Computer Science 136:
Elementary Algorithm Design and Data Abstraction
Winter 2013

University of Waterloo
David R. Cheriton
School of Computer Science

Unit 1 – Modules + I/O + Mutation

Introduction to Modules

- In CS 135, we put all of our functions in the same file
- In the "real world" this is often a poor strategy:
  - Becomes unwieldy for large programs
  - Doesn't facilitate code re-use between projects
  - Makes teamwork difficult
- Most programs are a collection of files

- There are many ways to divide a big program into smaller files
- A good strategy is to use **modules**, where each module is a group of functions that share a common aspect or purpose
- We explore *modular design* throughout this course

Encapsulation

- In CS 135, we introduced the term **encapsulation**
  - Structures were an example of *data* encapsulation
  - `local` functions (functions inside of other functions) were an example of *behaviour* encapsulation: `local` functions were “hidden” from (or “invisible”) to the outside
- Modules facilitate more advanced behaviour encapsulation, and we explore this later under the topic of **Separation of Concerns**

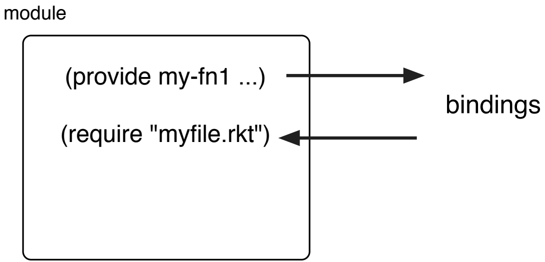
Modules: The "client"

- One our goals is to make our module re-usable by others
- It is helpful to think of "**clients**" who will be using our module
- Clients only need to know how to *use* our module: they do not need to know how we *implemented* our module

A good (non-computing) example of a well designed module is a AA battery. We can use it in a wide variety of applications, and yet most people do not know how the battery was constructed or how it works.

Modules

Modules allow you to control what definitions are available to your clients and what definitions are only available to your module



- **Bindings** describe relationships to other code
- `(provide my-fn1 ...)`: makes `(my-fn1 ...)` available to clients
- `(require "myfile.rkt")`: used by the client to gain access to the definitions (`provide`d) by the `myfile.rkt` module

Modules in Racket

- There is a `module` special form in Racket, but we won't normally use it
- When the very first line of your file is: `#lang racket`
  Racket automatically "wraps" your entire file into a module with the same name as your file name
- For this course, each Racket module will be a single `.rkt` file and you should always keep all of the relevant files (modules) together in the same directory (folder)
- For more information on modules:
  http://docs.racket-lang.org/guide/module-basics.html

`provide` Special Form

- Typically, modules `provide` functions, but you may provide constants as well (anything that is `define`d)
- The `provide` special form is like the "opposite" of the `local` special form: `local` makes definitions "invisible" outside of it, whereas `provide` makes definitions "visible"
- In other words, all definitions inside a module that are **not** explicitly listed by `(provide ...)` are automatically "local" to that module

Module Interface

- Modules should be documented with a module **interface** that gives the client all the information necessary to use your module
- The interface is written **for clients**: outsiders who want to use your module, but do not need to know how it is implemented
- The interface includes a description of the module, a list of definitions it provides (which is obvious in Racket), and for each function:
  - The contract for the function
  - The function header (with parameter names) and purpose
- The interface should also provide examples to illustrate how the module is used (often showing how several functions interact)

Module Interfaces in Racket

In Racket, the interface should appear at the top of your module file, before any definitions

```
#lang racket
;; A simple interface for a simple module
(provide sum-first)

;; sum-first: Int -> Int
;; (sum-first k) Produces the sum of the first k
;;   positive integers (1..k)

;; Examples: (sum-first 1) => 1
;;           (sum-first 10) => 55
```

In the body of your file, you should document all internal functions (not `provide`d) and you might need to augment the interface documentation with more implementation-specific details.

