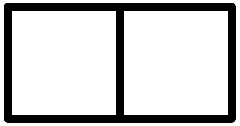
CS 360
Programming Languages
Day 3

Review

- Cons cell: two-piece structure (like a 2-member class in Java)



  - Also called a pair. left side called "car"; right side called "cdr"
  - **(cons *e1 e2*)** constructs a new cons cell (and returns it)
  - **(car *e*)** returns the car part of ***e***; **(cdr *e*)** returns the cdr of ***e***

- **'(v1 . v2)** constructs a "literal" cons cell.
- Drawing cons cells:
  - **(cons 1 2)**
  - **(cons 1 (cons 2 3))**
  - **(cons (cons 1 2) 3)**

Box-and-pointer notation with lists

- Key to differentiating pairs from lists: lists never have dots in them.
- **`'(1 . 2)`** versus **`'(1 2)`**

- How would you create **`'(1 . 2)`** with call(s) to cons?

- How would you create **`'(1 2)`** with call(s) to cons?

- What does **`(cons 1 '(2 3))`** create?

- What does **`(cons '(1) '(2 3))`** create?

Review

Huge progress in two lectures on the core pieces of Racket:

- Variables
  - `(define variable expression)`
- Functions
  - Build: `(define (f x1 x2 …) e)`
  - Use: `(f e1 … en)`
- Pairs
  - Build: `(cons e1 e2)` OR `'(v1 . v2)`
  - Use: `(car e)`, `(cdr e)`
- Lists
  - Build: `'()` `(cons e1 e2)` OR `'(v1 v2 v3 …)`

    `(list e1 e2 …)` `(append e1 e2 …)`
  - Use: `(null? e)` `(car e)` `(cdr e)`

The `cond` expression

We have two "if-then-else" expressions in Racket:

- `(if test e1 e2)`
  - evaluates to `e1` if test is `#t`, otherwise evaluates to `e2`.

-
  ```
  (cond (test1 e1)
        (test2 e2)
        ...
        (#t en))
  ```
  - evaluates to `e1` if `test1` is `#t`
  - evaluates to `e2` if `test2` is `#t`
  - (etc)
  - evaluates to `en` if all prior tests are `#f`
  - The last `#t` clause is optional, but is useful as an "else".

Processing nested lists

```
(define (length lst)
  (if (null? lst) 0
    (+ 1 (length (cdr lst)))))
```

```
(define (length-nested lst)
  (cond ((null? lst) 0)
        ((list? (car lst))
          (+ (length-nested (car lst))
             (length-nested (cdr lst))))
        (#t (+ 1 (length-nested (cdr lst))))))
```

Other useful functions and reminders

- **`(and e1 e2...)`**
- **`(or e1 e2...)`**
- **`(not expr)`**
  - e.g., **`(not (= a b))`**
- **`(remainder x y)`**
  - returns remainder of **`x`** divided by **`y`**
- Remember the differences between **`cons`**, **`list`**, and **`append`**:
- **`(cons item lst)`**
  - makes a new list with **`item`** as the first element, and the items in **`lst`** as the rest of the list.
- **`(list a b c...)`**
  - makes a new list of (**`a b c...`**)
- **`(append lst1 lst2...)`**
  - makes a new list of the items inside of **`lst1`**, then the items inside of **`lst2`**...

Syntax and Semantics

- **Syntax** are the “form rules” for a language
  - Can be thought of as grammar rules
  - Defines valid and invalid statements
  - Ex: **`round(x)`** is good syntax in Python but bad syntax in Racket

- **Semantics** are the “meaning rules” for a language
  - Defines meaningful statements
  - Ex:

    ```
    (car (cdr pairs))
    (cdr (car pairs))
    ```

  - Same exact syntax (nested functions), wildly different meanings

Interpreted vs Compiled Languages

- Compiled languages are translated “all at once” into machine code by a special program called the compiler, to be executed later
  - Generally faster
  - Source code stays private
  - Probably won’t work across platforms

- Interpreted languages are translated “line by line” into machine code and executed by a special program called the interpreter
  - Generally slower
  - Source code is usually public
  - Requires end user to have the interpeter

Types

C++ uses ***static typing***: most code can be checked at compile-time to make sure rules involving types are not violated.

```
int double(int n) {
   return 2 * n;
}
```

Python uses ***dynamic typing***: most code cannot be checked for type errors at compile-time; this has be delayed until run-time.

```
def double(n):
   return 2 * n
```

Dynamic typing

- Racket (like most Scheme or Lisp dialects) is dynamically typed.

- Some characteristics of dynamic typing:
  - Values have types, but variables do not.
    - A variable can refer to different types during its lifetime.

  - Most type-error bugs cannot be found before the program is run, and not until the offending line of code is encountered.
    - Possible to write code with type errors that aren't discovered for a long time, if buried in code that isn't executed often.

  - Traditionally (but not always), dynamically-typed languages are interpreted, whereas statically-typed languages are compiled.

"Manual" type-checking

- Dynamically-typed languages often have some way for the programmer to discover the type of a variable.
- In Racket (all of these return **`#t`** or **`#f`**):
  - **`number?`**
    - `also` **`integer?, rational?, real?`**
  - **`list?`**
  - **`pair?`**
  - **`string?`**
  - **`boolean?`**

- Enables a single function to do different things depending on the type of an argument.

Length of a list vs length of nested lists

- For "regular" list
  - if empty list, return 0
  - else return 1 + length of the cdr of the list.

- For a list with possible nested lists…
  - if empty list, return 0
  - if the car of the list is a list…   do what?
  - else (car is not a list)…   do what?

Length of a list vs length of nested lists

```
(define (length-nested lst)
  (cond ((null? lst) 0)
        ((list? (car lst))
           (+ (length-nested (car lst))
              (length-nested (cdr lst))))
        (#t (+ 1 (length-nested (cdr lst))))))
```

Side effects

- In programming, a function has a side effect if it modifies some state or has an observable interaction with functions outside of itself (other functions or the outside world).
- Mutation is an example of a side effect.
  - Also: printing to the screen, modifying files, etc
- Functional programming (in Racket, Scheme, LISP) traditionally avoids side effects as much as possible.
  - Makes it much simpler to reason about how a program works.
  - Without side effects, calling a function with a fixed set of arguments is guaranteed to always return the same value.

Side effects

- In Racket, function bodies may contain more than one expression, if the extra expressions ***come first and are evaluated only for their side effects.***
  - In "pure" functional programming, you don't have side effects.
  - But it's nice to have this facility at times.
  - For debugging, can use (displayln <whatever>) and (newline)
- Example:

```
(define (length lst)
    (displayln lst)
    (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```