INTRODUCTION TO SCHEME

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2019

Dalhousie University

- Functions are first-class values: Can be passed as function arguments, returned from functions.

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are mutable but mutation is discouraged.

SCHEME: A FUNCTIONAL PROGRAMMING LANGUAGE

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are mutable but mutation is discouraged.
- Imperative programming is possible but functional programming feels more natural in Scheme.

SCHEME: A FUNCTIONAL PROGRAMMING LANGUAGE

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are mutable but mutation is discouraged.
- Imperative programming is possible but functional programming feels more natural in Scheme.
- Easy to simulate a wide range of abstractions (OO, coroutines, exceptions ,...)

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are mutable but mutation is discouraged.
- Imperative programming is possible but functional programming feels more natural in Scheme.
- Easy to simulate a wide range of abstractions (OO, coroutines, exceptions ,...)

Note: Haskell is more elegant but harder to learn. You should give it a (serious) shot. Notes are still online.

We will use Chez Scheme as our Scheme interpreter:

```
$ chezscheme
Chez Scheme Version 9.5
Copyright 1984-2017 Cisco Systems, Inc.

> _
```

- Supports Scheme R^6RS
- Installed on `bluenose`

INTERACTIVE VS SCRIPT MODE

Similarly to Python, if we just type `chezscheme`, we get an interactive prompt.

In Scheme parlance, this is called the read-eval-print-loop (REPL).

INTERACTIVE VS SCRIPT MODE

Similarly to Python, if we just type `chezscheme`, we get an interactive prompt.

In Scheme parlance, this is called the read-eval-print-loop (REPL).

To run a program without dropping into interactive mode, we use the `--script` or `--program` command line option:

```
$ chezscheme --program helloworld.ss
Hello, world!
$ _
```

SCHEME SYNTAX

Scheme is a LISP dialect ("lots of irritating stupid parantheses"):

- Unusual but very simple syntax.
- Easy to parse, not so easy to read.

SCHEME SYNTAX

Scheme is a LISP dialect ("lots of irritating stupid parantheses"):

- Unusual but very simple syntax.
- Easy to parse, not so easy to read.

A Scheme program is a list of S-expressions. An S-expression is

- An atom: identifier, symbol, number, string, ...
- A list of S-expressions enclosed in parentheses

Examples

```
(define x (+ 2 3))
(display x)
```

DATA TYPES

Basic types:

- Integers
- Floats
- Booleans
- Characters
- Strings
- Symbols
- Functions

Compound types:

- Lists
- Vectors
- User-defined record types

SCHEME IS DYNAMICALLY TYPED

As Python, Scheme is dynamically typed:

- Variables have no types, values do
- As the program runs, values of different types can be assigned to the same variable.

SCHEME IS DYNAMICALLY TYPED

As Python, Scheme is dynamically typed:

- Variables have no types, values do
- As the program runs, values of different types can be assigned to the same variable.

```
> (define var 1)
> x
1
> (set! x "a string")
> x
"a string"
> _
```

INTEGERS

Literals:

- `1`, `321`, `-41`, ...

Literals:

- `1`, `321`, `-41`, ...

Arithmetic operations:

- `+`, `-`, `*`, `/`, `div` (integer division), `mod` (modulo), `expt` (power)

INTEGERS

Literals:

- `1`, `321`, `-41`, ...

Arithmetic operations:

- `+`, `-`, `*`, `/`, `div` (integer division), `mod` (modulo), `expt` (power)

Comparison:

- `=`, `<`, `>`, `<=`, `>=`

Literals:

- `1`, `321`, `-41`, ...

Arithmetic operations:

- `+`, `-`, `*`, `/`, `div` (integer division), `mod` (modulo), `expt` (power)

Comparison:

- `=`, `<`, `>`, `<=`, `>=`

```
> (+ 1 2 3)
6
> (= (- 3 2 1) 0)
#t
```

Literals:

- `1.0`, `321.0`, `-3.4e12`, ...

Literals:

- `1.0`, `321.0`, `-3.4e12`, ...