`require` Special Form

- (require "my-module.rkt") accesses the file `"my-module.rkt"` and evaluates everything in it, as though you were “running” the file
- This may produce *side-effects*, like printing results (more on side effects later)
- Definitions listed in `(provide ...)` of `my-module.rkt` will be accessible from the current (or *client*) program/module
- Provided definitions appear as if they were defined at the “top-level” in the client: Racket produces an error if the module and client have definitions with the same name
- If you `require` the same module more than once, additional `require`s are ignored

Module Testing

- You should avoid having any top-level evaluations in your module, otherwise those values will be displayed every time a client `require`s your module
- For each new module you create, you should also create a corresponding **testing module** to test the behaviour of your interface functions
- All of your CS 135 testing strategies (boundary points, testing all paths in a `cond`, etc.) should be used
- In the "real world", you may wish to have an *integration* test module that tests if *two or more* modules are working together properly

Testing sum.rkt

```
#lang racket ;; sum.rkt
(provide sum-first)

(define (sum-first n)
  (cond [(zero? n) 0]
        [else (+ n (sum-first (sub1 n)))]))
```

```
#lang racket ;; test-sum.rkt
(require "sum.rkt")
;; Test Module:  Should produce all #t

(equal? (sum-first 1) 1)
(equal? (sum-first 2) 3)
(equal? (sum-first 10) 55)
```

Remember that `check-expect` is no longer supported. You may want to create you own functions to help facilitate testing.

Changing The Module Implementation

- A key advantage of modules is that you can change the *implementation* of a module without changing the *interface* (how the module is used by the client)
- For example, you might be able to improve the *efficiency* of the module
- Consider the following implementation of `sum-first`:

```
(define (sum-first n)
  (/ (* n (add1 n)) 2))
```

- We discuss efficiency in more detail later, but clearly this implementation is much faster than the recursive implementation

Modules: Summary

With modules, you can:

- Break a large program into more manageable pieces
- Improve code re-use and make code available to others (clients)
- Control the interface: what is accessible to clients and what is not
- Change the implementation without changing the interface

The `begin` special form

- Before we learn about I/O in Racket, we introduce some new Racket language features
- The `begin` special form evaluates a *sequence* of expressions and produces the value of the final expression

```
(define mystery (begin
                   "four"
                   'four
                   (+ 2  2))
```

  The value of mystery is 4 (the value of the last expression)
- `begin` evaluates each expression, but it *ignores* all of the values except the final one

- This may not seem very useful, but it will be handy when we discuss I/O

Implicit `begin`

- You will rarely use `begin` because Racket often *implicitly* interprets a sequence of expressions as if you had used `begin`

```
(define (my-sqr x)
  (+ x x)
  (* x x))

  (my-sqr 5) => 25
```

- The most common circumstances where there is an implicit `begin` are:
  - The body of a *function* definition or `lambda`
  - The body of a `local`
  - The “answer” of a `cond` clause

- The previous example (defining a constant) is one of the few circumstances where `begin` is necessary, and we won’t see it very often

Implicit local

Similar to the implicit begin, Racket also uses an implicit local in the body of a function definition or lambda

Recall from CS135 your code to use Heron's formula to compute the area of a triangle with side lengths a, b, c:

```
(define (t-area a b c)
  (local
    [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

This is now equivalent to:

```
(define (t-area a b c)
   (define s (/ (+ a b c) 2))
   (sqrt (* s (- s a) (- s b) (- s c))))
```

The constant s, is implicitly local

Input and Output (I/O)

- In CS135, we designed functions with no user interaction
- Most programs interact with users and the “real world” via Input and Output (I/O)
- Racket displays (prints) *output* whenever there is a top-level expression in the program:

```
#lang racket
'four
"four"
(+ 2 2)
```

  prints ’four, "four" and 4

- This behaviour is **not** common: Most programming languages do not print values like this, and require special functions to display output

Basic Printing

- The `display` function prints output in the RunC (or DrRacket) interactions window:

