CS450
Structure of Higher Level Languages
Lecture 4: Pairs and lists
Tiago Cogumbreiro
Today we will learn...

- data structures as constructors and accessors
- pairs
- lists
- user-data structures
Function definition

Racket introduces a shorthand notation for defining functions.

```
(define (variable+ term+)
```

A function definition expects one or more variables (symbols). The first variable is the function variable. The remaining variables are the arguments of the function declaration. The one-or-more terms consist of the body of the function declaration. Which is a short-hand for:

```
(define variable (lambda (variable* term+))
```
Exercise

The McCarthy 91 function was invented by computer scientist John McCarthy to motivate formal verification.

\[
M(n) = \begin{cases} 
  n - 10 & \text{if } n > 100 \\
  M(M(n + 11)) & \text{if } n \leq 100 
\end{cases}
\]

- Implement the function in Racket
- What is \(M(99)\)?
Exercise

The McCarthy 91 function was invented by computer scientist John McCarthy to motivate formal verification.

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

- Implement the function in Racket
- What is \( M(99) \)?

The McCarthy 91 function is equivalent to

\[ M(n) = n - 10 \text{ if } n > 100 \]
\[ M(n) = 91 \text{ if } n \leq 100 \]
Data structures
Data structures

When presenting each data structure we will introduce two sets of functions:

- **Constructors**: functions needed to build the data structure
- **Accessors**: functions needed to retrieve each component of the data structure. Also known as *selectors*.

Each example we discuss is prefaced by some unit tests. We are following a Test Driven Development methodology.
The pair datatype

Constructor: cons

\[
\begin{align*}
\text{expression} &= \cdots \mid \text{pair} \\
\text{pair} &= (\text{cons} \ \text{expression} \ \text{expression})
\end{align*}
\]

Function cons constructs a pair with the evaluation of the arguments, which Racket prints as: '(v1 . v2)

Example

```
#lang racket
(cons (+ 1 2) (* 2 3))
```

Output

```
$ racket pair.rkt
'(3 . 6)
```
The pair datatype

Accessors: car and cdr

- Function car returns the left-hand-side element (the first element) of the pair.
- Function cdr returns the right-hand-side element (the second element) of the pair.

Example

```racket
(define pair (cons (+ 1 2) (* 2 3)))
(car pair)
(cdr pair)

$ racket pair.rkt
3
6
```
Pairs: example 1

Swap the elements of a pair: \((\text{pair-swap } p)\)

Spec

; Paste this at the end of "pairs.rkt"
(require rackunit)
(check-equal?
 (cons 2 1)
 (pair-swap (cons 1 2)))
Pairs: example 1

Swap the elements of a pair: \((\text{pair-swap} \ p)\)

Spec

```racket
; Paste this at the end of "pairs.rkt"
(require rackunit)
(check-equal? (cons 2 1) (pair-swap (cons 1 2)))
```

Solution

```racket
#lang racket
(define (pair-swap p) (cons (cdr p) (car p)))
```
Pairs: example 2

Point-wise addition of two pairs: \((\text{pair+} \ 1 \ r)\)

Unit test

```
(require rackunit)
(check-equal?
  (cons 4 6)
  (pair+ (cons 1 2) (cons 3 4)))
```
Pairs: example 2

Point-wise addition of two pairs: \((\text{pair+ } l \ r)\)

Unit test

```
(require rackunit)
(check-equal? 
  (cons 4 6)
  (pair+ (cons 1 2) (cons 3 4)))
```

Solution

```
#lang racket
(define (pair+ l r)
  (cons (+ (car l) (car r))
       (+ (cdr l) (cdr r))))
```
Pairs: example 3

Lexicographical ordering of a pair

```
(require rackunit)
(check-true (pair< (cons 1 3) (cons 2 3)))
(check-true (pair< (cons 1 2) (cons 1 3)))
(check-false (pair< (cons 1 3) (cons 1 3)))
(check-false (pair< (cons 1 3) (cons 1 0)))
```
Pairs: example 3

Lexicographical ordering of a pair

```
(define
(pair< l r)
(or
(< (car l) (car r))
(and (= (car l) (car r))
(< (cdr l) (cdr r))))))
```

```
(require rackunit)
(check-true (pair< (cons 1 3) (cons 2 3))))
(check-true (pair< (cons 1 2) (cons 1 3))))
(check-false (pair< (cons 1 3) (cons 1 3))))
(check-false (pair< (cons 1 3) (cons 1 0))))
```
Lists
Lists

Constructor: `list`

```
expression = ⋯ | list
list = (list expression* )
```

Function call `list` constructs a list with the evaluation of a possibly-empty sequence of expressions $e_1$ up to $e_n$ as values $v_1$ up to $v_n$ which Racket prints as: `(v_1 \ldots v_2)`

```
#lang racket
(list (+ 0 1) (+ 0 1 2) (+ 0 1 2 3))
(list)
```

```
$ racket list-ex1.rkt
'(1 3 6)
'()
```
Accessing lists

Accessor: empty?

You can test if a list is empty with function empty?. An empty list is printed as '().

```racket
#lang racket
(require rackunit)
(check-false (empty? (list (+ 0 1) (+ 0 1 2) (+ 0 1 2 3))))
(check-true (empty? (list)))
```
Lists are linked-lists of pairs

Accessors: car, cdr

Lists in Racket are implemented as a linked-list using pairs terminated by the empty list `'( ).

- **Function** car returns the head of the list, given a nonempty list. car originally meant Contents of Address Register.
- **Function** cdr returns the tail of the list, given a nonempty list. cdr originally meant Contents of Decrement Register.

```
(list 1 2 3 4)
```
Lists are built from pairs example

Constructor empty

```racket
#lang racket
(require rackunit)
(check-equal? (cons 1 (cons 2 (cons 3 (cons 4 empty)))) (list 1 2 3 4))
```