Arithmetic operations:

- `+`, `-`, `*`, `/`, `div` (integer division), `mod` (modulo), `expt` (power)
- `inexact->exact` (convert to exact number)

Literals:

- `1.0`, `321.0`, `-3.4e12`, ...

Arithmetic operations:

- `+`, `-`, `*`, `/`, `div` (integer division), `mod` (modulo), `expt` (power)
- `inexact->exact` (convert to exact number)

Comparison:

- `=`, `<`, `>`, `<=`, `>=`

Literals:

- `1.0`, `321.0`, `-3.4e12`, ...

Arithmetic operations:

- `+`, `-`, `*`, `/`, **`div`** (integer division), **`mod`** (modulo), **`expt`** (power)
- `inexact->exact` (convert to exact number)

Comparison:

- `=`, `<`, `>`, `<=`, `>=`

```
> (inexact->exact 20.0)
20
> (inexact->exact 1.2)
5404319552844595/4503599627370496
```

Literals:

- `#f` and `#t`

BOOLEANS

Literals:

- `#f` and `#t`

Logical operations:

- `and`, `or`, `not`

BOOLEANS

Literals:

- `#f` and `#t`

Logical operations:

- `and`, `or`, `not`

"Truthiness" of other types:

- Everything that's not `#f` is treated as true (even the empty list).

Literals:

- `#f` and `#t`

Logical operations:

- `and`, `or`, `not`

"Truthiness" of other types:

- Everything that's not `#f` is treated as true (even the empty list).

```
> (and 1 #t 3)
3
> (or 1 #t 3)
1
> (if '() "yes" "no")
"yes"
```

SYMBOLS

Literals:

- `'asymbol`, `'+`, ...
- Can be used to represent discrete data: `'red`, `'green`, `'blue`, ...

SYMBOLS

Literals:

- `'asymbol`, `'+`, ...
- Can be used to represent discrete data: `'red`, `'green`, `'blue`, ...

Equality testing:

- `eq?`

SYMBOLS

Literals:

- `'asymbol`, `'+`, ...
- Can be used to represent discrete data: `'red`, `'green`, `'blue`, ...

Equality testing:

- `eq?`

Conversion:

- `string->symbol`, `symbol->string`

CHARACTERS

Literals:

- `\#a`, `\#[`, ...
- Unicode characters: `\#x41`, ...
- Some named special characters: `\#tab`, `\#newline`, ...

CHARACTERS

Literals:

- `\#a`, `\#[`, ...
- Unicode characters: `\#x41`, ...
- Some named special characters: `\#tab`, `\#newline`, ...

Conversion:

- `char->integer`, `integer->char`

CHARACTERS

Literals:

- `\#a`, `\#[`, ...
- Unicode characters: `\#x41`, ...
- Some named special characters: `\#tab`, `\#newline`, ...

Conversion:

- `char->integer`, `integer->char`

Comparison:

- Case-sensitive: `char=?`, `char<?`, ...
- Case-insensitive: `char-ci=?`, `char-ci<?`, ...

Literals:

- `"This is a string with a\ttab"`

Literals:

- `"This is a string with a\ttab"`

Conversion:

- `string->list`, `list->string`, `string->number`, `number->string`

STRINGS

Literals:

- `"This is a string with a\ttab"`

Conversion:

- `string->list`, `list->string`, `string->number`, `number->string`

Comparison:

- Case-sensitive: `string=?`, `string<?`, ...
- Case-insensitive: `string-ci=?`, `string-ci<?`, ...

An anonymous function definition:

```
(lambda (x y)
  (+ x y))
```

FUNCTIONS

An anonymous function definition:

```
(lambda (x y)
  (+ x y))
```

Functions can be

FUNCTIONS

An anonymous function definition:

```
(lambda (x y)
  (+ x y))
```

Functions can be

- Applied to some arguments: `((lambda (x y) (+ x y)) 1 2)`

FUNCTIONS

An anonymous function definition:

```
(lambda (x y)
  (+ x y))
```

Functions can be

- Applied to some arguments: `((lambda (x y) (+ x y)) 1 2)`
- Passed to other functions: `(map + '(1 2) '(3 4))`

