Lecture #2 (plus practice session)

There are two **Boolean values** built-in in Racket:

`#t' (true) and `#f' (false). They can be used in `if' statements, for example:

```
 (if (< 2 3) 10 20)  -->  10
```

because (< 2 3) evaluates to `#t'.

As a matter of fact, ***any*** value except for `#f' is considered to be **true**, so:

```
  (if 0 1 2)  -->  1

  (if "false" 1 2)  -->  1

  (if "" 1 2)  -->  1

  (if null 1 2)  -->  1  ; null is also a built-in value

  (if #f 1 2)  -->  2       ; the only false value
```

Note: Racket is a **functional language** -- so **everything** has a value.

This makes the expression

```
  (if test consequent)
```

have no meaning when "test" evaluates to `#f'. This is unlike Pascal/C, where statements **do something** (side effect) like printing or an assignment -- here an if-statement with no

alternate part will just "do nothing" if the test is false...

Racket, however, must return some value -- it could decide on simply returning `#f' (or some unspecified value) as the value of

```
(if #f something)
```

as some implementations do, but Racket just declares it a syntax error.
(As we will see in the future, Racket has a more convenient `when' with a clearer intention.)

Well, *almost* everything has a value...

There are certain things that are part of Racket's syntax -- for example `if' and `define' are **special forms**, they do not have a value! More about this shortly.

(Bottom line: much more things do have a value, compared with other languages.)

`cond' is used for a sequence of `if...else if...else if...else'. The problem is that **nested `if's are inconvenient.** For example,

```
(define (digit-num n)
  (if (<= n 9)
    1
    (if (<= n 99)
      2
      (if (<= n 999)
        3
        (if (<= n 9999)
          4
          "a lot")))))
```

In C/Java/Whatever, you'd write:

```
function digit_num(n) {
  if (n <= 9)          return 1;
  else if (n <= 99)    return 2;
  else if (n <= 999)   return 3;
  else if (n <= 9999)  return 4;
  else return "a lot";
}
```

- (Side question: why isn't there a `return' statement in Racket?)

Bad indentation in Racket – DON'T DO

Trying to force Racket code to look similar:

```
(define (digit-num n)
  (if (<= n 9)
    1
  (if (<= n 99)
    2
  (if (<= n 999)
    3
  (if (<= n 9999)
    4
    "a lot")))))
```

is more than just bad taste -- the indentation rules are there for a reason, the main one is that you can see the structure of your program at a quick glance, and this is no longer true in the above code. (Such code will be penalized!)

So, instead of this, we can use Racket's `cond' statement, like this:

```
(define (digit-num n)
  (cond [(<= n 9)    1]
        [(<= n 99)   2]
        [(<= n 999)  3]
        [(<= n 9999) 4]
        [else        "a lot"]))
```

Note that `else' is a keyword that is used by the `cond' form – you should always use an `else' clause (for similar reasons as an `if', and we will need it when we use a typed language).

[Square brackets] are read by DrRacket like round parens, it will only make sure that the paren pairs match. We use this to make code more readable -- specifically, there is a major difference between the above use of "[]" from the conventional use of "()". Can you see what it is?

The general structure of a `cond':

```
(cond [test-1 expr-1]
      [test-2 expr-2]
      ...
      [test-n expr-n]
      [else   else-expr])
```

Example for using an if-statement, and a recursive function (not tail recursive):

```
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))
```

Use this to show the different tools, esp:

- special objects that *cannot* be used
- syntax-checker
- stepper

An example of converting it to tail recursive form:

```
(define (helper n acc)
  (if (zero? n)
    acc
    (helper (- n 1) (* acc n))))

(define (fact n)
  (helper n 1))
```

Lists & Recursion

Lists are a **fundamental** Racket data type.

A list is defined as either:

1. the **empty list** (`null', `empty', or '()),

2. a **pair** (`cons' cell) of anything and a **list**.

As simple as this may be, it gives us precise *formal* rules to prove that something is a list.

(Question: Why is there a "the" in the first rule?)

Examples:

```
null
(cons 1 null)
(cons 1 (cons 2 (cons 3 null)))
(list 1 2 3) ; a more convenient function to get the above
```

List operations -- predicates:

```
null? ; true only for the empty list
pair? ; true for any cons cell
list? ; this can be defined using the above
```

We can derive `list?' from the above rules:

```
(define (list? x)
  (if (null? x)
    #t
    (and (pair? x) (list? (rest x)))))
```

Or better yet…

```
(define (list? x)
  (or (null? x)
      (and (pair? x) (list? (rest x)))))
```

But why can't we define `list?' more **simply** as

```
(define (list? x)
  (or (null? x) (pair? x)))
```

The difference between the above definition and the proper one can be observed in the full Racket language, not in the student languages (where, there are no pairs with non-list values in their tails).

List operations -- destructors for pairs (cons cells):

```
first
rest
```

Traditionally called `car', `cdr'.

Also, any `c<x>r' combination for <x> that is made of up to four `a's
and/or `d's -- we will probably not use much more than `cadr', `caddr'
etc.

Example for recursive function involving lists:

```
(define (list-length list)
  (if (null? list)
    0
    (+ 1 (list-length (rest list)))))
```

Use different tools, esp:
* syntax-checker
* stepper

Question: How come we could use `list' as an argument? -- use the syntax checker

---

```
(define (list-length-helper list len)
  (if (null? list)
    len
    (list-length-helper (rest list) (+ len 1))))

(define (list-length list)
  (list-length-helper list 0))
```

Main idea: lists are a **recursive structure**, so **functions** that operate on lists **should be recursive** functions that follow the recursive definition of lists.

Another example for list function -- summing a list of numbers

```
(define (sum-list l)
  (if (null? l)
    0
    (+ (first l) (sum-list (rest l)))))
```

Also show how to implement `rcons', using this guideline.

More examples:

Define `reverse' -- solve the problem using `rcons'.

`rcons' can be generalized into something very useful: `append'.

* How would we use `append' instead of `rcons'?

* How much time will this take? Does it matter if we use `append' or
  `rcons'?

Redefine `reverse' using tail recursion.

* Is the result more complex? (Yes, but not too bad because it collects the elements in reverse.)
