A language should have the ability to work with arbitrary symbols as data rather than just being able to work with numbers or list of numbers.

Suppose we want to store a procedure or a multi-variate equation as data?
Suppose we want to create a list containing two elements a and b.
One cannot simply use the expression (list *a b*) to create (*a b*).
The interpreter would assume that *a* and *b* are symbols try to find if *a* and *b* have been defined and try to create a list with two values.

A language should have the ability to work with arbitrary symbols as data rather than just being able to work with numbers or list of numbers.

Suppose we want to store a procedure or a multi-variate equation as data?
Suppose we want to create a list containing two elements a and b.
One cannot simply use the expression (list *a b*) to create (*a b*).
The interpreter would assume that *a* and *b* are symbols try to find if *a* and *b* have been defined and try to create a list with two values.

In order to manipulate symbols we need a new element in our language: the ability to **quote** a data object.

The common practice in natural languages is to use quotation marks to indicate that a word or a sentence is to be treated literally as a string of characters.

Consider the following sentence.

- The magic words for the door are "My Dog's name".

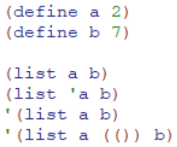
The preceding sentence do not violate any grammatical rules, however its meaning is unclear without the usage of quotes. Without the quotes, one might replace that portion with his/her dog's name. The quotes here imply that the words are not interchangeable with anything else.

We can use quotes to identify lists and symbols that are to be treated as data objects rather than as expressions to be evaluated.

We place a quotation mark (') only at the beginning of the object to be quoted.

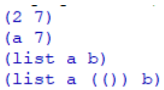
We can get away with this in Scheme syntax because we rely on blanks and parentheses to delimit objects.

Thus, the meaning of the single quote ' character is to *quote* the next object.



```
(define a 2)
(define b 7)

(list a b)
(list 'a b)
'(list a b)
'(list a (()) b)
```

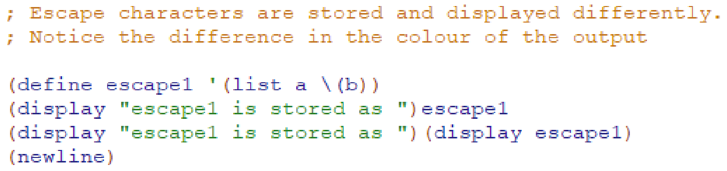
The corresponding output would be as follows



```
(2 7)
(a 7)
(list a b)
(list a (()) b)
```

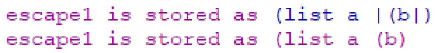
One might guess that the expression
'(list a (b) would produce (list a (b) as the output.
but the object following the quote has not be properly defined as a list.

So, trying to execute this expression by pressing enter would cause you to go to the next line, as the interpreter is assuming that some other portion of the expression is yet to be typed.



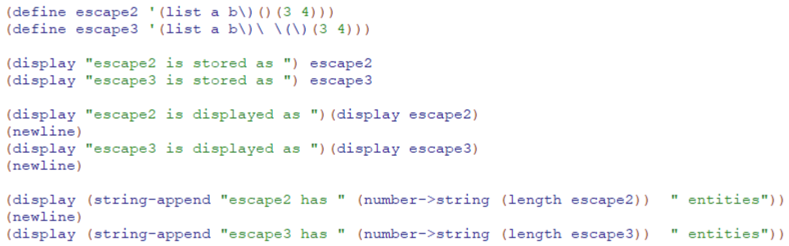
```
; Escape characters are stored and displayed differently.
; Notice the difference in the colour of the output

(define escape1 '(list a \(b))
(display "escape1 is stored as ")escape1
(display "escape1 is stored as ")(display escape1)
(newline)
```

This shows that internal representation of certain entities might be different from how they are displayed. Back-slash is used to introduce open and closed parenthesis as a part of an entity inside a list.



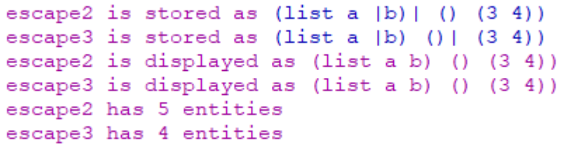
```
escape1 is stored as (list a |(b|)
escape1 is stored as (list a (b)
```

Also notice that the \ preceeds the (in the definition of escape1, but when escape1's storage is looked at, it stores a | before and after the respective entity of the escape1. This is to simplify storage and say that this entity has one or more escaped characters and it needs to escape all such characters.