FUNCTIONS

An anonymous function definition:

```
(lambda (x y)
  (+ x y))
```

Functions can be

- Applied to some arguments: `((lambda (x y) (+ x y)) 1 2)`
- Passed to other functions: `(map + '(1 2) '(3 4))`
- Stored in variables: `(define add (lambda (x y) (+ x y)))`

An anonymous function definition:

```
(lambda (x y)
  (+ x y))
```

Functions can be

- Applied to some arguments: `((lambda (x y) (+ x y)) 1 2)`
- Passed to other functions: `(map + '(1 2) '(3 4))`
- Stored in variables: `(define add (lambda (x y) (+ x y)))`
- Returned as function results:
  `(define (mkadder inc) (lambda (x) (+ x inc)))`

DEFINITIONS

To define a variable, use the syntax `(define varname ...)`

```
; A variable `one` that stores the integer 1
(define one 1)

; A variable `mkadder` that stores a function that takes one
; argument and returns a function.
(define mkadder
  (lambda (inc)
    (lambda (x)
      (+ x inc))))
```

FUNCTION DEFINITIONS

In Scheme, a named function `fun` is nothing but a variable `fun` that stores a function:

```
> (define add
    (lambda (x y) (+ x y)))
> (add 1 2)
3
```

In Scheme, a named function `fun` is nothing but a variable `fun` that stores a function:

```
> (define add
    (lambda (x y) (+ x y)))
> (add 1 2)
3
```

This is tedious to write, so Scheme has a shorter notation for defining functions (this is only a change in syntax!):

```
> (define (add x y)
    (+ x y))
> (add 1 2)
3
```

Definitions can occur inside function bodies. They are visible only inside the function (and inside nested functions).

```
> (define x 1)
> (define (fun)
    (define x 2)
    (define (inner)
      (set! x 10))
    (inner)
    (display x) (newline))
> (fun)
10
> (display x)
1
```

CONTROL STRUCTURES

The standard if-then(-else) looks like this in Scheme:

```
(if cond then-expr)
(if cond then-expr else-expr)
```

Example:

```
(define (sign x)
  (if (< x 0)
    -1
    (if (> x 0)
      1
      0)))
```

The branches of an if-statement are single statements!
What if I want to do more than one thing in each branch?

Sequencing syntax in Scheme:

```
(begin
  expr1
  expr2
  ...)
```

The value of a `begin` block is the value of its last expression.

A MORE COMPLEX EXAMPLE USING IF AND BEGIN

```
(define (sign x)
  (if (< x 0)
    (begin
      (display "Negative number") (newline)
      -1)
    (if (> x 0)
      (begin
        (display "Positive number") (newline)
        1)
      (begin
        (display "Zero") (newline)
        0))))
```

COND

`cond` is a multi-way `if` with implicit `begin` blocks:

```
(cond
  [cond1 expr1 expr2 ...]
  [cond2 expr1 expr2 ...]
  ...
  [else expr1 expr2 ...])
```

CHATTY SIGN FUNCTION USING COND

```
(define (sign x)
  (cond [(< x 0) (display "Negative number") (newline)
                 -1]

        [(> x 0) (display "Positive number") (newline)
                 1]

        [else    (display "Zero") (newline)
                 0]))
```

Scheme has no loops!

How do we repeat things? Recursion.

```
(define (print-one-to-ten)
  (define (loop i)
    (display i) (newline)
    (if (< i 10)
      (loop (+ i 1))))
  (loop 1))
```

Scheme has no loops!

How do we repeat things? Recursion.

```
(define (print-one-to-ten)
  (define (loop i)
    (display i) (newline)
    (if (< i 10)
      (loop (+ i 1))))
  (loop 1))
```

Recursion is much more natural in functional languages.

Iteration requires side effects and thus is not possible at all in purely functional lanugages such as Haskell. (This is a *good* thing!)

EFFICIENT RECURSION: TAIL RECURSION

Recursion requires a call stack. Can become big if there are many recursive calls.

In C, C++, Java, Python, ..., iteration is more efficient than recursion.

