An introduction to Scheme
Introduction

• A powerful programming language is more than just a means for instructing a computer to perform tasks.
• The language also serves as a framework within which we organize our ideas about processes.
• Thus, when we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas.
• Every powerful language has three mechanisms for accomplishing this:
  – primitive expressions, which represent the simplest entities with which the language is concerned.
  – means of combination, by which compound expressions are built from simpler ones,
  – means of abstraction, by which compound objects can be named and manipulated as units.

• In programming, we deal with two kinds of objects: procedures and data.

• Thus, any powerful programming language should be able to describe primitive data and primitive procedures and should have methods for combining and abstracting procedures and data.
Expressions

• In Scheme, expressions representing numbers may be combined with an expression representing a primitive procedure (such as + or *), to form a compound expression that represents the application of the procedure to those numbers.

• Examples:

  ==>  (+ 137 349)
       486

  ==>  (- 1000 334)
       666

  ==>  (* 5 99)
       495

  ==>  (/ 10 5)
       2

  ==>  (/ 10 6)
       1.66667

  ==>  (+ 2.7 10)
       12.7
Expressions (Cont.)

• Expressions, such as these, formed by delimiting a list of expressions within parenthesis, are called combinations.
  (operator operand operand ... operand)
• Prefix notation: place operands to the left of operators.
• Disadvantage: prefix notation could be confusing at first.
Expressions (Cont.)

• Advantages:
  – Prefix notation can accommodate procedures that take may take an arbitrary number of arguments
    
    ```scheme```
    `( + 21 35 12 7 )`  
    ```
    75
    ```
  – Combinations can be nested
    
    ```scheme```
    `( + (* 3 5) (- 10 6) )`  
    ```
    19
    ```
  – But... complex expressions are difficult to read:
    
    ```scheme```
    `( + (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6) )`  
    ```
• A critical aspect of a programming language is the means it provides for using names to refer to computational objects.

• In Scheme, the operator for naming things is called define

  ==> (define size 2)

  size

  ...

  ==> size

  2
Naming (Cont.)

```scheme
==> (* 5 size)
  10
==> (define pi 3.14159)
  pi
==> (define radius 10)
  radius
==> (* pi (* radius radius))
  314.159
==> (define circumference (* 2 pi radius))
  circumference
==> circumference
  62.8318
```
Naming (Cont.)

• `define` is the simplest means of abstraction in Scheme.

• It allows us to use simple names to refer to the results of compound operations, such as the circumference computed above.

• Complex programs are constructed by building, step-by-step, computational objects of increasing complexity.

• The interpreter must maintain some sort of memory that keeps track of the name-object pair. This memory is called the `environment`. 
Evaluating Combinations

• The (recursive) algorithm to evaluate combinations is:
  – Evaluate the subexpressions of the combination
  – Apply the procedure that is the value of the leftmost subexpression (the operator) to the argument that are the values of the other subexpressions (the operands).

• Expressions can be seen as trees.
• Values in trees are percolated upward starting from the terminal nodes and then combining at higher and higher levels.
• The leaves of these trees are primitive expressions such as numerals, built-in operators, or other names.
  – The value of numerals are the numbers that they name
  – The value of built in operators are the instruction sequences that carry out the corresponding operations
  – The values of other names are the objects associated with these names in the environment.

• Notice that define is as exception to the general evaluation rule
Compound procedures

- `define` can be used not only to associate names with values, but also to associate names with compound operations.

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

To square something multiply it by itself

- The general form of a procedure definition is

\[
\text{(define ( } \text{name} \text{ <formal parameters> } \text{ ) } \text{ <body> )}
\]
Compound Procedures (Cont.)

```scheme
==> (square 21)
441
==> (square (+ 2 5))
49
==> (define radius 10)
radius
==> (square (square 3))
81
```
• We can use square as the building block for other procedures, for example $x^2 + y^2$ can be expressed as $(+ (\text{square } x) (\text{square } y))$

• And this can be a new procedure

```scheme
==> (define (sum-of-squares x y) )
(+ (square x) (square y))
sum=of-squares
==> (sum-of-squares 3 4)
25
==> (define (f a) (sum-of-squares (+ a 1) (* a 2)))
f
==> (f 5)
136
```
Conditional Expressions and Predicates

• The general form of the conditional expression is
  \( \text{cond} ( \langle p1 \rangle \ \langle e1 \rangle ) \ ( \langle p2 \rangle \ \langle e2 \rangle ) \ldots ( \langle pn \rangle \ \langle en \rangle ) \ ) \)

• The arguments are pairs of expressions \( \langle p \rangle \ \langle e \rangle \) called clauses. The first expressions in each pair is a predicate (an expression whose value is interpreted as true or false). In Scheme, “false” is represented by \texttt{nil}, and any other object is interpreted a “true”.

