Structure of Higher Level Languages

Lecture 6: Nested definitions; caching

Tiago Cogumbreiro
Today we will learn...

- Manipulating the ASTs
- Functions as data-structures: exercises
- Storing functions in data-structures
- Currying
- Exists
- Map
The abstract syntactic tree (AST)

Representing code as data structures
### The AST of values

\[
\text{value} = \text{number} \mid \text{void} \mid \text{func-dec}
\]

\[
\text{func-dec} = (\text{lambda} \ (\text{variable}^{*} \ )) \ \text{term}^{+}
\]

#### Implementation

\[
\begin{align*}
(\text{define} \ (r:\text{value}? \ v)) \\
(\text{or} \ (r:\text{number}? \ v)) \\
(r:\text{void}? \ v) \\
(r:\text{lambda}? \ v))
\end{align*}
\]

\[
(\text{struct} \ r:\text{void} () \ #:\text{transparent}) \\
(\text{struct} \ r:\text{number} \ (\text{value}) \ #:\text{transparent}) \\
(\text{struct} \ r:\text{lambda} \ (\text{params} \ \text{body}) \ #:\text{transparent})
\]

#### How do we represent?

1. 10
2. (void)
3. (lambda () 10)

#### AST
The AST of values

\[
\text{value} = \text{number} \mid \text{void} \mid \text{func-dec}
\]
\[
\text{func-dec} = \text{lambda} \ (\ \text{variable}^* \ ) \ \text{term}^+ \)
\]

Implementation

\[
\begin{align*}
& (\text{define} \ (r:\text{value}? \ v)) \\
& \quad (\text{or} \ (r:\text{number}? \ v)) \\
& \quad (r:\text{void}? \ v) \\
& \quad (r:\text{lambda}? \ v))) \\
& (\text{struct} \ r:\text{void} () #:\text{transparent}) \\
& (\text{struct} \ r:\text{number} \ (\text{value}) #:\text{transparent}) \\
& (\text{struct} \ r:\text{lambda} \ (\text{params} \ \text{body}) #:\text{transparent})
\end{align*}
\]

How do we represent?

1. 10
2. (void)
3. (lambda () 10)

AST

\[
\begin{align*}
& (r:\text{number} \ 10) \ ; \gets 1 \\
& (r:\text{void}) \ ; \gets 2 \\
& (r:\text{lambda} \ (\text{list}) \ ; \gets 3 \\
& \quad (\text{list} \ (r:\text{number} \ 10)))
\end{align*}
\]
The AST of expressions

expression = value | variable | apply
apply = ( expression+ )

Implementation

(define (r:expression? e)
  (or (r:value? e)
      (r:variable? e)
      (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)

How do we represent?

1. \( x \)
2. \( f \ 10 \)

AST
The AST of expressions

expression = value | variable | apply
apply = ( expression+ )

Implementation

(define (r:expression? e)
  (or (r:value? e)
    (r:variable? e)
    (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)

How do we represent?

1. x
2. (f 10)

AST

; 1:
  (r:variable 'x)
; 2:
  (r:apply
    (r:variable 'f)
    (list (r:number 10))))
The AST of terms

term = define | expression
define = ( define identifier expression ) | ( define ( variable+ ) term+)

(define (r:term? t)
  (or (r:define? t)
      (r:expression? t)))
(struct r:define (var body) #:transparent)

Which Racket code is this?

(r:define (r:variable 'f)
  (r:lambda (list (r:variable 'y))
    (list
      (r:apply (r:variable '+)
        (list (r:variable 'y) (r:number 10))))))
The AST of terms

\[
\text{term} = \text{define} \mid \text{expression} \\
\text{define} = (\text{define identifier expression}) \mid (\text{define (variable+ term+})
\]

Which Racket code is this?

Answer 1

\[
\begin{align*}
(r:define (r:variable 'f) \\
(r:lambda (list (r:variable 'y)) \\
(list (r:apply (r:variable '+) \\
(list (r:variable 'y) (r:number 10))))))
\end{align*}
\]

Answer 2

\[
\begin{align*}
\text{(define (f y) (+ y 10))} \\
\text{(define f (lambda (y) (+ y 10))})
\end{align*}
\]
Functions as data-structures

Exercises
Exercise 1

What is the output of this program?

```
(define x 10)
(define (f x)
  (+ x 20))
(f 30)
```
Exercise 1

What is the output of this program?

\[(\text{define } x 10)\]
\[(\text{define } (f \ x) (\text{+ } x 20))\]
\[(f 30)\]

**Output:** 50

Because, parameter \( x \) shadows the outermost definition.
Exercise 2

What is the output of this program?

```
(define x 10)
(define f (lambda (x) (+ x 20)))
(f 30)
```
Exercise 2

What is the output of this program?

```
(define x 10)
(define f (lambda (x) (+ x 20)))
(f 30)
```

**Output:** 50

The code above is **equivalent** to the code below:
```
(define (f x) (+ x 20))
```
Exercise 3

What is the output of this program?