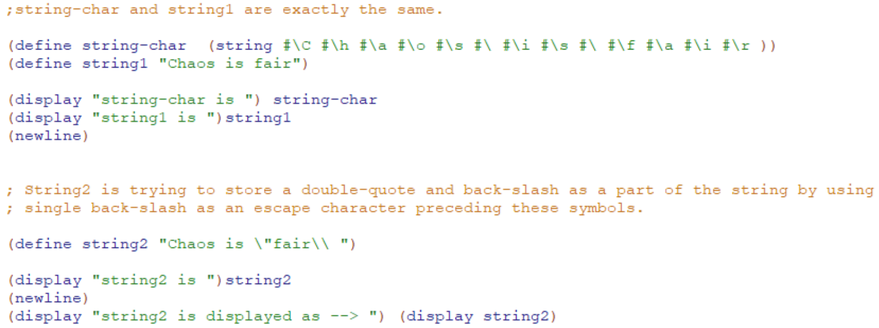
```
(define escape2 '(list a b\)()(3 4)))
(define escape3 '(list a b\)\ \(\)(3 4)))

(display "escape2 is stored as ") escape2
(display "escape3 is stored as ") escape3

(display "escape2 is displayed as ")(display escape2)
(newline)
(display "escape3 is displayed as ")(display escape3)
(newline)

(display (string-append "escape2 has " (number->string (length escape2))  " entities"))
(newline)
(display (string-append "escape3 has " (number->string (length escape3))  " entities"))
```

The observations have been mentioned in the next slide



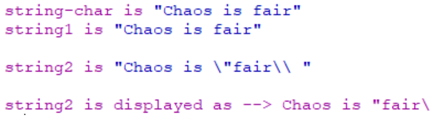
```
escape2 is stored as (list a |b)| () (3 4))
escape3 is stored as (list a |b) ()| (3 4))
escape2 is displayed as (list a b) () (3 4))
escape3 is displayed as (list a b) () (3 4))
escape2 has 5 entities
escape3 has 4 entities
```

- The definition of escape2 and escape3 (as well as their storage) make it evident that they are two different quoted-lists.
- However escape2 and escape3 are displayed exactly the same.
- If one tries to check the number of entities in escape2 and escape3, one realises the clear difference between the two.
- Scheme has several delimiters like
  ( ) ; " |
  ' (i.e. single quote) ‘ (i.e. backtick symbol above tab button) along with whitespace. That is why escape2 has 5 entities i.e "list", "a", "b)", "()" and "(3 4)", eventhough there is no whitespace between *b* and the last closing parenthesis.
- Scheme simplifies the storage and uses | before and after every entity which has an escaped character. This is evident from the third entity in escape3 which has several escaped characters.

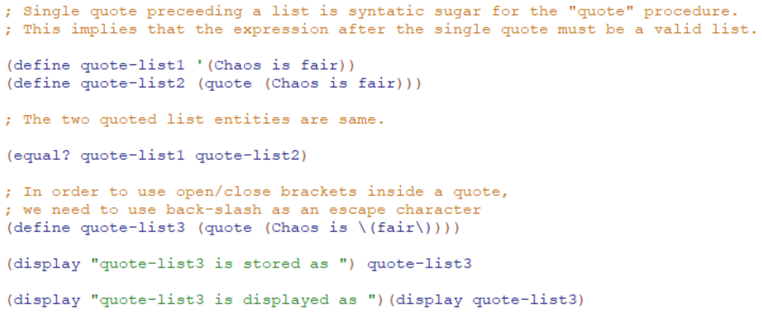


```
;string-char and string1 are exactly the same.

(define string-char  (string #\C #\h #\a #\o #\s #\  #\i #\s #\  #\f #\a #\i #\r ))
(define string1 "Chaos is fair")

(display "string-char is ") string-char
(display "string1 is ")string1
(newline)


; String2 is trying to store a double-quote and back-slash as a part of the string by using
; single back-slash as an escape character preceding these symbols.

(define string2 "Chaos is \"fair\\ ")

(display "string2 is ")string2
(newline)
(display "string2 is displayed as --> ") (display string2)
```