Decent functional languages have tail recursion.

Recursion requires a call stack. Can become big if there are many recursive calls.

In C, C++, Java, Python, ..., iteration is more efficient than recursion.

Decent functional languages have tail recursion.

Tail recursion

If the return value of a function equals the return value of its last function call, we can jump to the called function without building up a stack frame.

Recursion requires a call stack. Can become big if there are many recursive calls.

In C, C++, Java, Python, ..., iteration is more efficient than recursion.

Decent functional languages have tail recursion.

Tail recursion

If the return value of a function equals the return value of its last function call, we can jump to the called function without building up a stack frame.

The compiler effectively translates a tail-recursive function back into a loop!

THE GOOD OLD FACTORIAL FUNCTION

```
(define (factorial n)
  (if (< n 2)
      1
      (* n (factorial (- n 1)))))
```

This function is not tail-recursive, so calling this on large numbers will likely blow up the stack.

```
(define (factorial n)
  (define (fac f n)
    (if (< n 2)
        f
        (fac (* n f) (- n 1))))
  (fac 1 n)))
```

COMPUTING FACTORIALS THE TAIL-RECURSIVE WAY

```
(define (factorial n)
  (define (fac f n)
    (if (< n 2)
        f
        (fac (* n f) (- n 1))))
  (fac 1 n)))
```

Compare to an iterative Python version:

```
def factorial(n):
  f = 1
  while n >= 2:
    f *= n
    n -= 1
  return f
```

SCOPES

LET BINDINGS

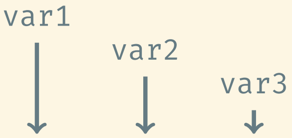
We normally do not define local variables in functions using `define`.
We use `let`-bindings:

```
(let ([var1 expr1]
      [var2 expr2]
      ...)
  ; var1, var2, ... are visible here
  ...)
; but not here
```

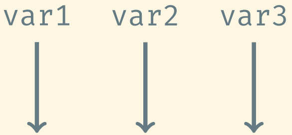
```
(let ([var1 expr1]
      [var2 expr2]
      [var3 expr3])
  ...)
```

⟵ var1, var2, var3 visible only here

```
(let* ([var1 expr1]
       [var2 expr2]
       [var3 expr3])
  ...)
```




```
(letrec ([var1 expr1]
         [var2 expr2]
         [var3 expr3])
  ...)
```



Our earlier example of using local definitions:

```
> (define x 1)
> (define (fun)
    (define x 2)
    (define (inner)
      (set! x 10))
    (inner)
    (display x) (newline))
> (fun)
10
> (display x)
1
```

AN EXAMPLE

Using a `let`-block, this looks like this:

```
> (define x 1)
> (define (fun)
    (let ([x 2])
      (define (inner)
        (set! x 10))
      (inner)
      (display x) (newline)))
> (fun)
10
> (display x)
1
```

AN EXAMPLE

But wait a second, `inner` is also a local definition:

```
> (define x 1)
> (define (fun)
    (let ([x 2]
          [inner (lambda () (set! x 10))])
      (inner)
      (display x) (newline)))
> (fun)
2
> (display x)
10
```

Oops!

AN EXAMPLE

What we need here is `let*` so `inner` modifies the correct `x`:

```
> (define x 1)
> (define (fun)
    (let* ([x 2]
           [inner (lambda () (set! x 10))])
      (inner)
      (display x) (newline)))
> (fun)
10
> (display x)
1
```

EVALUATION ORDER OF EXPRESSIONS IN LET BINDINGS

Do not count on the order of evaluation of expressions in a `let`- or `letrec`-block.

`let*` and `letrec*` guarantee sequential evaluation.

NAMED LET

Our method of turning our **`factorial`** function into a tail-recursive one is a common idiom, but it involves a lot of boilerplate:

```
(define (factorial n)
  (define (fac f n)
    (if (< n 2)
        f
        (fac (* n f) (- n 1))))
  (fac 1 n)))
```

NAMED LET

The “named let” construct allows us to write this in a way that looks like a more flexible loop construct.

```
(define (factorial n)
  (let fac ([f 1]
            [n n])
    (if (< n 2)
        f
        (fac (* n f) (- n 1)))))
```

This is 100% equivalent to our earlier definition. It is only easier to read.

