Scheme is a **dynamically-typed language**

- Types only become apparent at run time.
- Scheme associates types with values (or objects) rather than with variables.
- A variable may name a location where a value can be stored.
- A variable that does so is said to be **bound** to the location.

An **environment** is a set of **variable bindings**

- The environment in which your programs execute is a child (extension) of the environment containing the **system's bindings**; system names are visible to your programs.
- The environment in effect at some point in a program is called the current environment.
- Every REP loop has a current environment.
- A top-level definition may add a binding to an existing environment.
- The system doesn't run out of memory because the **garbage collector** reclaims the storage occupied by an object when the object cannot possibly be needed by a future computation.

Scheme is **statically-scoped**

- Each use of a variable is associated with a lexically apparent **binding** of that variable.
- Scope rules define the visibility rules for names in a programming language.
- Most languages are *statically-scoped*. A *block* defines a new scope. Variables can be declared in that scope, and aren't visible from the outside.

Scheme **procedures are objects**

- you can create them dynamically, store them in data structures, return them as the results of other procedures, and so on.

Lexical Conventions

Whitespace characters are spaces, newlines, and tabs.
Whitespace is used for readability and to separate tokens (indivisible lexical units).

All whitespace characters are **delimiters**. Also, the following characters act as delimiters:

```
(   )   ;   "   '   `   |
```

The beginning of a comment is indicated with a semicolon, `;`.

An **identifier** is a sequence of one or more non-delimiter characters. Scheme is **case-insensitive**.

- Certain identifiers are reserved for use as **keywords**; they should not be used as variables:

```
access                  define-syntax           macro
and                     delay                   make-environment
begin                   do                      named-lambda
bkpt                    fluid-let               or
case                    if                      quasiquote
cond                    in-package              quote
cons-stream             lambda                  scode-quote
declare                 let                     sequence
default-object?         let*                    set!
define                  let-syntax              the-environment . . .
```

- Any identifier that is not a keyword can be used as a variable.

A Scheme **expression** is a construct that returns a value.

An expression may be a ***literal***, a ***procedure call***, or a ***special form***.

Literal constants may be written as:

```
"abc"                                   =>  "abc"
145932                                  =>  145932
#t                                      =>  #t
```

Numeric, string, character, and boolean constants evaluate to the constants themselves.
Symbols, pairs, lists, and vectors require quoting.
The single quote indicates literal data; it suppresses evaluation:

```
'a                                      =>  a
‘(1 2 3)                                =>  (1 2 3)
```

Procedure Calls

are written by enclosing in parentheses the procedure to be called (the **operator**) and the arguments to be passed to it (the **operands**). The operator and operand expressions are *evaluated* and the resulting procedure is passed the resulting arguments.

```
(operator operand ...)
(+ 3 4)                                 =>  7
```

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables `+` and `*`.
New procedures are created by evaluating `lambda` expressions.

If the *operator* is a keyword, then the expression is a special form.

Special forms

A parenthesized expression that starts with a **keyword** is a **special form**.

```
(keyword component ...)
```

top-level variable definitions

`(define variable expression)`

`define` binds *`variable`* to a new location, in the *current environment*, and performs the assignment:

```
(define x 28)
x                                       =>  28

(define first car)
(first '(1 2))                          =>  1
```

assignments

`set!` *`variable [expression]`*

evaluates `expression` and stores the value in the location to which *`variable`* is bound.

```
(define x 2)
(+ x 1)                                 =>  3
(set! x 4)
(+ x 1)                                 =>  5
```

Local Bindings

`(let ((var val) ...)` exp_1 exp_2 `...)`

`let` establishes local variable bindings. Each variable `var` is bound to the value of the corresponding expression `val`. The body of the `let`, in which the variables are bound, is the sequence of expressions exp_1 exp_2 `...`

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))
```

```
(let ((x 'a) (y '(b c)))
  (cons x y))                    =>  (a b c)
```

The two main ways of declaring a variable are using define:

```
> (define x 2)
> (* x 3)
6
> x
2
```

the variable x continues to exist, it is defined for all to see, use and modify

and let, to establish local variable bindings:

```
> (let ((x 2) (y 4)) (* x y))
8
> x
;ERROR: "/usr/lib/scm/Iedline.scm": unbound variable:  x
```

once the block of code has finished executing, the variables x and y are no longer accessible

Variables in Scheme are dynamically typed, so there is no need to declare the type of the variable when you define it. You can change a variable to any value - the same variable can hold any type of data.

```
(define f
  (lambda(x)
    (cond
      ((= x 3) 5)
      (else "Error: Invalid input!"))))

