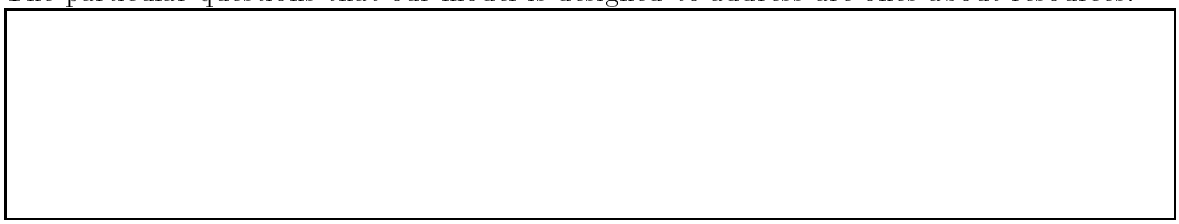
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1997

Lecture Notes – Feb. 11, 1997

Models of Computation

We need models to understand the evolution of a computational process under the direction of procedures. Our initial model is an informal substitution model.

The particular questions that our model is designed to address are ones about resources:



Here are the rules for the model:

- "Elementary expressions" are left alone: Elementary expressions are
  - numerals,
  - initial names of primitive procedures,
  - lambda expressions, naming procedures.
- A name bound by DEFINE: Rewrite the name as the value it is associated with by the definition.
- IF: If the evaluation of the predicate expression terminates in a non-false value
  - then rewrite the IF expression as the value of the consequent expression,
  - otherwise, rewrite the IF expression as the value of the alternative expression.
- Combination – (<operator> <operand1> ... <operandn>):
  - Evaluate the operator expression to get the procedure, and evaluate the operand expressions to get the arguments.
  - If the operator names a primitive procedure, do whatever magic the primitive procedure does.
  - If the operator names a compound procedure, evaluate the body of the compound procedure with the arguments substituted for the formal parameters in the body. Count internal definitions as part of the body. – watch out for substitution bugs here.

Here's an example evaluation:

```
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
(fact 3)
([ [n] (if (= n 1) 1 (* n (fact (- n 1))))] 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 ([ [n] (if (= n 1) 1 (* n (fact (- n 1))))] 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #f 1 (* 2 (fact (- 2 1)))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 ([ [n] (if (= n 1) 1 (* n (fact (- n 1))))] 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 (if #t 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 1))
(* 3 2)
6
```

There are other possible models, e.g. here are the changes for normal order evaluation.

- Combination – (<operator> <operand1> ... <operandn>):
  - Evaluate the operator expression to get the procedure.
  - If the operator names a primitive procedure, evaluate the operands and do whatever magic the primitive procedure does.
  - If the operator names a compound procedure, evaluate the body of the compound procedure with the operands substituted for the formal parameters in the body. Count internal definitions as part of the body. – watch out for substitution bugs here.

Here's an example of how they differ:

Applicative order	Normal Order
`((lambda (x) (+ x x)) (* 3 4))`	`((lambda (x) (+ x x)) (* 3 4))`
`((lambda (x) (+ x x)) 12)`	`(+ (* 3 4) (* 3 4))`
`(+ 12 12)`	`(+ 12 12)`
`24`	`24`

This may matter in some cases:

```
((lambda (x y) (+ x 2)) 3 (/ 1 0))
```

Examples of standard types of procedures:

Exponentiation using multiplication steps:

```
(define exp-1
  (lambda (a b)
    (if (= b 0)
        1
        (* a (exp-1 a (- b 1))))))
```

Second set of exponentiation rules:

```
(define exp-2
   (lambda (a b)
      (define exp-iter
         (lambda (a b ans)
           (if (= b 0)
               ans
               (exp-iter a (- b 1) (* ans a)))))
      (exp-iter a b 1)))
```

Using the Substitution Model on `exp-1` takes [ ] steps. We say that this gives rise to a [ ] Process.

Using the Substitution Model on `exp-2` takes [ ] steps. We say that this gives rise to a [ ] Process.

To capture the differences between these processes, we use the idea of Orders of Growth. Given some resource, such as space or time, the order of growth of a process describes how quickly the demands for that resource increase as we increase the size of the problem.

More formally, if $R(n)$ denotes the amount of resource needed to solve a problem of size n and we can find a constants k_1 and k_2 and a mathematical function $f(n)$ such that

$$k_1 f(n) \leq R(n) \leq k_2 f(n)$$

for large n, then $R(n)$ is said to have order of growth $f(n)$, which we write as $\Theta(f(n))$.

Fill in the table for the orders of growth of our exmaples

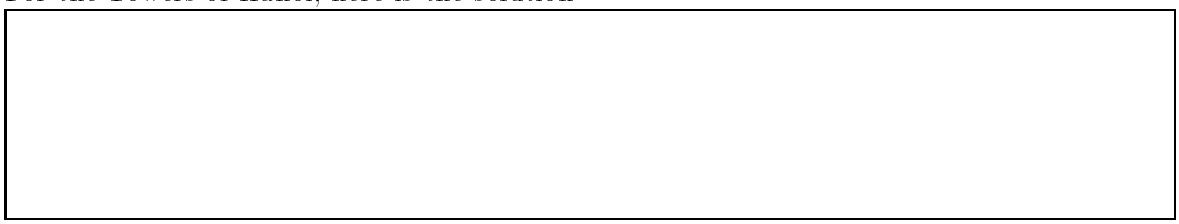
	Space	Time
Times-1		
Times-2		

Example: Towers of Hanoi – tree recursion

To solve this problem, we use the technique of Wishful Thinking, which says:

- decide how to reduce the general problem to a simple one
- pretend we can solve the simpler one
- use the simpler solution to construct a solution to the general problem

For the Towers of Hanoi, here is the solution



```
(define move-tower
  (lambda (size from to extra)
    (cond ((= size 0) true)
          (else (move-tower (1- size) from extra to)
                (print-move from to)
                (move-tower (1- size) extra to from)))))

(define print-move
   (lambda (from to)
     (write-line "Move top disk from ")
     (write-line from)
     (write-line " to ")
     (write-line to)))
```

The orders of growth for `move-tower` are [ ]. This is an example of an [ ] process.