Programming Languages

Lecture 3

Two other ways to build lists

- **`list`** function
  - Makes a list out of all arguments.
  - Arguments can be of any data type.
  - **`(list e1 e2 … en)`** evaluates **`e1`** through **`en`** to values **`v1`** through **`vn`**; returns the list **`'(v1 v2 … vn)`**.
- **append** function
  - Concatenates values inside lists given as arguments.
  - Arguments *must* be lists.
  - **`(append e1 e2 … en)`** evaluates **`e1`** through **`en`** to values **`v1`** through **`vn`**;
  - If **`v1`** = **`(v11 v12 … )`** and **`v2`** = **`(v21 v22 … )`** etc, then return value is **`(v11 v12 … v21 v22 … )`**.

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)`**

Today

- The big thing we need: local bindings
  - For style and convenience
  - A big but natural idea: nested function bindings

- Why not having mutation (assignment statements) is a valuable language feature
  - No need for you to keep track of sharing/aliasing, which C++ (and sometimes Python) programmers must obsess about
  - What makes global variables "bad" in most languages (languages that allow mutation)

Let-expressions

The construct for introducing local bindings is ***just an expression***, so we can use it anywhere we can use an expression

- Syntax: `(let ((var1 e1) (var2 e2) …) e)`
  - Each $\mathtt{var}_i$ is any *variable name,* each $\mathtt{e}_i$ is any *expression,* and **e** is also any *expression*.
- Evaluation: Evaluate each $\mathtt{e}_i$, assign each $\mathtt{e}_i$ to $\mathtt{var}_i$ (all at once) in an environment that includes the bindings from the enclosing environment.
- Result of whole let-expression is result of evaluating **e** in the new environment.

Silly examples

```
(define (silly1 z)
  (let ((x 5))
      (+ x z)))

; this one won't work!
(define (silly2 z)
  (let ((x 5) (answer (+ x z)))
      answer))

(define (silly2-fixed z)
  (let* ((x 5) (answer (+ x z)))
      answer))
```

Silly examples

```
(define (silly3 z)
  (let* ((x (if (> z 0) z 4)) (y (+ x 1)))
      (if (> x y) (* 2 x) (* y y))))

(define (silly4)
  (let ((x 1))
      (+
            (let ((x 2)) (+ x 1))
            (let ((y (+ x 2))) (+ y 1)))))
```

`silly4` is poor style but shows let-expressions are expressions

- Could also use them in function-call arguments, parts of conditionals, etc.
- Also notice shadowing

What’s new

- What’s new is *scope*: contexts within a program where a variable has a value.
  - Variables bound using **`let`** can be used in the body of the let-expression.
  - Variables bound using **`let*`** can be used in the body of let-expression and in later bindings in the same **`let*`**.
  - Bindings in **`let`**/**`let`*** *shadow* bindings of the same variable name from the enclosing environment(s).

- *Nothing else is new!*

Nested functions

- Good style to define helper functions inside the functions they help if they are:
  - Unlikely to be useful elsewhere
  - Likely to be misused if available elsewhere
  - Likely to be changed or removed later

- A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later

- But we need some additional syntax…

Nested functions

- let and let* don't let you define function bindings using the same variations that define does:
  - **`(define var expr)`** OK
  - **`(define (func x1 x2…) body-expr)`** OK
  - **`(let ((var expr) (var expr)…) expr)`** OK
    - Can't do **`(let (((func x1 x2…) body-expr) …) expr)`** NO

  - Note that define statements are *not* expressions, so they don't evaluate to values.
  - Can't do **`(let ((func (define …`** NO

Solution: internal defines

```
(define (f (x1 x2 … xn)
  (define (f1 (y1 y2 … yn) expr)
  (define (f2 (z1 z2 … zn) expr)
  expr)
```

- How does this not conflict with the idea of function bodies only having one expression?
- An additional define is NOT an expression.
  - Expressions can be evaluated to values.
  - Defines are not expressions, and have no values.