# Introduction to Scheme

1st Semester, A.Y. 2020-2021

**Scheme** is a **functional programming language** developed by MIT in the mid-1970's. It was inspired by the LISP Programming Language invented by *Guy Lewis Steele Jr* and *Gerald Jay Sussman*.

# Features

- Uses an **interactive shell** (Read-Evaluate-Print (REP) Loop) where user can load and run scheme program files or type commands.
- **No declaration is necessary** and all descriptor processing is done during execution. However, this makes Scheme *poorly suited for compilation*.
- Every operation is a function, and functions are defined entirely as expressions.
- Uses linked lists as basic data structure.
- All objects are allocated on the **heap**, referenced via pointers.
- **Relies purely on recursion** rather than iteration as control structure.
- **Arguments are evaluated recursively from left to right**. If functions are nested, innermost arguments are evaluated first.

# Basics

## A. Installation

To install the Scheme interactive shell on your Ubuntu, open the terminal and execute the following command:

```
sudo apt-get install guile-2.2
```

To run the interactive shell in Ubuntu, type `guile` in your terminal.
If you are using Windows, refer to this link: `http://s48.org/1.9/windows.html`.

## B. Syntax

Scheme follows the **prefix expression notation**. That is, the operator always preceeds the arguments. Scheme also encloses each operation in parentheses.

```
guile> (+ 1 2 3)          ;same as 1+2+3
guile> (/ 10 (+ 1 1))     ;same as 10/(1+1)
```

## C. Variables

Variable can be named using alphanumeric characters and special characters such as `! $ % & * + - . / : <`.

The `define` operation is used to create a variable:

```
guile> (define myvar 5)  ;creates a new variable
guile> myvar             ;prints the value of myvar
$1 = 5
guile> (set! myvar 1)    ;reassigns a value to myvar
guile> myvar
$2 = 1
guile> (+ $1 $2)         ;uses the implicit variables
$3 = 5
```

Remember, however, that the `set!` operation can only be used on already defined variables.

Also, whenever an output is given to the user, it is stored in an **implicit variable** whose name starts with a dollar sign, `$`. These variables can be used later on, as in the last expression in the example above.

## D. Comments

Comments begin with a semicolon (`;`).

## E. Loading Scheme Programs

Scheme programs have the file extension `.scm`. The `load` operation is used to import scheme files.

```
guile> (load "filename.scm")
```

# Data Types and Operations

## A. Primitives

### 1 Boolean

The value `true` is denoted by `#t` while `false` is `#f`.

### 2 Numbers

Scheme supports several types of numbers:

```
guile> 2       ;integer
guile> 2+i     ;complex number
guile> 5.6     ;real number
guile> 11/2    ;real number
guile> 1/3     ;rational number
```

Below are some operations involving numbers:

```
guile> (expt 2 3)       ;exponentiation (2^3=8)
guile> (max 1 2 3 2 1)  ;maximum (3)
guile> (min 3 2 1 2 3)  ;minimum (1)
guile> (abs -6)         ;absolute value (6)
guile> (modulo 5 3)     ;remainder (2)
```

### 3 Characters

Characters in Scheme are preceeded by `#\`. For example, `#\A`, `#\B`, and `#\C`. The space character is denoted by `#\space`, the tab character is `#\tab`, and the newline character is `#\newline`.

#### Case conversion

```
guile> (char-downcase #\A)
$4 = #\a
guile> (char-upcase #\a)
$5 = #\A
```

#### Character comparison

```
guile> (char<? #\a #\b)   ; #t
guile> (char>? #\a #\b)   ; #f
guile> (char<=? #\a #\b)  ; #t
guile> (char>=? #\a #\b)  ; #f
guile> (char=? #\a #\b)   ; #f
```

These comparison operators compares the ASCII values associated to each character.
For **case-insensitive** comparison, replace `char` with `char-ci`.

### 4 Symbols

Symbols are used as **identifiers for variables**. Unlike booleans, numbers, and characters (which are self-evaluating), a symbol evaluates to the value that variable holds.

To prevent Scheme from evaluating a symbol as a variable, use `quote`:

```
guile> temp              ;this will produce an error
  <insert ERROR here>    ;if a temp variable is not defined
guile> (quote temp)
$6 = temp
guile> 'temp
$7 = temp
```

## 5 Strings

Strings are written as **sequences of characters enclosed within doublequotes**.

Below are several ways to create and manipulate the contents of a string:

```
guile> "hello"                          ;a string
guile> (string #\h #\i #\!)             ;creating a string from characters
guile> (define s1 "bye")                ;assigning to a variable
guile> (define s2 (string #\h #\i #\!))
guile> (string-append "good" s1 s1)     ;string concatenation
guile> (string-ref s1 1)                ;same as s1[1]
guile> (string-set! s2 1 #\a)           ;same as s2[1]='a'
```

**String Operations**

```
guile> (string=? "hello" "hi")        ;string comparisons
guile> (string<? "Hello" "hello")
guile> (string>? "hello" "hi")
guile> (string<=? "Hello" "hello")
guile> (string>=? "hello" "hi")
guile> (string-ci=? "hello" "hi")     ;case-insensitive comparisons
guile> (string-ci>=? "hello" "hi")
guile> (string-ci<=? "hello" "hi")
guile> (string-ci>? "hello" "hi")
guile> (string-ci<? "hello" "hi")
guile> (string-length s1)             ;gets the length of s1
```

# B. Vectors

**Vectors are heterogenous structures** whose elements are indexed by integers (zero-indexing).

Below are ways to handle vectors:

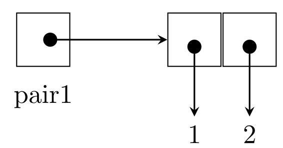
```
guile> (define vect1 #(#\a 'a "app" '(x y z)))        ;create vector using #
guile> (define vect2 (vector #\b 'b "bad" '(a b c))) ;using vector keyword
guile> (vector-length vect1)            ;get number of elements
guile> (vector-ref vect2 0)             ;same as vect2[0]
guile> (vector-set! vect2 1 "boss")     ;same as vect2[1] = "boss"
```

# C. Pairs

A pair, or a *dotted pair*, is a **record structure with two fields** called the `car` and the `cdr` fields. These fields can be accessed using the functions with the same name.

