CS450

Structure of Higher Level Languages

Lecture 15: Exercises

Tiago Cogumbreiro

Press arrow keys ← → to change slides.

Solving the homework assignment

Solving the homework assignment

1. **learn what each symbol means for each language:** relate the AST to the data-structures (Exercise 1, Exercise 2, Exercise 3)
2. **learn how to write environment operations:** using functions `e:env-get` and `e:env-put`. (Exercise 3)
3. **learn how to implement a function defined by branches:** understand pattern matching and any extra conditions (Exercise 1)
4. **learn how to implement inductive definitions:** reading "fraction" rules; implementing constraints (Exercise 2, Exercise 3)
5. **formal methods:** discussed in the first lecture of this module; investigate and cite your sources (Exercise 4 and 5) — not covered by this lecture

(1) Learn what each symbol means for each language

Learn what each symbol means for each language

- The homework assignment shows **2 different languages!**
- λ_S: Exercise 1 and Exercise 2
- λ_E: Exercise 3

UMass Boston

λ_S (Exercises 1 and 2)

$$e ::= v \mid x \mid (e_1\ e_2) \qquad v ::= n \mid \lambda x.e$$

```
(define-type s:value (U s:number s:lambda))
(define-type s:expression (U s:value s:variable s:apply))

(struct s:number ([value : Number]))
(struct s:variable ([name : Symbol]))
(struct s:lambda ([param : s:variable] [body : s:expression]))
(struct s:apply ([func : s:expression] [arg : s:expression]))
```

λ_E: values (Exercise 3)

$$v ::= n \mid \{E, \lambda x.e\}$$

```
(define-type e:value (U e:number e:closure))
; n
(struct e:number ([value : Number]))
; {E,λx.e}
(struct e:closure (
    [env : e:environ]        ; E
    [param : e:variable]     ; x
    [body : e:expression]    ; e
  )
)
```

λ_E: expressions (Exercise 3)

$$e ::= v \mid x \mid (e_1\ e_2) \mid \lambda x.e$$

```
(define-type e:expression (U e:value e:variable e:apply e:lambda))
; x
(struct e:variable ([ name : Symbol ]))
; (e1 e2)
(struct e:apply (
    [func : e:expression] ; e1
    [arg : e:expression]  ; e2
))
; λx.e
(struct e:lambda (
    [param : e:variable]   ; x
    [body : e:expression]  ; e
))
```

(2) Learn how to write environment operations

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$:

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$: `(hash)`
- How can we encode a lookup $E(x)$:

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$: `(hash)`
- How can we encode a lookup $E(x)$: `(e:env-get E x)`
- How can we encode environment extension $E[x \mapsto v]$:

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$: `(hash)`
- How can we encode a lookup $E(x)$: `(e:env-get E x)`
- How can we encode environment extension $E[x \mapsto v]$: `(e:env-put E x v)`

Test-cases

Test-cases

Function `(check-e:eval? env exp val)` is given in the template to help you test effectively your code.

> The use of `check-e:eval` is **optional**. You are encouraged to play around with `e:eval` directly.

1. The first parameter is an S-expression that represents an ***environment***. The S-expression must be a list of pairs representing each variable binding. The keys must be symbols, the values must be serialized λ_E values

```
[] ; The empty environment
[ (x . 1) ]  ; An environment where x is bound to 1
[ (x . 1) (y . 2) ] ; An environment where x is bound to 1 and y is bound to 2
```

2. The second parameter is an S-expression that represents the a valid λ_E **expression**
3. The third parameter is an S-expression that represents a valid λ_E **value**

UMass Boston

Serialized expressions in λ_E

> Each line represents a **quoted** expression as a parameter of function `e:parse-ast`. For instance, `(e:parse-ast '(x y))` should return `(e:apply (e:variable 'x) (list (e:variable 'y)))`.

```
1                           ; (e:number 1)
x                           ; (e:variable 'x)
(closure [(y . 20)] (lambda (x) x))
; (e:closure
;     (hash (e:variable 'y) (e:number 20))
;     (e:lambda (list (e:variable 'x)) (list (r:variable 'x))))
(lambda (x) x)              ; (e:lambda (list (e:variable 'x)) (list (e:variable 'x)))
(x y)                       ; (e:apply (e:variable 'x) (list (e:variable 'y)))
```

Test cases

```
; x is bound to 1, so x evaluates to 1
(check-e:eval? '[(x . 1)] 'x 1)
; 20 evaluates to 20
(check-e:eval? '[(x . 2)] 20 20)
; a function declaration evaluates to a closure
(check-e:eval? '[] '(lambda (x) x) '(closure [] (lambda (x) x)))
; a function declaration evaluates to a closure; notice the environment change
(check-e:eval? '[(y . 3)] '(lambda (x) x) '(closure [(y . 3)] (lambda (x) x)))
; because we use an S-expression we can use brackets, curly braces, or parenthesis
(check-e:eval? '{(y . 3)} '(lambda (x) x) '(closure [(y . 3)] (lambda (x) x)))
; evaluate function application
(check-e:eval? '{} '((lambda (x) x) 3)  3)
; evaluate function application that returns a closure
(check-e:eval? '{} '((lambda (x) (lambda (y) x)) 3)  '(closure {[x . 3]} (lambda (y) x)))
```

