Functions, Cons Cells and Lists

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

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:

\[
\text{(define (pow base exp)}
\text{  (if (< exp 1))}
\text{    1}
\text{    (* base (pow base (- exp 1))))})
\]

Note: The body includes a (recursive) function call:

\[
\text{(pow base (- exp 1))}
\]

Rewinding a bit...

Function definitions: 2 questions

- Syntax: \(\text{(lambda (x1 ... xn) e)}\)
  - \(x1\) through \(xn\) are variable names, called the \text{parameters}
  - \(e\) is any expression, called the \text{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: \((e_0 \ e_1 \cdots \ e_n)\)

Evaluation:
1. (Under current dynamic environment,) evaluate \(e_0\) through \(e_n\) to values \(v_0, v_1, \ldots, v_n\).
2. If \(v_0\) is not a function \((\text{lambda} \ (x_1 \ldots x_n) \ e)\) expecting \(n\) arguments, raise an error.
3. Result is evaluation of \(e\) in an environment extended to map \(x_1\) to \(v_1\), \(x_2\) to \(v_2\), \ldots, \(x_n\) to \(v_n\)
   - ("An environment is the environment where the function was defined -- for now we are careful to avoid problems, later we define more precisely.

Example function call

- \(((\text{lambda} \ (x) \ (* \ x \ x)) \ (- \ 12 \ 8))\)
- Evaluation derivation?
  - Assume initial environment is empty.
  - What about recursion?

Function bindings and recursion

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

\[
(\text{define} \ \text{square} \\
 (\text{lambda} \ (x) \\
 (* \ x \ x)))
\]

- \text{define} adds binding to function's environment
  - omitted earlier, supports recursion

\[
(\text{define} \ \text{pow} \\
 (\text{lambda} \ (base \ exp) \\
 (\text{if} \ (< \ exp \ 1) \\
 1 \\
 (* \ base \ (\text{pow} \ base \ (- \ exp \ 1)))))
\]

Syntactic sugar

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

\[
(\text{define} \ (\text{pow} \ base \ exp) \\
 (\text{if} \ (< \ exp \ 1) \\
 1 \\
 (* \ base \ (\text{pow} \ base \ (- \ exp \ 1))) )
\]

- \text{Syntactic sugar}: define handy simpler syntax for common idiom.
  - Implemented via textual translation to existing features.
  - \text{i.e.,} not a a few feature.
Have we seen function applications already?

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

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

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

Examples

Functions can take and return pairs

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

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

Cons cells can contain anything, including other cons cells

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

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...

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.