(define (factory k)
  (lambda () k))

(factory 10)
Exercise 3

What is the output of this program?

```
(define (factory k)
  (lambda () k))

(factory 10)
```

**Output:** `<procedure>`

Although if Racket displayed code, we would get: `(lambda () 10)`

```
((factory 10))
; Outputs: 10
```
Exercise 3

Step-by-step evaluation

```
(factory 10) =
((lambda (k) (lambda () k)) 10) =
(lambda () 10)
```

Why is factory replaced by a lambda?

User input

```
(define (factory k)
  (lambda () k))
```

Internal representation

```
(define factory
  (lambda (k)
    (lambda () k)))
```
Exercise 3

Looking at function application more closely

```
(lambda (k)
  (lambda () k)) ; ← body of function
  10 ; ← argument
)

; Remove outer lambda and replace each parameter by argument
; (lambda () k) ← body of function
; \___ replace parameter k by argument 10
(lambda () 10) ; ← return value
```
Exercise 4

Q1: What is the output of this program?

```scheme
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))
g
```
Exercise 4

Q1: What is the output of this program?

```
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))
g
```

**Output:** `(lambda (b) (cond [b 1] [else 2]))`

Q2: How do I call `g` to obtain 1?
Exercise 4

Q1: What is the output of this program?

```lisp
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))
g
```

Output: `(lambda (b) (cond [b 1] [else 2]))`

Q2: How do I call `g` to obtain 1?

Solution: `(g #t)`
Implementing a pair with functions alone

If we can capture one parameter, then we can also capture two parameters. Let us implement a pair-data structure with only functions!

```scheme
(define (cons x y)
  (lambda (b) ; ← we use a parameter to choose which stored data to return
    (cond [b x] [else y])))) ; ← passing #t returns x
    ; passing #f returns y

; We now define our own 'car' and 'cdr'
(define (car f) (f #t)); Returns the first element of the pair
(define (cdr f) (f #f)); Returns the second element of the pair

(define p (cons 10 20)) ; Same as: (define (p b) (cond [b 10] [else 20]))
(car p); Returns 10 because (car p) → (p #f) → 10
(cdr p); Returns 20 because (cdr p) → (p #t) → 20
```
Functions in data structures
Functions stored in data structures

"Freeze" one parameter of a function

In this example, a frozen data-structure stores a binary-function and the first argument. Function apply1 takes a frozen data structure and the second argument, and applies the stored function to the two arguments.

```scheme
(struct frozen (func arg1) #:transparent)

(define (apply1 fr arg)
    (define func (frozen-func fr)) ; Bind a function to a local variable
    (define arg1 (frozen-arg1 fr))
    (func arg1 arg)) ; Call a function bound to a local variable

(define frozen-double (frozen * 2)); Store function '*' in a data structure
(define (double x) (apply1 frozen-double x))
(check-equal? (* 2 3) (double 3))
```
Unfolding \((\text{double } 3)\)

\[
(\text{double } 3) \\
= (\text{apply1 frozen-double } 3) \\
= (\text{apply1 (frozen } * \ 2 \ 3) \\
= (\text{define fr (frozen } * \ 2)) \\
((\text{frozen-func fr) (frozen-arg1 fr) 3}) \\
= (* \ 2 \ 3) \\
= 6
\]
Functions stored in data structures

Apply a list of functions to a value

```scheme
#lang racket
(define (double n) (* 2 n))  ; A list with two functions:
 ; * doubles a number
 ; * increments a number
(define p (list double (lambda (x) (+ x 1))))  ; Applies each function to a value
(define (pipeline funcs value)
  (cond [(empty? funcs) value]
        [else (pipeline (rest funcs) ((first funcs) value))]))
; Run the pipeline
(check-equal? (+ 1 (double 3)) (pipeline p 3))
```
Creating functions dynamically
Functions in Racket automatically capture the value of any variable referred in its body.