In general, a named `let`-block

```
(let name ([var1 expr1]
           [var2 expr2]
           ...)
  ...
  (name arg1 arg2 ...)
  ...)
```

is translated into

```
(define (name var1 var2 ...)
   ...
   (name arg1 arg2 ...)
   ...)
(name expr1 expr2 ...)
```

COMPOUND DATA TYPES

PAIRS

Pairs are pervasive in all LISPs.

A pair holding two items x and y is constructed using

```
(cons x y)
```

PAIRS

Pairs are pervasive in all LISPs.

A pair holding two items `x` and `y` is constructed using

```
(cons x y)
```

The elements of a pair `xy` are accessed using:

```
(car xy) ; first element
(cdr xy) ; second element (pronounce "coulder")
```

PAIRS

Pairs are pervasive in all LISPs.

A pair holding two items `x` and `y` is constructed using

```
(cons x y)
```

The elements of a pair `xy` are accessed using:

```
(car xy) ; first element
(cdr xy) ; second element (pronounce "coulder")
```

Example

```
> (define xy (cons 1 2))
> (car xy)
1
> (cdr xy)
2
```

Lists are defined inductively:

LISTS

Lists are defined inductively:

- The empty list `'()` (not `()`!) is a list.

LISTS

Lists are defined inductively:

- The empty list `'()` (not `()`!) is a list.
- A pair `(cons head tail)` is a list if
  - `head` is any data item (possibly a list) and
  - `tail` is a list.

LISTS

Lists are defined inductively:

- The empty list `'()` (not `()`!) is a list.
- A pair `(cons head tail)` is a list if
  - `head` is any data item (possibly a list) and
  - `tail` is a list.

Example

```
(cons 1 (cons 2 (cons 3 '()))) ; In Python: [1, 2, 3]
```

LISTS

Lists are defined inductively:

- The empty list `'()` (not `()`!) is a list.
- A pair `(cons head tail)` is a list if
    - `head` is any data item (possibly a list) and
    - `tail` is a list.

Example

```
(cons 1 (cons 2 (cons 3 '()))) ; In Python: [1, 2, 3]
```

The recursive structure makes lists perfect as sequence types to be manipulated using recursive functions.

Next: A more convenient syntax :)

A more convenient syntax to define lists:

```
(list 1 2 3)
```

SOME CONVENIENT LIST FUNCTIONS

`(null? lst)`: Is `lst` empty?

`(length lst)`: The length of `lst`

SOME CONVENIENT LIST FUNCTIONS

`(null? lst)`: Is `lst` empty?

`(length lst)`: The length of `lst`

`(car lst)`: The first element of `lst`

`(cdr lst)`: The tail of `lst`

SOME CONVENIENT LIST FUNCTIONS

`(null? lst)`: Is `lst` empty?

`(length lst)`: The length of `lst`

`(car lst)`: The first element of `lst`

`(cdr lst)`: The tail of `lst`

`(cadr lst)` = `(car (cdr lst))`

`(cddr lst)` = `(cdr (cdr lst))`

SOME CONVENIENT LIST FUNCTIONS

`(null? lst)`: Is `lst` empty?

`(length lst)`: The length of `lst`

`(car lst)`: The first element of `lst`

`(cdr lst)`: The tail of `lst`

`(cadr lst)` = `(car (cdr lst))`

`(cddr lst)` = `(cdr (cdr lst))`

`caaaar`, `caaadr`, ..., `cddddr`

SOME CONVENIENT LIST FUNCTIONS

`(null? lst)`: Is `lst` empty?

