CS450

Structure of Higher Level Languages

Lecture 5: Modules, tail-call optimization, structs, functions as values

Tiago Cogumbreiro

Today we learn...

- Modules
- Tail-call optimization
- Structs
- Defining an Abstract Syntax Tree
- Functions as values
  - functions in data-structures
  - dynamically creating functions

Modules

Modules

- Modules encapsulate a unit of functionality
- A module groups a set of constants and functions
- A module encapsulates (hides) auxiliary top-level functions
- Each file represents a module

Modules in Racket

> Each file represents a module. A bindings becomes visible through the `provide` construct. Function `(require "filename")` loads a module

- `(provide (all-defined-out))` makes all bindings visible
- `(provide a c)` makes binding `a` and `c` visible
- `(require "foo.rkt")` makes all bindings of the module in file `foo.rkt` visible in the current module. Both files have to be in the same directory.

File: `foo.rkt`

```
#lang racket
; Make variables a and c visible
(provide a c)
(define a 10)
(define b (+ a 30))
(define (c x) b)
```

File: `main.rkt`

```
(require "foo.rkt")
(c a)
; b is not visible
```

Tail-call optimization

Why does it work?

Call stack & Activation frame

- **Call Stack:** To be able to call and return from functions, a program internally maintains a stack called the ***call-stack***, each of which holds the execution state at the point of call.
- **Activation Frame:** An activation frame maintains the execution state of a running function. That is, the activation frame represents the local state of a function, it holds the state of each variable.
- **Push:** When calling a function, the caller creates an activation frame that is used by the called function (eg, to pass arguments to the function being called).
- **Pop:** Before a function returns, it pops the call stack, freeing its local state.

Consider executing the factorial

Program

```
(define (fact n)
  (cond
    [(= n 1) 1]
    [else
     (* n (fact (- n 1)))]))
```

Evaluation

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Call-Stack

```
[n=3,return=(* 3 (fact 2))]
[n=3,return=(* 3 ?)],[n=2,return=(* 2 (fact 1))]
[n=3,return=(* 3 ?)],[n=2,return=(* 2 ?)],[n=1,return=1]
[n=3,return=(* 3 ?)],[n=2,return=2]
[n=3,return=6]
```

Call-stack and recursive functions

> Recursive functions pose a problem to this execution model, as **the call-stack may grow unbounded**! Thus, most non-functional programming languages are conservative on growing the call stack.

```
def fact(n):
  return 1 if n ≤ 1 else n * fact(n - 1)
fact(1000)
```

Outputs

```
  File "<stdin>", line 1, in fact
RuntimeError: maximum recursion depth exceeded
```

Factorial: attempt #2

Program

```
(define (fact n)
  (define (fact-iter n acc)
    (cond
      [(= n 0) acc]
      [else
        (fact-iter (- n 1) (* acc n)) ]))
  (fact-iter n 1))
(fact 3)
```

Evaluation

```
(fact 3)
(fact-iter 3 1)
(fact-iter 2 3)
(fact-iter 1 6)
6
```

Factorial: attempt #2

Call stack

```
[n=3,return=(fact-iter 3 1)]
[n=3,return=?],[n=3,acc=1,return=(fact-iter 2 3)]
[n=3,return=?],[n=3,acc=1,return=?],[n=2,acc=3,return=(fact-iter 1 6)]
[n=3,return=?],[n=3,acc=1,return=?],[n=2,acc=3,return=?],[n=1,acc=6,return=6]
[n=3,return=?],[n=3,acc=1,return=?],[n=2,acc=3,return=6]
[n=3,return=?],[n=3,acc=1,return=6]
[n=3,return=6]
```

Tail position and tail call

> The ***tail position*** of a sequence of expressions is the last expression of that sequence.

```
(lambda ()
  exp1
  ; ...
  expn) ⟵ tail position
```