  ```
  (display "Hello, World!\n")
  ```

  - Recall that "\n" is shorthand for generating a “newline”

- This is one example of how `begin` can be useful:

  ```
  (begin
    (display "The answer is ")
    (display (sqrt 2))
    (display " (approximately)\n"))
  ```

  prints:

  ```
  The answer is 1.4142135623730951 (approximately)
  ```

Formatted printing with `printf`

- The only other output function we introduce is `printf` (print **f**ormatted)
- The result of the previous example:

```
(display "The answer is ")
(display (sqrt 2))
(display " (approximately)\n")
```

  could be also be achieved with:

```
(printf "The answer is ~a (approximately)\n"
        (sqrt 2))
```

Formatted printing with `printf`

- `printf` consumes one or more arguments:
  - The first argument of `printf` must be a `String`
  - There must be an additional argument (of `Any` type) for each `"~a"` that appears in the string
  - Each `"~a"` is replaced by the value of the corresponding argument (in order) as if it was `display`ed

```
(printf "an example with ~a ~a ~a ~a items"
        "four" 'four (+ 2 2) 4)

        an example with four four 4 4 items
```

- `printf` supports several different display formats in addition to `"~a"`, but that is all we need in this course

Input

The function `(read)` can pause the current program, wait for keyboard input and then produce the value entered:

```
(define (sayhello)
  (define x (read))
  (printf "Hello ~a. Boy I love CS136.\n" x))
```

```
> (sayhello)
George
Hello George.  Boy I love CS136.
>
```

The read function

The `(read)` function is quite complicated, so we present a simplified overview:

- The `(read)` function may not pause the program and wait for input: if you typed multiple values before hitting [enter] during the first `(read)`, the second `(read)` automatically produces the second value
- `(read)` interprets your input as if a single quote `'` is inserted before whatever you type (not really, but close enough)
  - `1 => 1`
  - `one => 'one`
  - `"one"=> "one"`
  - `one two => 'one 'two [2 values]`
  - `(one two 3) => '(one two 3) [1 list]`

Let's Play a Game of Mad Libs

Try this in your RunC environment

```
(define (madlib)
  (printf "Let's play Mad Libs! Enter 4 words:\n")
  (printf "a Verb, Adverb, Noun & Adjective:\n")
  (define verb (read))
  (define adverb (read))
  (define noun (read))
  (define adj (read))
  (printf "The two ~as were too ~a to ~a ~a.\n"
          noun adj verb adverb))

(madlib)
```

Implicit begin with printf

The implicit `begin` and `printf` can really help debug (trace) your code:

```
(define (fib x)
  (cond [(<= x 2)
           (printf "Base case reached x = ~a\n" x)
           1 ]
          [else (printf "Recursive case with x = ~a\n" x)
                (+ (fib (- x 1))
                   (fib (- x 2)))]))
```

Try this with `(fib 10)`

The Contract for `display`

- `display` and `printf` are different than any other functions we have seen so far...
- They print to the display, but they produce *nothing*
- Racket uses the special value `#<void>` to represent nothing, and we will use `Void` in our contracts as a special type
- The contract for `display` is:
  `display: Any -> Void`
- Aside: You can't actually type the value `#<void>` into your program: use the function `(void)` instead

my-add

Consider the function `my-add`:

```
;; my-add:  Num Num -> Num
(define (my-add x y)
  (define answer (+ x y))
  (printf "~a plus ~a is ~a\n" x y answer)
  answer)
```

Every time the function is applied it also prints a message. The contract is technically correct, but it doesn't fully capture everything that happens.

That's because this function also produces a **side effect**.

Side Effects

- A function has a **side effect** when anything happens *in addition* to the function producing a value
- In other words, a side effect is when something in “the world” changes as a result of applying the function
- In the case of `display`, the interactions window output changes
- Aside: Some purists insist that a function with a side effect is no longer a function and use the term “procedure” or “routine”, but we will continue to use the term function

More Side Effects

Side effects can change "the world" in more ways than simply printing output, and we have seen some side effects before:

- The `define` special form adds new definitions to "the world"
- The `error` function stops a program from running
- Any top-level evaluation in Racket prints a value
- The `require` special form causes a new module to be evaluated

It is very important that we document any side effects our functions have.

To facilitate this, we will revisit our **contract** syntax.

Preconditions and Postconditions

- We extend our function contract syntax by adding sections for **preconditions** and **postconditions**
- The **preconditions** section lists all of the conditions that must be met before the function can be applied
  - Typically, the preconditions section identifies any restrictions on the arguments
- The **postconditions** section lists the conditions that will be met *if* the preconditions were met when the function was applied
  - The postconditions section describes what the function produces and identify any side effects
- These sections strengthen the contract: “If you follow my contract and meet my preconditions, I will promise the postconditions”

Full Contract for `my-add`

Often you can formally specify the pre- and post- conditions as logical (Boolean) statements, and so it is common practice to use `true` when no conditions exist.

```
;; my-add: Num Num -> Num
;;         PRE: true
;;         POST: produces x + y
;;               prints a message

(define (my-add x y)
  (define answer (+ x y))
  (printf "~a plus ~a is ~a\n" x y answer)
  answer)
```

More Contract Examples

```
;; sum-first: Int -> Int
;;            PRE: k >=1
;;            POST: produces an integer >= 1
(define (sum-first k) ...)
```

```
;; list-max:  (listof Num) -> Num
;;           PRE: lon is not empty
;;           POST: produces the largest number in lon
(define (list-max lon) ...)
```

Preconditions are a concise and convenient way to express restrictions on arguments, and postconditions can help make purpose statements more expressive.

Printing vs. Values

Because Racket prints the value of each top level expression, there can be some confusion between the two concepts.

```
;; take-headache-pill: -> String
;;                     PRE: true
;;                     POST: produces "Headache gone!"
;;                           prints Nausea
(define (take-headache-pill)
  (display "Nausea\n")
  "Headache gone!")
```

`(take-headache-pill)` produces the value `"Headache gone!"` and has the side effect of printing the word "`Nausea`".

Headaches

```
(define (take-headache-pill)
   (display "Nausea\n")
   "Headache gone!")
```

Interaction window:

```
> (take-headache-pill)
Nausea
"Headache gone!"
> (string-length (take-headache-pill))
Nausea
14
> (define my-state (take-headache-pill))
Nausea
> my-state
"Headache gone!"
```

Revisiting Module Interfaces with Side Effects

- Remember that the target audience for the *module interface* is the **client**
- You should describe side effects in the interface if they will affect the client's understanding of how the module is to be used, but you want to avoid any implementation-specific details
- The documentation you provide with the function body should include a detailed description of any side effects

More Side Effects

- As soon as “the world” can change, there is the possibility that a function may not produce the same value every application, even with *identical* arguments.
- The `current-seconds` and `random` functions may or may not have side effects, but they clearly indicate that “the world” is changing.

```
> (current-seconds) ;; number of seconds since 1970
1357840653
> (current-seconds) ;; wait 3 seconds
1357840656

> (random 6) ;; random value from 0..5
2
> (random 6)
5
```

Constant Definitions

Up to this point, the following has been a *constant* definition:

```
(define x 6)
```

and we have referred to `x` as a *constant* because the value of `x` will *always* be 6.

If you've read HtDP, you may have noticed that they refer to `x` as a *variable*, which seems silly because something that is variable is the *opposite* of something that is constant.

set!

The `set!` special form has the **ultimate** side effect:
It changes "the world".

