Efficiency and Formulating Abstractions with Higher Order Procedures

Abelson & Sussman & Sussman
sections:(first part 1.2) & 1.3

Lecture contents

- In this lecture we will look at:
- The processes generated by the evaluation of recursive definitions (A&S 1.2)
  - "Iterative" vs "recursive" definitions.
  - Efficiency
- Formulating abstractions with higher-order procedures(A&S 1.3)
  - procedures as arguments
  - constructing procedures using lambda
  - procedures as general methods
  - procedures as returned values
- We will defer the underlined topics until next lecture

Why understanding process is important

- When writing a program, it is important to be able to visualise what the program will do when it runs.
- Without this knowledge you cannot tell if a program does what you want it to do.
  - or does its job efficiently!
- Recursive definitions describe how a computational process gets from one stage to the next.
  - It describes local processing.
- You, as the programmer, need to understand how these elements of local processing are joined together to form a global process.

Example: Factorial

- Recursive definitions can have different efficiencies.
- Consider the following definition of factorial:

```
(define (factorial n)
   (if (= n 1)
        1
       (* n (factorial (- n 1)))))
```

- This definition makes one recursive call to factorial and then multiplies the result by n.
- The value of n must be recorded somewhere so we know what to multiply by when the recursive call: factorial(n-1) returns.
  - We must also remember that the next thing we have to do, after returning, is multiplication.

Example: Factorial(cont’d)

- Sample evaluation:

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

- This process is linear-recursive.
  - some processing is done after each recursive call.

A more efficient factorial

- A more efficient version of factorial:

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

- fact-iter does not perform any computation after the recursive call
  - No extra information to remember.
  - That is, it is tail-recursive

A more efficient Factorial (cont’d)

- Sample evaluation of new version:

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

- This process is linear-iterative.
  - much more efficient
  - carries the state of the computation in the parameters.
- Note that both definitions are recursive but only the first one generates a recursive process.

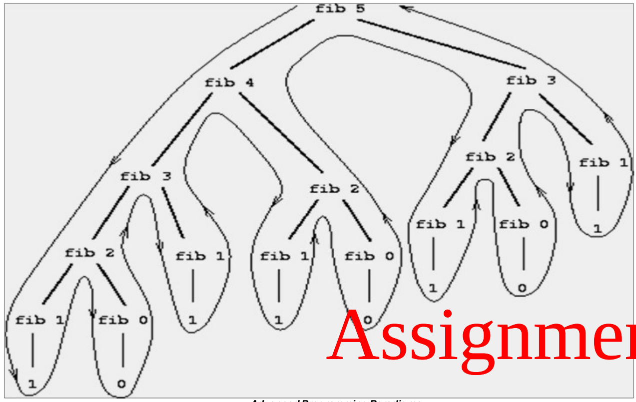
Tree recursion

- In the last example, both definitions made a maximum of one recursive call for each recursive call.
  - The term *linear* is derived from this (calls form a straight line)
- It is possible for each recursive call to generate > 1 recursive calls.
- This is called Tree recursion.
- Example: generator for n^{th} Fibonacci number:

```
(define (fib n)
   (cond ((= n 0) 0)
         ((= n 1) 1)
         (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

Tree recursion

- This definition generates the following (inefficient) process for (fib 5).



Efficiency

- The last solution was very inefficient.
  - can you see why?
- A more efficient definition is:

```
(define (fib n)
    (fib-iter 1 0 n))