To create a pair, you need to use the cons function:

```
guile> (define pair1 (cons 1 2))     ; will return (1 . 2)
```



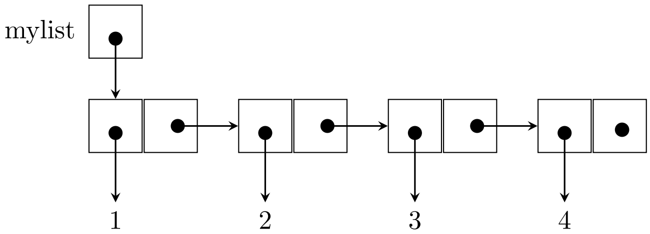
```
guile> (define pair2 (cons 'a 'b)) ;creates a new pair
guile> (car pair2)                 ;returns a
guile> (cdr pair2)                 ;returns b
guile> (define pair3 (cons 'a '())
guile> (cdr pair3)                 ;returns ()
guile> (cons 'a (cons 'b 'c))      ;returns ('a 'b . 'c)
guile> (set-car! pair2 2)          ;changes the car
guile> (set-cdr! pair2 5)          ;changes the cdr
```

## D. Lists

A list is a list of pairs, whose `cdr` fields hold pointers that string them together, and whose `car` fields hold the values.

The `list` function is used to create a list:

```
guile> (define mylist (list 1 2 3 4))     ;will return (1 2 3 4)
```




```
guile> (car mylist)
guile> (cdr mylist)
guile> (car (cdr (cdr mylist)))  ;returns 3
guile> (caddr mylist)            ;shorthand for the command above
guile> (list 'a 'b 'c)
guile> (cons 'a (cons 'b (cons 'c '())))
guile> (list-ref mylist 0)     ;will return 1
guile> (list-tail mylist 1)    ;will return (2 3 4)
guile> (list-tail mylist 2)    ;will return (3 4)
guile> (length mylist)         ;returns the length of the list x
guile> (reverse mylist)        ;returns a reversed version of the list
guile> (append mylist (reverse mylist)) ;does not modify the original list
guile> '(+ 1 2 3)              ;creates a list containing the arguments
```

# Other Operations

## A. Type Checking

```
guile> (boolean? #t)
guile> (number? 42)
guile> (complex? 2+i)
guile> (real? 32.3)
guile> (rational? 22/7)
guile> (integer? 3)
guile> (char? #\a)
guile> (symbol? 'ask)
guile> (string? "bye")
guile> (vector? x)
guile> (pair? pair1)
guile> (list? '(1 2 3))
```

## B. Type Conversion

```
guile> (char->integer #\A)     ; returns 65
guile> (integer->char 66)      ; returns B
guile> (string->number "100") ; returns 100
guile> (number->string 99)     ; returns "99"
guile> (string->list "soup")  ;returns (#\s #\o #\u #\p)
guile> (list->string '(#\c #\o #\r #\e))  ; returns "core"
guile> (string->symbol "sym1"); returns sym1
guile> (symbol->string ack)    ; returns "ack"
guile> (list->vector '(a b c)); returns #(a b c)
guile> (vector->list #(a b))  ; returns (a b)
```

## C. Relational Operators

```
guile> (< 1 2)      ;returns #t or #f
guile> (= 42 42)
guile> (> 5 6.2)
guile> (>= 34 30)
guile> (<= 34 30)
```

## D. Logical Operators

Both and and or operations follow short-circuit evaluation.

```
guile> (not #t)                          ; returns #f
guile> (not #f)                          ; returns #t
guile> (and (+ 1 2)(= 1 2)(> 3 4)) ; returns #f
guile> (and (+ 1 2)(+ 0 0))           ; returns 0
guile> (or #f (- 1 3) #f)              ; returns -2
guile> (or (> 1 5)(< 3 2))            ; returns #t
```

## E. Equality Predicates

```
guile> (equal? list1 list2) ;deep element by element structural comparison
guile> (define x '(1 2))
guile> (define y '(1 2))
guile> (eq? x y)    ;will return #f since
guile> (define y x) ;eq checks if two values are referring to the same object
guile> (eq? x y)    ;will now return #t since y is now defined as x
```

# 1 Control Constructs

## A. If-Expression

```
;if only
(if (condition)                Example:
    (action for true)          (if (= x 5) (+ x 2))
)

;if-else
(if (condition)                (if (= x 5)
   (action for true)              (+ x 2)
   (action for false)             (set! x 5)
)                              )

;if-elseif-else can be achieved by nesting
```

## B. If-ElseIf-Else using cond

```
(cond                                  Example:
    (test 1                            (cond
        (action 1))                        ((< x 5) "less than")
    (test 2                                ((> x 5) "greater than")
        (action 2))                        (else "equal")
        ...                            )
    (else
        (action N))
)
```

## C. Functions

```
;syntax
(define (function-name param1 param2 ... paramN)
     (function body)
)

;EXAMPLES
(define (square n)            ;this can be called
     (* n n)                  ;using (square N)
)                             ;where N is any number


(define (firstInDictionary str1 str2)
     (if (string-ci<? str1 str2)
          str1
          str2
     )
)
```