Implementing inductive definitions

A primer

Implementing inductive definitions

A primer

> Disciplining an ambiguous presentation medium to communicate a precise mathematical meaning (**notation** and **convention**)

- Implementing algorithms written in a mathematical notation
- Discuss recursive functions (known as inductive definitions)
- Present various design choices
- We are restricting ourselves to the specification of functions
  (If $M(x) = y$ and $M(x) = z$, then $y = z$)

UMass Boston

Equation notation

- Function $M(n)$ has one input n and one output after the equals sign.
- Each rule declares some pre-conditions
- The result of the function is only returned if the pre-conditions are met

Formally

$$
\begin{aligned}
M(n) &= n - 10 \quad \text{if } n > 100 \\
M(n) &= M(M(n + 11)) \quad \text{if } n \leq 100
\end{aligned}
$$

Implementation

- Each branch of the `cond` represents a rule
- The condition of each branch should be the pre-condition

Equation notation

- Function $M(n)$ has one input n and one output after the equals sign.
- Each rule declares some pre-conditions
- The result of the function is only returned if the pre-conditions are met

Formally

$$M(n) = n - 10 \quad \text{if } n > 100$$
$$M(n) = M(M(n + 11)) \quad \text{if } n \leq 100$$

Implementation

- Each branch of the `cond` represents a rule
- The condition of each branch should be the pre-condition

```
(define (M n)
  (cond
    [(> n 100) (- n 10)]
    [(<= n 100) (M (M (+ n 11)))]))
```

Fraction notation

- We can use the "fraction"-based notation to represent pre-conditions
- Above is a pre-condition, below is the result of the function
- The result is only available if the pre-condition holds

Formally

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{n \leq 100}{M(n) = M(M(n + 11))}$$

Fraction notation

- We can use the "fraction"-based notation to represent pre-conditions
- Above is a pre-condition, below is the result of the function
- The result is only available if the pre-condition holds

Formally

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{n \leq 100}{M(n) = M(M(n + 11))}$$

Implementation

```
(define (M n)
  (cond
    [(> n 100) (- n 10)]
    [(<= n 100) (M (M (+ n 11)))]))
```

Multiple pre-conditions in fraction-notation

- Fraction-based notation admits multiple pre-conditions
- The result only happens if **all** pre-conditions are met (logical conjunction)
- We are only interested in function calls that do always succeed (ignore errors)
- Since we are defining functions, only one output is possible at any time

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{M(n+11) = x \qquad M(x) = y \qquad n \leq 100}{M(n) = y}$$

- In the second rule, note the implicit dependency between variables
- The dependency between variables, specifies the implementation order (eg, `x` must be defined before `y`)

Multiple pre-conditions in fraction-notation

- Fraction-based notation admits multiple pre-conditions
- The result only happens if **all** pre-conditions are met (logical conjunction)
- We are only interested in function calls that do always succeed (ignore errors)
- Since we are defining functions, only one output is possible at any time

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{M(n+11) = x \qquad M(x) = y \qquad n \leq 100}{M(n) = y}$$

- In the second rule, note the implicit dependency between variables
- The dependency between variables, specifies the implementation order (eg, `x` must be defined before `y`)

```
(define (M n)
  (cond
    [(> n 100) (- n 10)]
    [(<= n 100)
     (define x (M (+ n 11)))
     (define y (M x))
     y]))
```

UMass Boston

The equal sign is optional

- The distinction between input and output should be made clear by the author of the formalism

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{M(n+11) = x \qquad M(x) = y \qquad n \leq 100}{M(n) = y}$$

The equal sign is optional

- The distinction between input and output should be made clear by the author of the formalism

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{M(n+11) = x \qquad M(x) = y \qquad n \leq 100}{M(n) = y}$$

We can use any symbol!

Let us define the M function with the 📻 symbol. The intent of notation is to aid the reader and reduce verbosity.

$$\frac{n > 100}{n \text{ 📻 } n - 10} \qquad \frac{n + 11 \text{ 📻 } x \qquad x \text{ 📻 } y \qquad n \leq 100}{n \text{ 📻 } y}$$

> How do we write $M(M(n+11))$?

