Programming Paradigms

Week 7 Lecture 1

Finlay Smith

Finlay.smith@universityofgalway.ie

Recap 1/2

- **Functional Programming**
  - Lisp / scheme / racket
  - S-expressions
  - Nested lists vs cons pairs
  - Cons, Car, cdr (construct, first, rest)
  - Define:
    - Variables
    - Functions

Recap 2/2

- **Functional Programming cont.**
  - (list …)
  - (append …)
  - (if expr expr expr)
  - (cond expr expr expr)

Recursion

Recursion

- In scheme, control is achieved through recursion.
- Recursion: Functions that reference themselves
- A recursive function A is one that calls itself or calls another function which calls A.

Recursion

Recursive function

```
function a(args):

        //Do something

        a(args)
```

Mutual Recursion

```
function b(args):
        //do something
        c(args)

function c(args):
        //do something
        b(args)
```

Recursion

As with iteration, must ensure that a recursive function will not continue to run indefinitely. To ensure this, if using recursion we must ensure that:

1. There are certain criteria, called base criteria, for which the function does not call itself.
2. Each time the function does call itself (directly or indirectly), it must be closer to the base criteria.

A recursive function with these two properties is said to be well defined.

Recursion

Infinite iteration (loop)

```
int i = 0;
while(i < 10):
        print(“hello iteration”)
```

Will run forever.

Infinite recursion:

```
function b(int i):
        print(“hello recursion”)
        b(i)
```

Will either:

- Run forever*
- Cause a stack overflow.

* We’ll talk about tail recursion next week

Well Defined recursion:

- **Base Case:** Must have a condition for which the function will not call itself (will stop)
- **Reduce:** For each recursive call, must move towards the base case (reduce)

Activations

With recursion, we have the illusion of multiple copies of the procedure/function existing.

These are referred to as **activations.** These activations appear and disappear as the algorithm advances.

A number of activations may exist at the same time. Of the activations existing at any given time, only one is actively progressing.

The others are effectively in limbo, each waiting for another activation to terminate before it can terminate.

General Approach to solving problems recursively:

1. What is the base case?
2. What should the answer be when we are at the base case?
3. How do you reduce to get to this base case ? (often taking the cdr of a list)
4. What other work needs to be done for each function call?
    a. (e.g., creating a new list, etc.)?
5. How can these steps be put together?

Base cases:

- For numbers: often 0 or 1
  - (= var 0)
- For lists: often when list is empty
  - (empty? lst)
- Note: May have more than one base case to check

Mistakes to avoid

- Wrong number of arguments passed to function - they have to match
- Not having base case.
  - Or a base case that is impossible to reach.
- Not reducing to base case.