Example

```racket
#lang racket
(define (frozen-* arg1)
  (define (get-arg2 arg2)
    (* arg1 arg2))
; Returns a new function
; every time you call frozen-
  get-arg2)
(require rackunit)
(define double (frozen-* 2))
(check-equal? (* 2 3) (double 3))
```

Evaluating `(frozen-* 2)`

```racket
(frozen-* 2)
= (define (get-arg2 arg2) (* 2 arg2)) get-arg2
= (lambda (arg2) (* 2 arg))
```

Evaluating `(double 3)`

```racket
(double 3)
= ((frozen-* 2) 3)
= ((lambda (arg2) (* 2 arg2)) 3)
= (* 2 3)
= 6
```
Currying functions
Freezing binary-function

(define (apply1 fr arg)
  (define func (frozen-func fr))
  (define arg1 (frozen-arg1 fr))
  (func arg1 arg))

(define frozen-double (frozen * 2))
(define (double x) (apply1 frozen-double x)
(check-equal? (* 2 3) (double 3))

Our freeze function is more general than freeze-\* and simpler than frozen-double. We abstain from using a data-structure and use Racket's variable capture capabilities.
Generalizing "frozen" binary functions

Attempt #2

```scheme
(define (freeze f)
  (define (expect-1 arg1)
    (define (expect-2 arg2)
      (f arg1 arg2))
    expect-2)
  expect-1)

(define frozen-* (freeze *))
(define double (frozen-* 2))
(check-equal? (* 2 3) (double 3))
```

Evaluation

```scheme
(define frozen-* (freeze *))
= (define frozen-
  (define (expect-1 arg1)
    (define (expect-2 arg2)
      (* arg1 arg2))
    expect-2)
  expect-1)

(define double (frozen-* 2))
= (define double
  (define (expect-2 arg2) (* 2 arg2))
  expect-2)

(double 3)
= (* 2 3)
```
Currying functions

Currying is the general technique of "freezing" functions with multiple parameters. It provides a way of delaying (and caching) the passage of multiple arguments by means of new functions.

A curried function \( \text{curry}_{f,n,a}(x) \) is a unary function annotated with an uncurried function \( f \) arguments \( a \) and a number of expected arguments \( n \) that can be recursively defined as:

\[
\begin{align*}
\text{curry}_{f,n+1,[a_1,\ldots,a_n]}(x) &= \text{curry}_{f,n,[a_1,\ldots,a_n,x]} \\
\text{curry}_{f,0,[a_1,\ldots,a_n]}(x) &= f(a_1,\ldots,a_n,x)
\end{align*}
\]

```racket
#lang racket
(define frozen- (curry *))
(define double (frozen- 2))
(require rackunit)
(check-equal? (* 2 3) (double 3))
```
In some programming languages functions are curried by default. Examples include Haskell and ML.

The term currying is named after Haskell Curry, a notable logician who developed combinatory logic and the Curry-Howard correspondence (practical applications include proof assistants).

Haskell was born in Millis, MA (1 hour drive from UMB).

Source: public domain
Uncurried functions

- All arguments must be provided at call-time, otherwise error.

Python example

```python
def add(l, r):
    return l + r

add(10)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: add() missing 1 required positional argument: 'r'
```
Curried functions

If we provide one argument to a 2-parameters function, the result is a 1-parameter function that expects the second argument.

Haskell example

```
-- Define addition
add x y = x + y
-- Define adding 10 to some number
add10 = add 10
-- 10 + 30
add10 30
-- 40
```
Currying in Racket

Function curry **converts** an uncurried function into a curried function.

```racket
#lang racket
(define curried-add (curry +))
(define add10 (curried-add 10))
(require rackunit)
(check-equal? (+ 10 30) (add10 30))
```

**HW2**

- In HW2 you will need to implement the reverse, function *uncurry*.
- You are now ready to solve exercises 1, 4, and 5.
Currying functions

**Currying** is the general technique of "freezing" functions with multiple parameters. It provides a way of delaying (and caching) the passage of multiple arguments by means of new functions.

A curried function $\text{curry}_{f, n, a}(x)$ is a unary function annotated with an uncurried function $f$ arguments $a$ and a number of expected arguments $n$ that can be recursively defined as:

\[
\begin{align*}
\text{curry}_{f, n+1, [a_1, \ldots, a_n]}(x) &= \text{curry}_{f, n, [a_1, \ldots, a_n, x]} \\
\text{curry}_{f, 0, [a_1, \ldots, a_n]}(x) &= f(a_1, \ldots, a_n, x)
\end{align*}
\]
Exercise 6

What is the output of this program?