UMass Boston

Pattern matching rules

- The pre-condition is implicitly defined according to the **structure** of the input
- ***First rule:*** can only be applied if the list is empty
- ***Second rule:*** can only be applied if there is at least one element in the list

$$\mathrm{qs}([]) = []$$

$$\frac{\mathrm{qs}([x \mid x < p \land x \in l]) = l_1 \qquad \mathrm{qs}([x \mid x \geq p \land x \in l]) = l_2}{\mathrm{qs}(p :: l) = l_1 \cdot [p] \cdot l_2}$$

Pattern matching rules (implementation)

```
(define (qs l)
    (cond [(empty? l) empty] ; qs([]) = []
          [else
            ; Input: p :: r
            (define p (first l))
            (define r (rest l))
            ; qs([ x | x < p /\ x \in l]) = l1
            (define l1 (qs (filter (lambda (x) (< x p)) r)))
            ; qs([ x | x >= p /\ x \in l]) = l2
            (define l2 (qs (filter (lambda (x) (>= x p)) r)))
            ; l1 . p . l2
            (append l1 (cons p l2))]))
```

(4) Implementing constraints

(4) Implementing constraints

$$\frac{\overbrace{\texttt{eval}(e_f) = \underbrace{(\lambda(x)\ e_b)}_{\text{pattern match the return value!}}}^{\text{define 1}} \qquad \overbrace{\texttt{eval}(e_a) = v_a}^{\text{define 2}} \qquad \overbrace{\texttt{eval}(\texttt{subst}(e_b, x, v_a)) = v_b}^{\text{define 3}}}{\texttt{eval}((e_f\ e_a)) = \underbrace{v_b}_{\text{return this}}}$$

Homework assignment

- **Exercise 1.** Function $e[x := v]$ is `(s:subst exp var val)`, where e is `exp`, x is `var`, and v is `val`.
- **Exercise 2.** Function $e \Downarrow v$ is `(s:eval subst exp)`, where e is `exp`, v is the return value (not displayed in the function signature).

  > In the exercise, parameter `subst` represents the substitution function (local tests use your own implementation, remote tests use a correct implementation of subst).

- **Exercise 3.** Function $e \Downarrow_E v$ is `(e:eval env exp)`, where e is `exp`, E is `env`, v is the return value (not displayed in the function signature).

Church's encoding

Church's encoding

- Alonzo Church created the λ-calculus
- Church's Encoding is a treasure trove of λ-calculus expressions: it shows how natural numbers can be encoded
- Let us go through Church's encoding of booleans
- Examples taken from Colin Kemp's PhD thesis (page 17)

Encoding Booleans with λ-terms

Why? Because you will be needing test-cases.

```
(require rackunit)
(define ns (make-base-namespace))
(define (run-bool b) (((eval b ns) #t) #f))

; True
(define TRUE '(lambda (a) (lambda (b) a)))
(define FALSE '(lambda (a) (lambda (b) b)))
(define (OR a b) (list (list a TRUE) b))
(define (AND a b) (list (list a b) FALSE))
(define (NOT a) (list (list a FALSE) TRUE))
(define (EQ a b) (list (list a b) (NOT b)))
; Test
(check-equal?
    (run-bool (EQ TRUE (OR (AND FALSE TRUE) TRUE)))
    (equal? #t (or (and #f #t) #t)))
```

Hash-tables

TL;DR: A data-structure that stores pairs of key-value entries. There is a lookup operation that given a key retrieves the value associated with that key. Keys are unique in a hash-table, so inserting an entry with the same key, replaces the old value by the new value.

- Hash-tables represent a (partial) injective function.
- Hash-tables were covered in CS310.
- Hash-tables are also known as maps, and dictionaries. We use the term hash-table, because that is how they are known in Racket.

Hash-tables in Racket

Constructors

1. Function `(hash k1 v1 ... kn vn)` a hash-table with the given key-value entries. Passing zero arguments, `(hash)`, creates an empty hash-table.
2. Function `(hash-set h k v)` copies hash-table `h` and adds/replaces the entry `k` `v` in the new hash-table.

Accessors

- Function `(hash? h)` returns `#t` if `h` is a hash-table, otherwise it returns `#f`
- Function `(hash-count h)` returns the number of entries stored in hash-table `h`
- Function `(hash-has-key? h k)` returns `#t` if the key is in the hash-table, otherwise it returns `#f`
- Function `(hash-ref h k)` returns the value associated with key `k`, otherwise aborts

Hash-table example

```
(define h (hash))                      ; creates an empty hash-table
(check-equal? 0 (hash-count h))        ; we can use hash-count to count how many entries
(check-true (hash? h))                 ; unsurprisingly the predicate hash? is available

(define h1 (hash-set h "foo" 20))      ; creates a new hash-table where "foo" is bound to 20
(check-equal? (hash "foo" 20) h1)      ; (hash-set (hash) "foo" 20) = (hash "foo" 20)

(define h2 (hash-set h1 "foo" 30))
(check-equal? (hash "foo" 30) h2)      ; in h2 "foo" is the key, and 30 the value
(check-equal? 30 (hash-ref h2 "foo")) ; ensures that hash-ref retrieves the value of "foo"
(check-equal? (hash "foo" 20) h1)      ; h1 remains the same
```