Functions, Cons Cells and Lists

adapted from materials by Dan Grossman at the University of Washington

Function definitions

Functions: most important building block in 251
- Like Java methods, have arguments and result
- But no classes, **`this`**, **`return`**, etc.

Example *function definition*:

```
(define (pow base exp)
    (if (< exp 1)
        1
        (* base (pow base (- exp 1)))))
```

Note: The *body* includes a (recursive) *function call*:
`(pow base (- exp 1))`

Recursion

- If you're not yet comfortable with recursion, you will be soon ☺
  - Will use for most functions taking or returning lists
  - And many other things
- Calls to same function solve "simpler" problems
- We will not use a single loop until sometime late in the semester, at least 3 languages later.
  - Loops often (not always) obscure simple, elegant solutions

Rewinding a bit…

Function definitions: 2 questions

- Syntax: **`(lambda`** `(x1 ... xn) e)`
  - *x1* through *xn* are variable names, called the **parameters**
  - *e* is any expression, called the **body**
- Evaluation: ***A function is a value!***
  - Evaluates to itself.
  - (No evaluation of "inside" yet.)
- Anonymous function -- no name, no binding.

Function calls

A new kind of expression: 2 questions

Syntax: ***(e0 e1 ... en)***

Evaluation:

1. (Under current dynamic environment,) evaluate **e0** through **en** to values **v0, v1, …, vn.**
2. If **v0** is not a function **(lambda (x1 ... xn) e)** expecting *n* arguments, raise an error.
3. Result is evaluation of ***e*** in an environment extended to map **x1** to **v1**, …, **xn** to **vn**
   - ("An environment" is the environment where the function was defined -- for now we are careful to avoid problems, later we define more precisely.)

6

Example function call

- `((lambda (x) (* x x)) (- 12 8))`
- Evaluation derivation?
  - Assume initial environment is empty.

- What about recursion?

7

Function bindings and recursion

- How do we get named functions? (Function bindings?)
- A function binding is an expression, so use existing tools:

```
(define  square
    (lambda  (x)
        (* x x)))
```

- **define** adds binding to function's environment
  - omitted earlier, supports **recursion**

```
(define  pow
    (lambda  (base  exp)
        (if  (< exp 1)
             1
             (* base  (pow base  (- exp 1))))))
```

8

Syntactic sugar

- Function bindings are very common -- shorter syntax, but meaning identical to earlier syntax.

```
(define  (pow base exp)
    (if  (< exp 1)
         1
         (* base  (pow base  (- exp 1)))))
```

- ***Syntactic sugar***: define handy simpler syntax for common idiom.
  - Implemented via textual translation to existing features.
  - *i.e.*, not a a few *feature.*

9

Have we seen function applications already?

- Surprise! What else looks like a function application?

- What looks like a function application but really is not?

10

Cons cells and lists

So far: numbers, booleans, conditionals, variables, functions
- Now ways to build up data with multiple parts
- This is essential
- Java examples: classes with fields, arrays

Now:
- *Cons cells*: pairs of values

Then:
- *Lists*: any "number of pieces" that all have the same type

11

Cons cells (2-tuples)

Need a way to *build* cons cells and a way to *access* the pieces

Build:
- **`cons`** function, takes 2 arguments

Access:
- **`car`** function, takes 1 argument
  - returns first part if argument is a cons cell
- **`cdr`** function, takes 1 argument
  - returns second part if argument is a cons cell

Terms: cons for construct, car/cdr historical accident
- *mnemonic: car precedes cdr in alpha-order, accesses 1st part*

12

Examples

Functions can take and return pairs

```
(define (swap-pair pair)
    (cons (cdr pair) (car pair)))


(define (sort-pair pair)
    (if (< (car pair) (cdr pair))
        pair
        (swap pair)))
```

13

Nesting

Cons cells can contain anything, including other cons cells

```
(car (cdr (cons 1 (cons 2 3))))
```

14

Lists

A cons cell has only two pieces.

A list can have any number of elements.

Need ways to *build* lists and *access* the pieces…

15

Lists are built of cons cells

- The empty list is a value:

```
null
```

- A non-empty list is a cons cell hold a value in the **car** and a list in the **cdr.**

```
(cons 1 (cons 2 (cons 3 null)))
```

- **Recursive structure, recursive functions!**
- Useful for list functions: **null?** function returns #t if its argument is **null.**

16