> When a function call is in the tail position we named it the ***tail call***.

```
(lambda ()
  exp1
  ; ...
  (f ...)) ⟵ f is a tail call
```

Tail call and the call stack

A tail call does not need to push a new activation frame! Instead, the called function can "reuse" the frame of the current function. For instance, in `(fact 3)`, the call `(fact-iter 3 1)` is a tail call.

```
[n=3,return=(fact-iter 3 1)]
[n=3,return=?],[n=3,acc=1,return=(fact-iter 2 3)]
```

Can be rewritten with:

```
[n=3,return=(fact-iter 3 1)]
[n=3,acc=1,return=(fact-iter 2 3)]
```

In attempt #2, both calls to `fact-iter` are tail calls.

Tail-Call Optimization

- Eschews the need to allocate a new activation frame
- In a recursive tail call, the compiler can convert the recursive call into a loop, which is more efficient to run (recall our $5\times$ speedup)

Revisiting user data structures

User data structures

Recall the 3D point from Lecture 3

```
; Constructor
(define (point x y z) (list x y z))
; Accessors
(define (point-x pt) (first pt))
(define (point-y pt) (second pt))
(define (point-z pt) (third pt))
```

And the `name` data structure

```
; Constructor
(define (name f m l) (list f m l))
; Accessor
(define (name-first n) (first n))
(define (name-middle n) (second n))
(define (name-last n) (third n))
```

How do we prevent such errors?

```
(define p (point 1 2 3))
(name-first p) ; This should be an error, and instead it happily prints 1
```

Introducing `struct`

```
#lang racket
(require rackunit)
(struct point (x y z) #:transparent)
(define pt (point 1 2 3))
(check-equal? 1 (point-x pt))  ; the accessor point-x is automatically defined
(check-equal? 2 (point-y pt))  ; the accessor point-y is automatically defined
;
(struct name (first middle last))
(define n (name "John" "M" "Smith"))
(check-equal? "John" (name-first n))
(check-true (name? n))   ; We have predicates that test the type of the value
(check-false (point? n)) ; A name is not a point
(check-false (list? n))  ; A name is not a list
; (point-x n) ;; Throws an exception
; point-x: contract violation
;   expected: point?
;   given: #<name>)
```

Benefits of using structs

- Reduce boilerplate code
- Ensure type-safety

Implementing Racket's AST

Implementing Racket's AST

Grammar

```
expression = value | variable | apply | define
value = number | void | lambda
apply = ( expression+ )
lambda = ( lambda ( variable* ) term+)
```

Implementing values

```
value = number | void | lambda
lambda = ( lambda ( variable* ) term+)
```

Implementing values

value = *number* | *void* | *lambda*
lambda = `(` `lambda` `(` *variable** `)` *term+*`)`

```
(define (r:value? v)
  (or (r:number? v)
      (r:void? v)
      (r:lambda? v)))
(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)
```

> We are using a prefix `r:` because we do not want to redefined standard-library definitions.

Implementing expressions

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

Implementing expressions

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

```
(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)
```

> In `r:apply` we distinguish between the expression that represents the function `func`, and the (possibly empty) list of arguments `args`.

Implementing terms

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

Implementing 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)
```

> For our purposes of defining the semantics in terms of implementing an interpreter, we do not want to distinguish between a basic definition and a function definition, as this would unnecessarily complicate our code. We, therefore, represent a definition with a single structure, which pairs a variable and an expression (eg, a lambda). In our setting, the distinction between a basic and a function definition is syntactic (not semantic).

Summary of `struct`

