Announcements

1. Midterm coming
2. PS'es announced earlier (+1 day)
3. Extra day (+1 day)

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

$$
\begin{array}{rl}
\textit{Env-exp} ::= & \texttt{(empty-env)} \\
::= & \texttt{(extend-env } \textit{Identifier} \ \textit{Scheme-value} \ \textit{Env-exp}\texttt{)}
\end{array}
$$

4

Implementation

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

empty-env : $() \to Env$

```
(define empty-env
  (lambda () (list 'empty-env)))
```

extend-env : $Var \times SchemeVal \times Env \to Env$

```
(define extend-env
  (lambda (var val env)
    (list 'extend-env var val env)))
```

apply-env : $Env \times Var \to SchemeVal$

```
(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

$Env = Var \to SchemeVal$

empty-env : $() \to Env$

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))
```

extend-env : $Var \times SchemeVal \times Env \to Env$

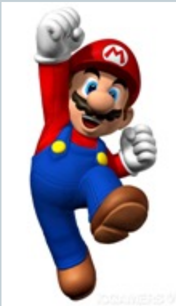
```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var)))))
```

apply-env : $Env \times Var \to SchemeVal$

```
(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 **Rac**ket **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

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

- Alternative syntax

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

- 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?)))
```

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

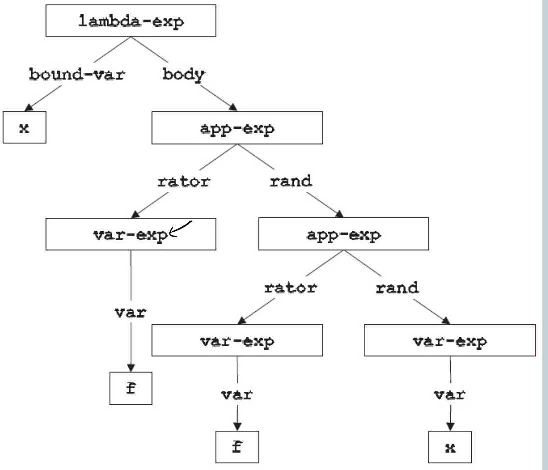
13

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

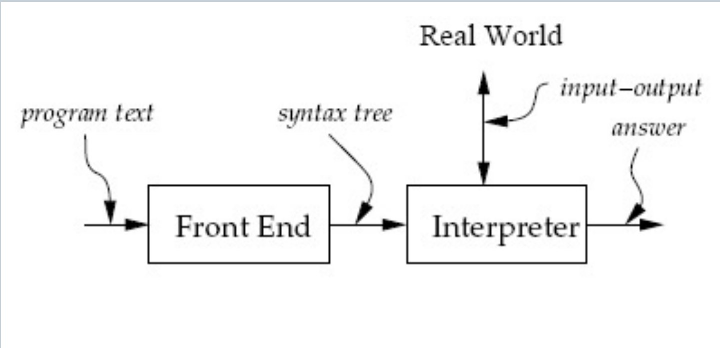
- Assertions for specification

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

- Use rules from earlier chapters and specifications to compute values

22

The big picture – interpreter

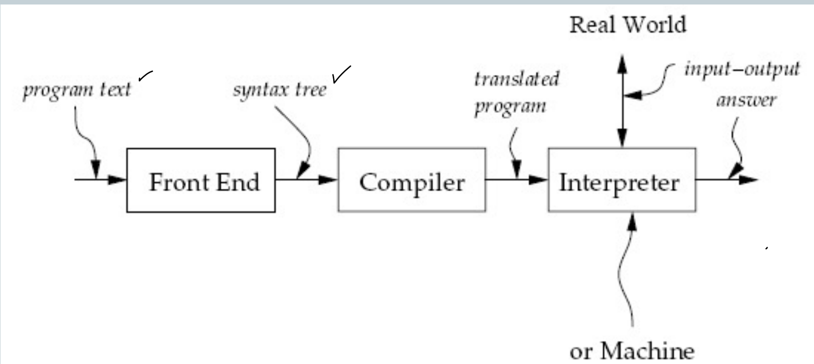


Execution via interpreter

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

23

The big picture – compiler



Execution via 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

LET: our pet language

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

27

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

28