```
(define x 6) ;; x => 6
...
(set! x 10) ;; x => 10
...
(set! x "what?") ;; x => "what?"
```

This is called **mutation**: The value of `x` above has *mutated* from it's original value (and even changed it's type!).

`set!` could have also been called `re-define` . The `!` in the name is like a giant caution from the Racket authors that `set!` should be used sparingly, and that mutation can be dangerous.

Like `display`, the `set!` special form produces `Void`.

Functions with memory

```
#lang racket
(provide remember recall)

(define mem 'nothing)
(define (remember x) (set! mem x))
(define (recall) mem)
```

```
> (remember "Buy milk")
> (recall)
"Buy milk"
> (remember "Brush your teeth")
> (recall)
"Brush your teeth"
```

list-max with mutation

```
;; list-max:  (listof Num) -> Num
;;            PRE: lon is not empty
;;            POST: produces the largest number in lon
(define (list-max lon)
  (define sofar (first lon))
  (define (list-max/mut lst)
    (cond [(empty? lst) (void)]
          [else (set! sofar (max sofar (first lst)))
                (list-max/mut (rest lst))]))
  (list-max/mut (rest lon))
  sofar)
```

Note that this solution is *not* the best one in Racket

- It is harder to reason about and demonstrate correctness
- It is potentially slower

The Passport office

We want to design a module to simulate a passport office.

When you arrive at the office you take a ticket and get a number, and then you patiently wait for your number to be served.

Our module interface should have two functions:

```
;; next-ticket: -> Nat
;;             PRE: true
;;             POST: increments & produces the ticket #
;;
;; next-serve: -> Nat
;;            PRE: true
;;            POST: increments & produces the service #
;;
;; Examples:
;; (next-ticket) => 1
;; (next-ticket) => 2
;; (next-serve) => 1
;; (next-ticket) => 3
;; (next-serve) => 2
```

Example: Passport office

```racket
#lang racket ;; passport.rkt

(provide next-ticket next-serve)

(define ticket 0)
(define serving 0)

(define (next-ticket)
  (set! ticket (add1 ticket)) ;; increment ticket
  ticket)

(define (next-serve)
  (set! serving (add1 serving)) ;; increment serving
  serving)
```

Module Security

What if someone wants to hack the system?

```
> (next-ticket)
3685
> (next-serve)
12
;; What!  I'll never get out of here..  hmm..
> (set!  serving 3684)
;; Heh heh heh.
> (next-serve)
3685
;; That's me!
```

This would work if all the code was in the same file, but because `serving` is local to the `passport.rkt` module, clients cannot do this.

Duplication of Effort

- The `next-ticket` and `next-serve` functions are pretty similar
- What if we want to add more queues?

```
(define another-counter 0)

(define (next-another)
  (set! another-counter (add1 another-counter))
  another-counter)
```

- This approach doesn't scale well
- Remember that we can have a function that produces another function...

Building A Counter Function

The following function produces another function:

```
(define (make-counter)
  (define count 0)   ; implicit local
  (define (next-counter)
    (set! count (add1 count))
    count)
  next-counter) ; produce the local helper function
```

Recall from CS 135 that each time we apply `make-counter`, it generates a new *(fresh)* version of `count` and `next-counter`.

```
(define my-counter (make-counter))
(my-counter) => 1
(my-counter) => 2...
```

Finally, we re-write `make-counter` with `lambda`, and create a new module.

Counter Module

```
#lang racket ;; counter.rkt
;; module for generating a counter
(provide make-counter)

;; make-counter: -> ( -> Nat)
;;               PRE: true
;;               POST: produces a function for incrementing
;;                     a counter
(define (make-counter)
  (define count 0)
  (lambda () (set! count (add1 count))
             count))
```

```
#lang racket ;; passport.rkt [very condensed]
(provide next-ticket next-serve)
(require "counter.rkt")
(define next-ticket (make-counter))
(define next-serve (make-counter))
```

Summary

- We were once again able to change the implementation of a module (`passport.rkt`) without changing it's interface
- Instead of just enhancing `passport.rkt`, we saw an opportunity to create the `counter.rkt` module, which could be re-used in other applications
- We have seen how *I/O*, *side effects* and *mutation* have changed our "world" and made it more complicated
- In the next unit, we build upon all of the ideas from this unit to design more complicated programs