(define (fib-iter a b count)
   (if (= count 0)
   b
   (fib-iter (+ a b) a (- count 1))))
```

- Note the use of variables to hold:
  - The last two results
  - and a count of the new numbers required
- This version is O(n) efficient. The last version was O(k^n)

Recall complexity analysis from data structures courses

Consider performing 10^6 operations per sec with problem size 10^5

TABLE 4.1 EXECUTION TIMES ($n=10^5$ INPUT VALUES, ASSUMING 10^6 OPERATIONS PER SECOND)

Function	Running Time
2^n	More than a century
n^3	31.7 years
n^2	2.8 hours
$n\sqrt{n}$	31.6 seconds
$n \log n$	1.2 seconds
n	0.1 seconds
$\sqrt{n}$	3.2×10^{-4} seconds
$\log n$	1.2×10^{-5} seconds

Example - Counting Change

- Recursion is a powerful programming tool.
- Consider the problem of counting the number of ways to change a certain amount of money with a certain set of coins.
  - So, for example, we could ask the question:

  How many ways can you change $2.10 given the denominations:

  5c, 10c, 20c,50c,$1 and $2?
- Where do you start with a solution to this problem?

Solution specification

- The following statement is true:
- The number of ways to count an amount *a* using *n* kinds of coins equals.
  - The number of ways to change amount *a* using all but the first kind of coin plus.
  - The number of ways to change amount *a-d* using all *n* kinds of coins, where *d* is the denomination of the first coin.
- This provides a way sub-divide a problem. Now we need to know when to stop this subdivision process.
- We stop when:
  - *a* is exactly 0c, there is exactly one way to change 0c.
  - *a* < 0c, there are zero ways to change <0c.
  - *n* is 0, there are zero ways to change any amount of money.

Counting Change: code

```
(define (count-change amount) (cc amount 5))

(define (cc amount kinds-of-coins)
   (cond ((= amount 0) 1)
         ((or (< amount 0) (= kinds-of-coins 0)) 0)
         (else (+ (cc amount (- kinds-of-coins 1))
                  (cc (- amount
                   (first-denomination kinds-of-coins))
                  kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
   (cond ((= kinds-of-coins 1) 5)
         ((= kinds-of-coins 2) 10)
         ((= kinds-of-coins 3) 20)
         ((= kinds-of-coins 4) 50)
         ((= kinds-of-coins 5) 100)
         ((= kinds-of-coins 6) 200)))
```


Problems (section 1.2)

- To understand the conditions that terminate the recursion in the change problem, trace *completely* a simple example, such as making change for 25 cents from 5 and 10 cent coins.
- The code in the last example is not very efficient. Define the order of efficiency of this code.
- Describe how getting the program to remember the results that it generates could make the algorithm more efficient. How much more efficient?
- Exercises 1.11 and 1.12

Abstractions with Higher-Order Procs (1.3)

- One of the most useful things about programming languages is the ability to attach labels to code.
  - It helps document code.
  - It makes programming a lot easier
  - scheme supports this using the `define` keyword.
- In Scheme, we can also treat code as a value - implications:
  - we can give values a label (as above), but we can also...
  - pass values into procedures
  - return values as results of procedures
  - combine values using operations
- Scheme allows procedures to be treated as values by supporting the first three capabilities directly.
  - given this, we can define our own operations on procedures.

Abstractions with Higher-Order Procedures

- Procedures that treat other procedures as data are called higher-order procedures.
- Higher-order procedures add another level of expressive power to a programming language.
  - We can define our own patterns of computation.
  - We can then “plug” the procedures of our choice into these patterns.
  - In most languages, these patterns are built into the language and cannot be changed.

Patterns of computation

- Consider the following examples:

```
(define (sum-integers a b)
   (if (> a b)
       0
       (+ a (sum-integers (+ a 1) b))))
(define (sum-cubes a b)
   (if (> a b)
       0
       (+ (cube a) (sum-cubes (+ a 1) b))))
(define (pi-sum a b)
   (if (> a b)
     0
     (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

- Notice the common elements of these definitions.

Capturing patterns of computation.

- The pattern followed by the preceding definitions is:

```
define (name a b)
   (if (> a b)
     0
     (+ (term a) (name (next a) b)))
```

- The scheme function defining this pattern is:

```
(define (sum term a next b)
   (if (> a b)
        0
       (+ (term a) (sum term (next a) next b))))
```

- Now we have a general-purpose procedure that captures the concept of summation.
  - and also allows us to write shorter, less repetitive, code.

Using patterns of computation

- Now use our general-purpose procedure to define
- The sum of integers:

```
(define (identity x) x)
(define (sum-integers a b)
   (sum identity a inc b))
```

- The sum of cubes:

```
(define (inc n) (+ n 1))
(define (sum-cubes a b)
    (sum cube a inc b))
```

- and many others
  - see exercise 1.31, 1.32 and 1.33