• The predicate \( \langle p1 \rangle \) is evaluated first. If it is false, then \( \langle p2 \rangle \) is evaluated. If its value is also false then \( \langle p3 \rangle \) is evaluated. This process continues until a predicate is found whose value is true (non-\texttt{nil}) in which case, the interpreter returns the corresponding consequent expression \( \langle e \rangle \). If none of the \( \langle p \rangle \)s is found to be true, the \texttt{cond} returns a value of false.
(define (abs x)
  (cond
    ((> x 0) x)
    ((= x 0) 0)
    (else (- x))))

Same procedure using else:

(define (abs x)
  (cond
    ((< x 0) (- x))
    (else x)))

Same procedure using if:

(define (abs x)
  (if (< x 0)
      (- x)
      x))
• In addition to primitive predicates <, =, and >, there are logical composition operators such as and, or and not.
• For example, the condition that $x$ be in the range $5 < x < 10$:
  
  \[(\text{and} \ (> \ x \ 5) \ (< \ x \ 10))\]

• We can define a predicate to test whether one number is greater than or equal to another as:

\[
\text{(define} \ (>\text{= } x \ y) \\
\text{(or} \ (> \ x \ y) \ (= \ x \ y))\text{)}
\]

or equivalently:

\[
\text{(define} \ (>\text{= } x \ y) \\
\text{(not} \ (< \ x \ y)))\text{)}
\]
Newton’s method solves an equation in one variable \( f(x) = 0 \), by carrying out the iteration

\[
x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}
\]

where \( x_0 \) is a first approximation (guess) to the root.

To compute the square root of a number \( a \), we need to solve the equation \( f(x) = x^2 - a = 0 \)

\[
x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = x_n - \frac{x_n - a / x_n}{2}
\]
(define (sqrt-iter xn a)
  (if (good-enough? xn a)
      xn
      (sqrt-iter (improve xn a) a)))

(define (improve xn a)
  (average xn (/ a xn)))

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

(define (good-enough? xn a)
  (< abs (- (square xn) x) .001))

(define (sqrt a)
  (sqrt-iter 1 a))
It is better to encapsulate auxiliary procedures inside sqrt

(define (sqrt a)
  (define (sqrt-iter xn a)
    (if (good-enough? xn a)
        xn
        (sqrt-iter (improve xn a) a)))

  (define (improve xn a)
    (average xn (/ a xn)))

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

  (define (good-enough? xn a)
    (< abs (- (square xn) x)) .001))

  (sqrt-iter 1 a))
Because of lexical scoping, we don’t need that many reference to a
define (sqrt a)
   (define (sqrt-iter x)
       (if (good-enough? xn)
           xn
           (sqrt-iter (improve xn))))
  (define (improve xn)
       (average xn (/ a xn)))
 (define (average x y)
       (/ (+ x y) 2))
 (define (good-enough? xn a)
       (< abs (- (square xn) a)) .001))

(sqrt-iter 1)
A simple recursive procedure to compute the factorial of a number

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

Can be transformed into iterative form

(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
      product
      (iter (* counter product)(+ counter 1))))

  (iter 1 1)
A simple recursive procedure to compute Fibonacci numbers

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

Can be transformed into iterative form as follows

```
(define (fib n)
  (define (iter a b count)
    (if (= count 0)
      b
      (iter (+ a b) a (- count 1)))
  (iter 1 0 n))
```
• How many different ways can we make change of $1.00, given half-dollars, quarters, dimes, nickels, and pennies?

Number of ways to change amount $a$ using $n$ kinds of coins =
Number of ways to change amount $a$ using all but the first kind of coin
+ Number of ways to change amount $a-d$ using all $n$ kinds of coins, where $d$ is the denomination of the first kind of coin.
(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 (first-denomination kinds-of-coins))
             kinds-of-coins)
            (cc amount (- kinds-of-coins 1)))))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
    (= kinds-of-coins 2) 5)
    (= kinds-of-coins 3) 10)
    (= kinds-of-coins 4) 25)
    (= kinds-of-coins 5) 50)))
The substitution model for procedure application