`(length lst)`: The length of `lst`

`(car lst)`: The first element of `lst`

`(cdr lst)`: The tail of `lst`

`(cadr lst)` = `(car (cdr lst))`

`(cddr lst)` = `(cdr (cdr lst))`

`caaaar`, `caaadr`, ..., `cddddr`

`(append (list 1 2) (list 3) '() (list 4 5))` =
`(list 1 2 3 4 5)`

LIST TRANSFORMATIONS = POWERFUL CONTROL STRUCTURES

Most things that we express (quite clumsily) using loops are instances of common transformation patterns on sequences:

Most things that we express (quite clumsily) using loops are instances of common transformation patterns on sequences:

- `map`: Apply a function to every element in a sequence. (Generalizes to tuples of sequences using multivariate functions.)

LIST TRANSFORMATIONS = POWERFUL CONTROL STRUCTURES

Most things that we express (quite clumsily) using loops are instances of common transformation patterns on sequences:

- `map`: Apply a function to every element in a sequence. (Generalizes to tuples of sequences using multivariate functions.)
- `filter`: Extract a subsequence of elements satisfying a given predicate.

LIST TRANSFORMATIONS = POWERFUL CONTROL STRUCTURES

Most things that we express (quite clumsily) using loops are instances of common transformation patterns on sequences:

- `map`: Apply a function to every element in a sequence. (Generalizes to tuples of sequences using multivariate functions.)
- `filter`: Extract a subsequence of elements satisfying a given predicate.
- `fold-left`: Accumulate the elements in a list left-to-right (e.g., sum)

LIST TRANSFORMATIONS = POWERFUL CONTROL STRUCTURES

Most things that we express (quite clumsily) using loops are instances of common transformation patterns on sequences:

- `map`: Apply a function to every element in a sequence. (Generalizes to tuples of sequences using multivariate functions.)
- `filter`: Extract a subsequence of elements satisfying a given predicate.
- `fold-left`: Accumulate the elements in a list left-to-right (e.g., sum)
- `fold-right`: Accumulate the elements in a list right-to-left

LIST TRANSFORMATIONS = POWERFUL CONTROL STRUCTURES

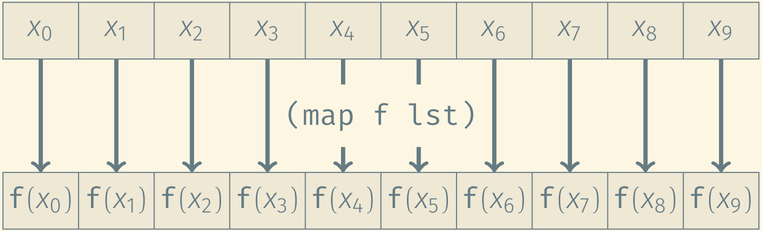
Most things that we express (quite clumsily) using loops are instances of common transformation patterns on sequences:

- `map`: Apply a function to every element in a sequence. (Generalizes to tuples of sequences using multivariate functions.)
- `filter`: Extract a subsequence of elements satisfying a given predicate.
- `fold-left`: Accumulate the elements in a list left-to-right (e.g., sum)
- `fold-right`: Accumulate the elements in a list right-to-left

We can write powerful programs in terms of these transformations (Google MapReduce).

If we want to apply a function to each element in a list and produce a list of the results, this is what `map` does:

```
> (map (lambda (x) (* 2 x))
       (list 1 2 3 4 5))
(2 4 6 8 10)
```



We can also do this to two (or more) lists:

```
> (map + (list 1 2 3 4 5)
         (list 6 7 8 9 10))
(7 9 11 13 15)
```

(All input lists must have the same length!)

Another common idiom is to extract the sublist of elements that meet a given condition (or predicate). `filter` takes care of this:

```
> (filter even? (list 1 2 3 4 5))
(2 4)
```

FOLD-LEFT

Assume for now that `+` accepts only two arguments.

How do we implement a function `sum` that sums the elements in a list?

FOLD-LEFT

Assume for now that `+` accepts only two arguments.

