend-env
  (lambda (var val env)
    (list 'extend-env var val env)))

apply-env : Env × Var → SchemeVal
(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
      ((eqv? (car env) 'extend-env)
       (let ((saved-var (cadr env))
             (saved-val (caddr env))
             (saved-env (cadddr env)))
         (if (eqv? search-var saved-var)
           saved-val
           (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))
```

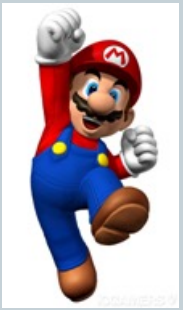
5

Procedural Representation

```
Env = Var → SchemeVal

empty-env : () → Env
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))

extend-env : Var × SchemeVal × Env → Env
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var)))))

apply-env : Env × Var → SchemeVal
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

6

The general form of `define-datatype`



```
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (bvar symbol?)
    (bval expval?)
    (saved-env environment?))
  (extend-env-rec
    (id symbol?)
    (bvar symbol?)
    (body expression?)
    (saved-env environment?)))
```

```
(define-datatype type-name type-predicate-name
  {(variant-name {(field-name predicate)}*)}+)
```

7

Example uses of `define-datatype`

- Lets define a “triple” structure using racket

Depending on how you look at it, **Racket** is

- a *programming language*—a dialect of Lisp and a descendant of Scheme;

  > See Dialects of Racket and Scheme for more information on other dialects of Lisp and how they relate to Racket.

- a *family* of programming languages—variants of Racket, and more; or
- a set of *tools*—for using a family of programming languages.

Where there is no room for confusion, we use simply *Racket*.

Racket's main tools are

- **racket**, the core compiler, interpreter, and run-time system;
- **DrRacket**, the programming environment; and
- **raco**, a command-line tool for executing **Ra**cket **co**mmands that install packages, build libraries, and more.

8

Example uses of `define-datatype`

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?)))
```

9

Nugget

We can represent any data structure easily using define-datatype

10

Lecture 11
Abstract Syntax, Representation, Interpretation

T. METIN SEZGIN

11

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

12

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

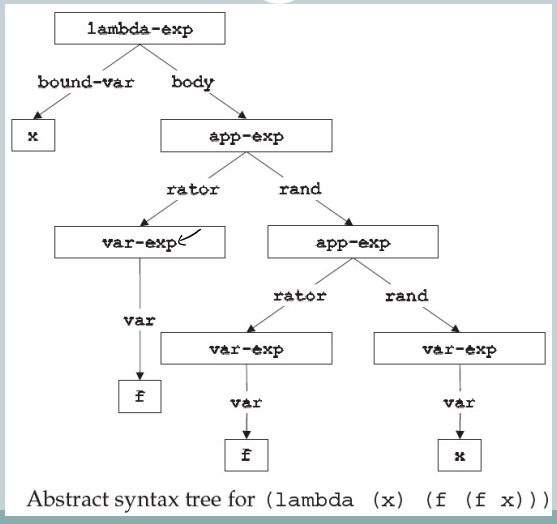
13

Nugget

We can use abstract syntax to represent programs as trees

14

A specific example



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

15

Nugget

Parsing takes a program builds a syntax tree

16

Parsing expressions

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))
           (parse-expression (caddr datum)))
         (app-exp
           (parse-expression (car datum))
           (parse-expression (cadr datum)))))
      (else (report-invalid-concrete-syntax datum)))))
```

17

Nugget

Unparsing goes in the reverse direction

18

"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
```

19

The next few weeks

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

20

Nugget

Semantics is all about evaluating programs, finding their "value"

21

Notation

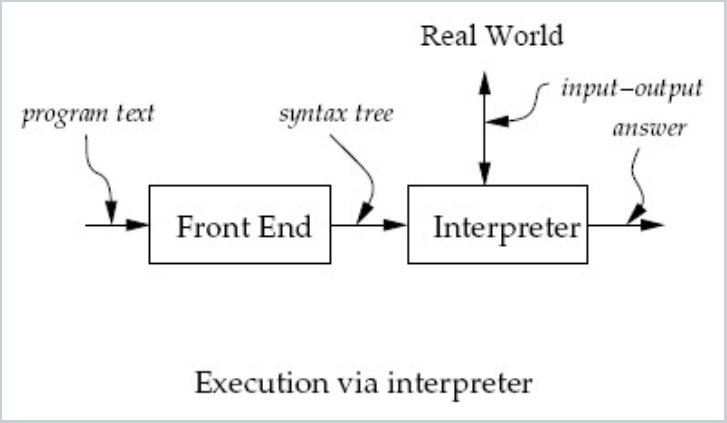
- Assertions for specification

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

- Use rules from earlier chapters and specifications to compute values

22

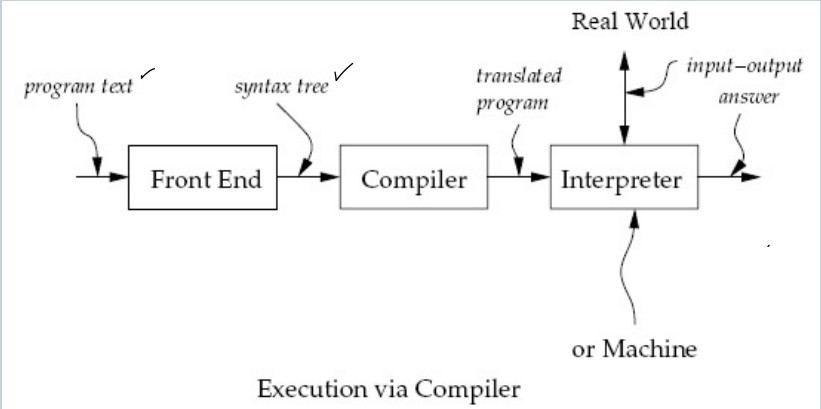
The big picture – interpreter



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

23

The big picture – compiler



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

24

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;
}
```

25

Nugget

Evaluating programs, requires understanding the expressions of the language

26

Lecture 12
Let

T. METIN SEZGIN

27

Nuggets of the lecture

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

28

Nugget

Let is a simple but expressive language

29

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)`

30

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)`

31

Nugget

Steps of inventing a language

32

Components of the language

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

33

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?)))
```

34

Nugget

Values

35

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	: $ExpVal \to Bool$

36

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)))
```

37

Nugget

We specify the meaning of expressions first

38

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	$: Exp \times Env \to ExpVal$

39

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

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

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

  - Observer

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

40

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

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

```
(value-of (let-exp var exp1 body) ρ)
= (value-of body [var=(value-of exp1 ρ)]ρ)
```

  - Observer

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

41

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⌉
```

42