(define y (f 5))
```

Conditionals

if *predicate consequent [alternative]*

```
(if (> 3 2) 'yes 'no)                    =>  yes
(if (> 2 3) 'yes 'no)                    =>  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                             =>  1
```

cond clause clause ...

Each *clause* has this form:
(predicate *expression* ...)

The last *clause* may be an **`else` clause**, which has the form:
(else *expression expression* ...)

A `cond` expression does the following:

1. Evaluates the *predicate* expressions of successive *clause*s until one evaluates to a true value.
2. When a *predicate* evaluates to a true value, `cond` evaluates the *expression*s in the associated *clause* in left to right order, and returns the result of evaluating the last *expression* in the *clause* as the result.

```
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))                     =>  equal
```

Sequencing

You can write an expression that is an ordered sequence of other expressions, using `begin`.

`begin` *`expression expression ...`*

The *expression*s are evaluated sequentially from left to right, and the value of the last *expression* is returned. This expression type is used to sequence side effects such as input and output.

```
(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))

(begin (one)
       (two))
```

- In terms of control flow, a (begin ...) expression is like a { ... } block in C.
- `begin` expressions aren't just code blocks, though, because they are expressions that return a value.
- A begin returns the value of the last expression in the sequence. For example, the begin expression above returns the value returned by the call to `two`.
- The bodies of procedures work like `begin`s as well. If the body contains several expressions, they are evaluated in order, and the last value is returned as the value of the procedure call.

lambdas

The **`lambda`** syntactic form is used to create procedures.
A `lambda` expression evaluates to a procedure.

`lambda` ***`formals expression expression ...`***

`formals`, the formal parameter list, is referred to as a **lambda list**.

```
(lambda (x) (+ x 3))            => #<procedure>
((lambda (x) (+ x 3)) 7)        => 10
```

With a top-level definition and a procedure call:

```
(define add3
   (lambda (x) (+ x 3)))                  =>  unspecified

(add3 3)                                  =>  6
```

- The environment in effect when the `lambda` expression is evaluated is remembered as part of the procedure; it is called the **closing environment**.
- When the procedure is later called with some arguments, the closing environment is extended by binding the variables in the formal parameter list to fresh locations, and the locations are filled with the arguments.
- The new environment created by this process is referred to as the **invocation environment**.
- Once the invocation environment has been constructed, the *expression*s in the body of the `lambda` expression are evaluated sequentially in it.
- The result of evaluating the last *expression* in the body is returned as the result of the procedure call.

Example of Interacting with a User

```
(define greet
  (lambda (name)
    (display (string-append "Hello " name "!"))
    (display " What is your favorite number?")
    (let ((num (read)))
      (if (equal? num 5)
          (begin
            (display "Great! ")
            (display num)
            (display " is my favorite number too."))
          (display (string-append  (number->string num) " is ok."))))))

(greet "Tom")
```

Scheme provides a procedure called read that reads one value of any type

begin when there's more than one

convert num to a string

Numeric Procedures

Procedure	Meaning
`(+` x_1 `...` x_n`)`	The sum of $x_1, \ldots, x_n$
`(*` x_1 `...` x_n`)`	The product of $x_1, \ldots, x_n$
`(-` x y`)`	Subtract y from x
`(-` x`)`	Minus x
`(/` x y`)`	Divide x by y
`(max` x_1 `...` x_n`)`	The maximum of $x_1, \ldots, x_n$
`(min` x_1 `...` x_n`)`	The minimum of $x_1, \ldots, x_n$
`(sqrt` x`)`	The square root of x
`(abs` x`)`	The absolute value of x

More Conditional Forms

Forms	Meaning
`(or` x_1 `...` x_n`)`	Logical or
`(and` x_1 `...` x_n`)`	Logical and
`(not` x`)`	Logical negation

A **predicate** is a procedure that always returns a boolean value (#t or #f). By convention, predicates usually have names that end in `?'.

A **mutation procedure** is a procedure that alters a data structure. By convention, mutation procedures usually have names that end in `!'.

Numeric Predicates

Predicate	Meaning
`(zero?` x`)`	x is zero
`(positive?` x`)`	x is positive
`(negative?` x`)`	x is negative
`(even?` x`)`	x is even
`(odd?` x`)`	x is odd

Relational Predicates

Predicate	Meaning
`(=` x y`)`	x is equal to y
`(<` x y`)`	x is less than y
`(<` x y`)`	x is greater than y
`(<=` x y`)`	x is no greater than y
`(>=` x y`)`	x is no less than y

Type Predicates

Predicate	Meaning
`(number?` x`)`	x is a number
`(boolean?` x`)`	x is a Boolean value (i.e., `#t`, `#f`)

Lists

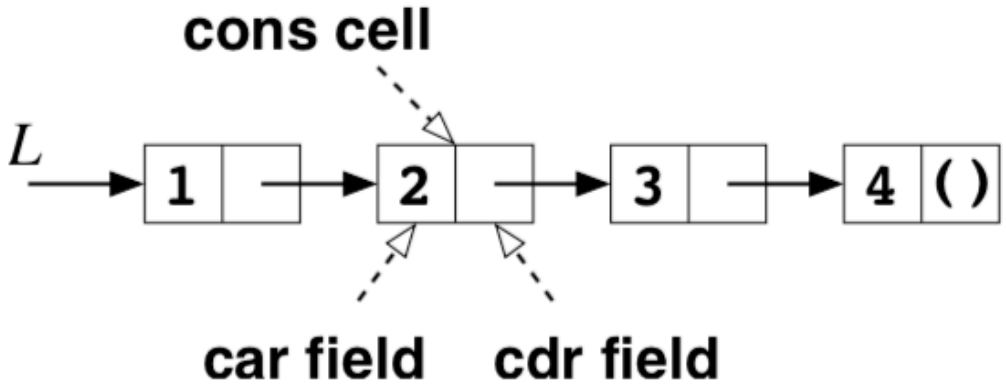
- A **pair** (sometimes called a dotted pair) is a data structure with two fields called the **car** and **cdr** fields (for historical reasons).

Dotted Pairs: (car . cdr)

- Pairs are created by the procedure **`cons`**. The car and cdr fields are accessed by the procedures **`car`** and **`cdr`**.

- Pairs are used primarily to represent **lists**.
- A list can be defined recursively as either the empty list or a pair whose cdr is a list.

