TEL AVIV UNIVERSITY
Department of Computer Science
0368.1105 – Extended Introduction to Computer Science
Fall Semester, 2000/2001

Handout for Lecture 2

Language Components

- Primitives
- Means of combination
  - procedure application
  - compound data structures
- Means of abstraction
  - naming
  - procedures
  - data abstraction

Scheme Basics

Rules for scheme

1. (Almost) Every expression has a value (**returned** when an expression is **evaluated**).
2. Every value has a type.

Rules for evaluation

1. If self-evaluating, return value
2. If a name, return value associated with name in environment
3. If a special form, do something special
4. If a combination, then
   - evaluate all of the subexpressions in combination (any order)
   - apply the operator to the values of the operands (arguments) and return the result

Rules for application

1. If procedure is primitive procedure, just do it
2. If procedure is a compound procedure, then evaluate the body of the procedure with the formal parameters replaced by the actual argument values.

The special form if

```
(if <predicate> <consequent> <alternative>)
```

Evaluate the **predicate**. If it is #f, then evaluate the **alternative** and return its value, otherwise evaluate the **consequent** and return its value.

```
;;; examples
(define sum-sq
  (lambda (n)
    (if (zero? n)
        0
        (+ (sum-sq (- n 1))
           (square n)))))

(define odd?
  (lambda (n)
    (if (zero? n)
        #f
        (not (odd? (- n 1))))))

;;; Mutual recursion - another version of odd?
(define odd?
  (lambda (n)
    (if (zero? n)
        #f
        (even? (- n 1)))))

(define even?
  (lambda (n)
    (if (zero? n)
        #t
        (odd? (- n 1)))))
```

Combining define and lambda

- An alternative version of define for procedures.

  ```
  (define (<name> <parameter-1> ... <parameter-n>) <body> )
  ```

  is equivalent to

  ```
  (define <name> (lambda (<parameter-1> ... <parameter-n>) <body> ))
  ```

- "Syntactic sugar".
- For example:

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

  is equivalent to

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

Square roots by Newton's method

```
(define sqrt-iter
  (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)
                   x))))

(define improve
  (lambda (guess x)
    (average guess (/ x guess))))

(define average
  (lambda (x y)
    (/ (+ x y) 2)))

(define good-enough?
  (lambda (guess x)
    (< (abs (- (square guess) x)) 0.001)))

(define sqrt
  (lambda (x)
    (sqrt-iter 1.0 x)))
```

```
;;; Alternate syntax for defining functions

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))

(define (improve guess x)
  (average guess (/ x guess)))

(define (average x y)
  (/ (+ x y) 2))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (sqrt x)
  (sqrt-iter 1.0 x))

;; Block-structured version
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))

;; Taking advantage of lexical scoping

(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```