The output would be

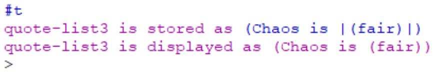


```
string-char is "Chaos is fair"
string1 is "Chaos is fair"

string2 is "Chaos is \"fair\\ "

string2 is displayed as --> Chaos is "fair\
```



```
; Single quote preceeding a list is syntatic sugar for the "quote" procedure.
; This implies that the expression after the single quote must be a valid list.

(define quote-list1 '(Chaos is fair))
(define quote-list2 (quote (Chaos is fair)))

; The two quoted list entities are same.

(equal? quote-list1 quote-list2)

; In order to use open/close brackets inside a quote,
; we need to use back-slash as an escape character
(define quote-list3 (quote (Chaos is \(fair\))))

(display "quote-list3 is stored as ") quote-list3

(display "quote-list3 is displayed as ")(display quote-list3)
```

The output would be



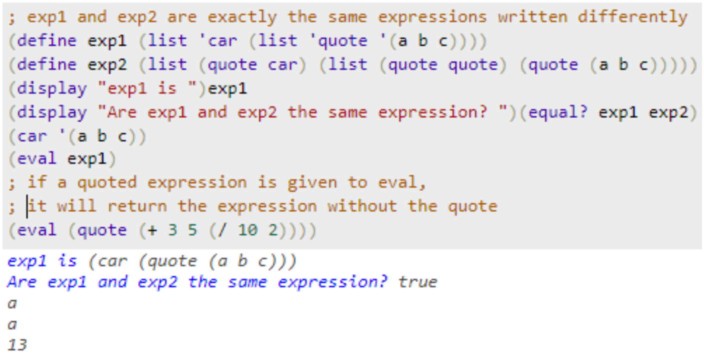
```
#t
quote-list3 is stored as (Chaos is |(fair)|)
quote-list3 is displayed as (Chaos is (fair))
>
```

The use of the quotation mark violates the general rule that all compound expressions in our language should be delimited by parentheses and look like lists.

We can recover this consistency by introducing a special form quote, which serves the same purpose as the quotation mark.

This is important because it maintains the principle that any expression seen by the interpreter can be manipulated as a data object.

For instance, we could construct the expression (car '(a b c)), which is the same as (car (quote (a b c))), by evaluating (list 'car (list 'quote '(a b c)))



```
; exp1 and exp2 are exactly the same expressions written differently
(define exp1 (list 'car (list 'quote '(a b c))))
(define exp2 (list (quote car) (list (quote quote) (quote (a b c)))))
(display "exp1 is ")exp1
(display "Are exp1 and exp2 the same expression? ")(equal? exp1 exp2)
(car '(a b c))
(eval exp1)
; if a quoted expression is given to eval,
; it will return the expression without the quote
(eval (quote (+ 3 5 (/ 10 2))))
```

```
exp1 is (car (quote (a b c)))
Are exp1 and exp2 the same expression? true
a
a
13
```

exp2 showcases that we do not need to design a completely different type of construct and can reuse the list and procedure construct to store data in unconventional ways.

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

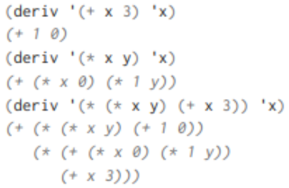
$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

Using these rules one can create a *deriv* procedure to differentiate data which contains algebraic expressions. (Interested students can look at the details of *deriv* from the textbook)

These rules ensure that the expression on the right is simpler than the expression on the left and termination conditions ensure that a terminating procedure can be written to capture these rules.



```
(deriv '(+ x 3) 'x)
(+ 1 0)
(deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
   (* (+ (* x 0) (* 1 y))
      (+ x 3)))
```

This code also presents an unsimplified answer, just like in the case of rational numbers for similar reasons.

Strings can be thought of consisting of a sequence of characters, whereas lists can be thought of consisting of a sequence of words.

Using quoted lists instead of strings allows us to directly work with bigger entities(say words). Strings also allow this functionality, but for that one would be required a regular expression approach to split strings based on white-spaces between words.

- What are the merits of string over lists and vice-versa?
- Is there a straight-forward mechanism for convertibility between strings and lists?
  Alternatively, can you come up with a mechanism to ensure that given a string *s*, can you devise a two procedures *f* and *g* such that *f* converts a string to a list and g converts a list to a string and if apply successively, then
  $(f\ (g\ x)) = (g\ (f\ x)) = x$
  Hint:- Extra Whitespaces in the string.
- Can/Should one design general purposes languages without the concept of strings?