Program

```scheme
(define curried-add
  (lambda (arg1)
    (lambda (arg2)
      (+ arg1 arg2))))

(define a (curried-add 10))
(define b (curried-add 20))

a
b
(a 30)
(b 40)
```
Exercise 6

What is the output of this program?

Program

```
(define curried-add
  (lambda (arg1)
    (lambda (arg2)
      (+ arg1 arg2))))

(define a (curried-add 10))
(define b (curried-add 20))

(a 30)
(b 40)
```

Output

```
(lambda (arg2) (+ 10 arg2))
(lambda (arg2) (+ 20 arg2))
40
60
```
Functional patterns: Does it exist?
Element in the list?

Let us implement a function `member` that tests whether or not a list contains a value.

Specification

```scheme
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```
Let us implement a function `member` that tests whether or not a list contains a value.

**Specification**

```scheme
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

**Solution**

```scheme
(define (member x l)
 (cond
   [(empty? l) #f]
   [(equal? (first l) x) #t]
   [else (member x (rest l))]))
```

Is the solution tail-recursive?
Element in the list?

Let us implement a function `member` that tests whether or not a list contains a value.

**Specification**

```scheme
;; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

**Solution**

```scheme
(define (member x l)
  (cond
   [(empty? l) #f]
   [(equal? (first l) x) #t]
   [else (member x (rest l))]))
```

Is the solution tail-recursive? **Yes!**
Element in the list?

Overview of our solution

Recursive code mirrors the structure your data!

Think of how many constructors your data has, those will be your recursive cases.

- **Case empty**: the empty list constructor, same as \((\text{list})\)
- **Case cons**: add one element to the list with the \((\text{cons} \ x \ 1)\) constructor
- Recursive call must handle "smaller" data
  - with lists: \((\text{rest} \ 1)\)
  - with numbers: \((+ \ n \ 1)\) if you approach an upper bound
  - with numbers: \((- \ n \ 1)\) if you approach a lower bound
A general recursion pattern for handling lists

1. Case empty (handle-base)
2. Case cons (handle-step)
3. Recursive call handles "smaller"

(define (rec v)
  (cond
   [(base-case? v) (handle-base v)]
   [else (handle-step v (rec (decrement v)))]))
A general recursion pattern for handling lists

1. Case empty (handle-base)
2. Case cons (handle-step)
3. Recursive call handles "smaller"

Example for member

```scheme
(define (member x l)
  (cond
    [(empty? l) #f]; handle-base: #f
    [else
     (cond
      [(equal? (first l) x) #t]; handle-step
      [else (member x (rest l))])])
)
```

In this version, we make the base and handle-steps explicit. Previous solution coalesces nested conds into one.
Common mistake 1

Forgetting the base case

• **Symptom:** first contract violation

Example

```scheme
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

Base case missing

```scheme
(define (member x l)
  (cond
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))

; first: contract violation
; expected: (and/c list? (not/c empty?))
; given: '()
; [,bt for context]
```
Common mistake 2

Forgetting to make the list smaller

- **Symptom:** program hangs (runs forever) for some inputs

Correct

```lisp
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))])))
```

Incorrect

```lisp
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x l)]))
```

Common mistake 2

Forgetting to make the list smaller

- **Symptom:** program hangs (runs forever) for some inputs
Generalizing member
Exists prefix in list?

Spec

```
(require rackunit)
(check-true (string-prefix? "Racket" "R")) ; available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))
```
exists prefix in list?

spec

(require rackunit)
(check-true (string-prefix? "Racket" "R")); available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))

solution

(define (match-prefix? prefix l)
  (cond
   [(empty? l) #f]
   [(string-prefix? (first l) prefix) #t]
   [else (match-prefix? prefix (rest l))])))
Can we generalize the search algorithm?

; Example 1
(define (member x l)
  (cond
   [(empty? l) #f]
   [(equal? (first l) x) #t]
   [else (member x (rest l))])))

; Example 2
(define (match-prefix? x l)
  (cond
   [(empty? l) #f]
   [(string-prefix? (first l) x) #t]
   [else (match-prefix? x (rest l))])))
Can we generalize the search algorithm?

Example 1

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

Example 2

```
(define (match-prefix? x l)
  (cond
    [(empty? l) #f]
    [(string-prefix? (first l) x) #t]
    [else (match-prefix? x (rest l))]))
```

Example 1

```
(define (exists predicate l)
  (cond
    [(empty? l) #f]
    [(predicate (first l)) #t]
    [else (exists predicate (rest l))]))
```

Example 2

```
(define (match-prefix? x l)
  (exists
    (lambda (y) (string-prefix? y x)) l))
```

Solution