```
(define L '(1 2 3 4))
```



Lists

- The objects in the car fields of successive pairs of a list are the **elements** of the list.
- The **length** of a list is the number of elements, which is the same as the number of pairs.
- The **empty list** is a special object of its own type (it is not a pair); it has no elements and its length is zero.

The most general notation for Scheme pairs is the "dotted" notation `(`*`c1`*` . `*`c2`*`)` where *c1* is the value of the car field and *c2* is the value of the cdr field.
For example, `(4 . 5)` is a pair whose car is `4` and whose cdr is `5`.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `()`.

For example, the following are equivalent notations for a list of symbols:

```
(a b c d e)
(a . (b . (c . (d . (e . ()))))
```

Lists

cons *object object ...*

`cons` conses together the last two arguments rather than consing the last argument with the empty list. If the last argument is not a list the result is an improper list. If the last argument is a list, the result is a list consisting of the initial arguments and all of the items in the final argument. If there is only one argument, the result is the argument.

```
(cons 'a 'b 'c)                    =>  (a b . c)
(cons 'a 'b '(c d))                =>  (a b c d)
(cons 'a)                          =>  a
```

list *object ...*

Returns a list of its arguments.

```
(list 'a (+ 3 4) 'c)                =>  (a 7 c)
(list)                              =>  ()
```

car *pair*
Returns the contents of the car field of *pair*. Note that it is an error to take the `car` of the empty list.

```
(car '(a b c))                          =>  a
(car '((a) b c d))                      =>  (a)
(car '(1 . 2))                          =>  1
(car '())                               error--> Illegal datum
```

cdr *pair*
Returns the contents of the cdr field of *pair*. Note that it is an error to take the `cdr` of the empty list.

```
(cdr '((a) b c d))                      =>  (b c d)
(cdr '(1 . 2))                          =>  2
(cdr '())                               error--> Illegal datum
```

set-car! *pair object*
Stores *object* in the car field of *pair*. The value returned by `set-car!` is unspecified.

```
> (define x '(a b c d))
> (set-car! x 'z)
> x
(z b c d)
```

set-cdr! *pair object*
Stores *object* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

```
> (set-cdr! x '(y x))
> x
(z y x)
```

Procedure	Meaning
(list x_1 x_2 **...** x_n**)**	create a list containing the arguments
(list? x**)**	test if x is a list
(null? x**)**	test if x is the empty list
(pair? x**)**	test if x is a **cons** cell
(member x L**)**	test if x is a member of list L
(length L**)**	the number of members of list L

Recursion with Lists

```
(define list-sum
  (lambda (L)
    (if (null? L)
      0
      (+ (car L) (list-sum (cdr L))))))
```

```
(list-sum '(2 -1 4))

= (+ 2 (+ -1 (+ 4 0)))
= 5
```

A10

```
(#%require (lib "27.ss" "srfi"))
(random-source-randomize! default-random-source)

(define play_guess
  (lambda (answer num-of-guesses)
    ;(display answer)
    (display "Enter guess ")
    (let ((guess (read)))
      (if (equal? answer guess)
          (begin
            (display "Correct! ")
            (display num-of-guesses)
            (display " guesses... ")
            (display  "Please enter your name:" )
            (let ((name (read)))
              (display (string-append "Good game, " name))
              (newline)))
          (begin
            (if (< guess answer)
                (display "Higher...")
                (display "Lower..."))
            (play_guess answer (+ num-of-guesses 1) ))))))

; Generate random number and play the game
(display "Guess a number from 1 to 100: \n")
(play_guess (+ (random-integer 99) 1) 1)
```