- Consider
  \[(\text{define (f a) (sum-of-squares (+ a 1) (* a 2))})\]
- To evaluate \((f 5)\) we begin by replacing the formal parameter \(a\) by 5:
  \[(\text{sum-of-squares (+ 5 1) (* 5 2))}\]
- Now, \((+ 5 1)\) produces 6 and \((* 5 2)\) produces 10, so, we must apply the procedure \text{sum-of-squares} to 6 and 10. By applying substitution we get:
  \[(+ (\text{square} 6) (\text{square} 10)).\]
- And then
  \[(+ (* 6 6) (* 10 10))\]
- Which reduces to
  \[(+ 36 100)\]
- And finally to
  \[136\]
• An alternative evaluation model would first expand each procedure definition in terms of simpler and simpler procedures until it obtained an expression involving only primitive operators:

\[(\text{sum-of-squares} \; (+ \; 5 \; 1) \; (* \; 5 \; 2))\]
\[(+ \; (\text{square} \; (+ \; 5 \; 1)) \; (\text{square} \; (* \; 5 \; 2)))\]
\[(+ \; (* \; (+ \; 5 \; 1) \; (+ \; 5 \; 1)) \; (* \; (* \; 5 \; 2) \; (* \; 5 \; 2)))\]

• And then reduce:

\[(+ \; (* \; 6 \; 6) \; (* \; 10 \; 10))\]
\[(+ \; 36 \; 100)\]
\[136\]

• This alternative “fully expand and then reduce” evaluation method is known as *normal-order evaluation*.
  – Occurs in macros and call by name parameter passing

• The “evaluate the arguments and then apply” method that the interpreter actually uses is called *application-order evaluation*. 
• Consider the following three procedures:

(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 (* a (+ a 2))) (pi-sum (+ a 4) b))))
• We could generate each of the procedures by by filling in the slots in the template:

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

To achieve this goal, we define:

```
(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a) (sum term (next a) next b))))
```
(define (sum-cubes a b)
  (sum cube a 1+ b))

(define (pi-sum a b)
  (define (pi-term x)
    (/ 1 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))

(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2)) add-dx b) dx))

\[
\int_a^b f = \left[ f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) \right] dx
\]
(define (pi-sum a b)
  (sum (lambda (x) (/ 1 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))
Using let to define local variables

(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x (square a))
      (* y b)
      (* a b)))

(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
           (- 1 y)))

(define (f x y)
  ((lambda (a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
• The general form of let is:

\[
\text{(let (\(<\text{var1}> \ <\text{expr1}>\))}
\]

\[
\quad (\text{\(<\text{var2}> \ <\text{expr2}>\))
\]

\[
\quad \quad \ldots
\]

\[
\quad (\text{\(<\text{varn}> \ <\text{exprn}>\))
\]

\[
\quad \text{\(<\text{body}>\))}
\]

• This is a shorthand notation for

\[
\text{(\(\lambda (\(<\text{var1}> \ <\text{var2}> \ldots \ <\text{varn}>\))}
\]

\[
\quad (\text{\(<\text{body}>\))}
\]

\[
\quad \text{\(<\text{expr1}>\)}
\]

\[
\quad \text{\(<\text{expr2}>\)}
\]

\[
\quad \quad \ldots
\]

\[
\quad \text{\(<\text{exprn}>\))}
\]
• An important difference between let and define is that in a let expression the variables are bound simultaneously.
• For example, in an environment where x is bound to 2, the expression

\[
((x \ 3) \ (y \ (+ \ x \ 2))) \ (* \ x \ y))
\]

will have the value 12, but the sequence

\[
(define \ x \ 3) \\
(define \ y \ (+ \ x \ 2)) \\
(* \ x \ y)
\]

will result in 15.
Procedures as returned values

- To define a procedure that implements Newton’s method, we need a way to evaluate the derivative.
- This can be done numerically with the following procedure:
  ```scheme
  (lambda (x)
    (/ (- (f (+ x dx)) (f x))
       dx))
  )
  
  - We can express the derivative itself as a procedure:
  ```scheme
  (define (deriv f dx)
    (lambda (x)
      (/ (- (f (+ x dx)) (f x))
         dx)))
  )
  ```

  ```scheme
  ==> ((deriv cube .001) 5)
  75.015
  ```
• We can use deriv to implement Newton’s method:

(define (newton f guess)
    (if (good-enough? guess f)
        guess
        (newton f (improve guess f))))

(define (improve guess f)
    (- guess (/ (f guess)
        ((deriv f .001) guess)))))

(define good-enough? guess f) (< (abs (f guess)) .001))

==> (newton (lambda(x) (-x (cos x))) 1)
Infinite or lazy data structures

(define naturals
  (define (next n) (cons n (delay (next (+ n 1))))))
  (next 1))

(define (tail stream) (force (cdr stream)))

(car naturals) -> 1
(car (tail naturals)) -> 2
(car (tail (tail naturals))) -> 3

See section 6.6.2 in the textbook.