```
(struct point (x y z) #:transparent)
```

Simplifies the definition of data structures:

- Creates selectors automatically, eg, `point-x`
- Creates type query, eg, `point?`
- Ensures that functions of a given struct can only be used on values of that struct.
  Because, not everything is a list.

> What is `#:transparent`? A transparent struct prints its contents when rendered as a string.

Functions as values

What is functional programming

Functional programming has different meanings to different people

- Avoid mutation
- **Using functions as values**
- A programming style that encourages recursion and recursive data structures
- A programming model that uses ***lazy*** evaluation (discussed later)

First-class functions

- **Functions are values:** can be passed as arguments, stored in data structures, bound to variables, …
- **Functions for extension points:** A powerful way to factor out a common functionality

Functions as parameters

Functions as parameters

Monotonic increasing function (for one input)

> Function `monotonic?` takes a function `f` as a parameter and a value `x`, and then checks if `f` increases monotonically for a given `x`.

Example

```
#lang racket
(define (double n) (* 2 n))
(define (monotonic? f x)
  (≥ (f x) x))
;; Tests
(require rackunit)
(check-true (monotonic? double 3))
(check-false (monotonic? (lambda (x) (- x 1)) 3))
```

How do we evaluate?

```
(monotonic? double 3)
```

Functions as parameters

Monotonic increasing function (for one input)

> Function `monotonic?` takes a function `f` as a parameter and a value `x`, and then checks if `f` increases monotonically for a given `x`.

Example

```
#lang racket
(define (double n) (* 2 n))
(define (monotonic? f x)
  (≥ (f x) x))
;; Tests
(require rackunit)
(check-true (monotonic? double 3))
(check-false (monotonic? (lambda (x) (- x 1)) 3))
```

How do we evaluate?

```
  (monotonic? double 3)

= (≥ (double 3) 3)
= (≥ ((lambda (n) (* 2 n)) 3) 3)
= (≥ (* 2 3) 3)
= (≥ 6 3)
= #t
```

Functions as parameters

Recursively apply a function n-times

> Function `apply-n` takes a function `f` as parameter, a number of times `n`, and some argument `x`, and then recursively calls `(f (f (... (f x))))` an `n`-number of times.

```
#lang racket
(define (apply-n f n x)
  (cond [(≤ n 0) x]
        [else (apply-n f (- n 1) (f x))]))
;; Tests
(require rackunit)
(define double (lambda (x) (* 2 x)))
(check-equal? (* 2 (* 2 (* 2 1))) (apply-n double 3 1))
(check-equal? (+ 3 (+ 3 (+ 3 1))) (apply-n (lambda (x) (+ 3 x)) 3 1))
```

Example apply-n

Let us unfold the following...

```
(apply-n double 3 1)                    ; (≤ 3 0) = #f
```

Example `apply-n`

Let us unfold the following...

```
  (apply-n double 3 1)                  ; (≤ 3 0) = #f

= (apply-n double (- 3 1) (double 1))
= (apply-n double 2 2)                  ; (≤ 2 0) = #f
= (apply-n double (- 2 1) (double 2))
= (apply-n double 1 4)                  ; (≤ 1 0) = #f
= (apply-n double (- 1 1) (double 4))
= (apply-n double 0 8)                  ; (≤ 0 0) = #t
= 8
```

Functions as data-structures

Functions as data-structures

The following is a function that returns a constant value (returns 3 always):

```
(define three:2 (lambda () 3))

(three:2) ; ← We need to call the function to obtain its contents
          ;    Note that we are passing 0 parameters.
```

Note the difference...

The following is a variable binding (not a function!):

```
(define three:1 3)
```

Variable `three:1` ***evaluates*** to the number `3`.

A factory of constant-return functions

We can generalize the procedure by creating a function that returns a new function declaration that returns a given parameter n.

```
(define (factory n)
  (lambda () n)) ; ⟵ calling 'factory' creates a new function dynamically
                 ;     each new function captures the given 'n',
                 ;     which changes according to how it was called

(define three:4 (factory 3)) ; ← same as: (define three:4 (lambda () 3))
(three:4) ; Returns: 3

(define four:1 (factory 4)) ; ← same as: (define four:1 (lambda () 4))
(four:1) ; Returns 4
```