How do we implement a function `sum` that sums the elements in a list?

```
(define (sum lst) (fold-left + 0 lst))
```

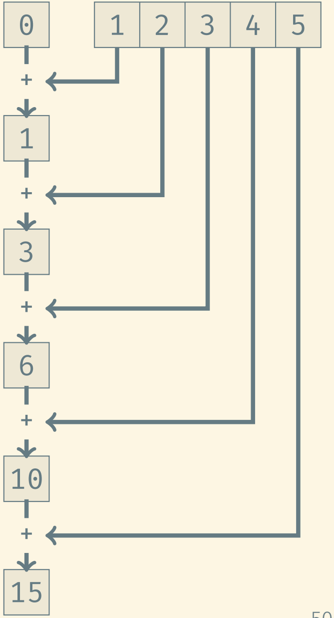
Assume for now that `+` accepts only two arguments.

How do we implement a function `sum` that sums the elements in a list?

```
(define (sum lst) (fold-left + 0 lst))
```

Example

```
(fold-left + 0 (list 1 2 3 4 5))
```



FOLD-RIGHT

We could implement `sum` by folding right-to-left instead:

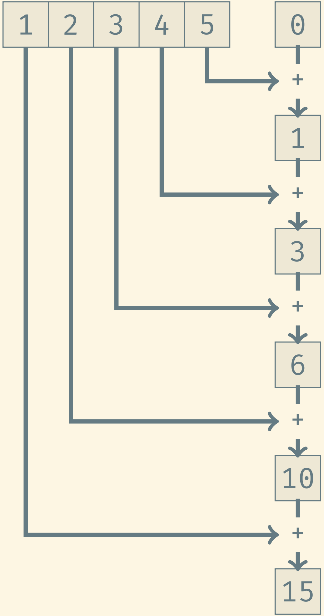
```
(define (sum lst) (fold-right + 0 lst))
```

We could implement `sum` by folding right-to-left instead:

```
(define (sum lst) (fold-right + 0 lst))
```

Example

```
(fold-right + 0 (list 1 2 3 4 5))
```



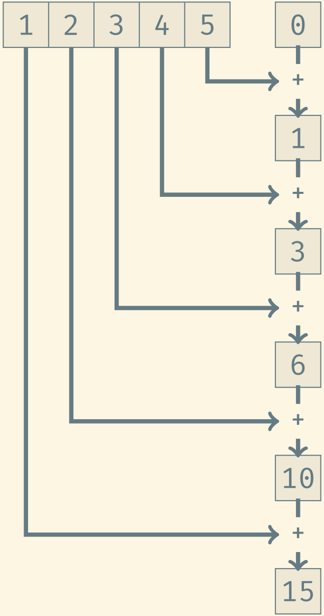
We could implement `sum` by folding right-to-left instead:

```
(define (sum lst) (fold-right + 0 lst))
```

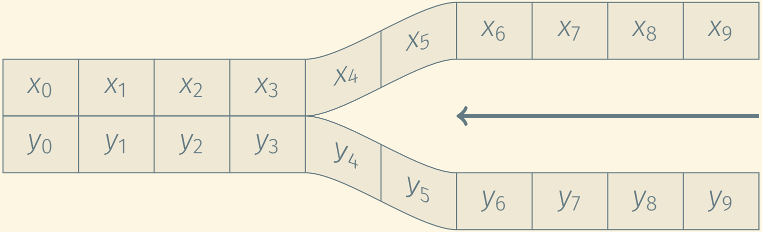
Example

```
(fold-right + 0 (list 1 2 3 4 5))
```

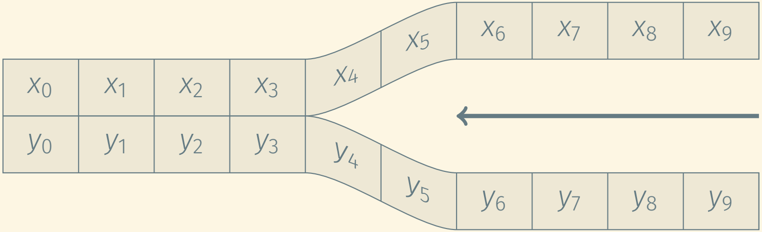
This is generally less efficient than left-folding but has its uses!



Sometimes, we have a pair (or more) lists where the *i*th elements in these lists are logically associated with each other. We may want to combine them into a list of pairs (or tuples) of associated elements.

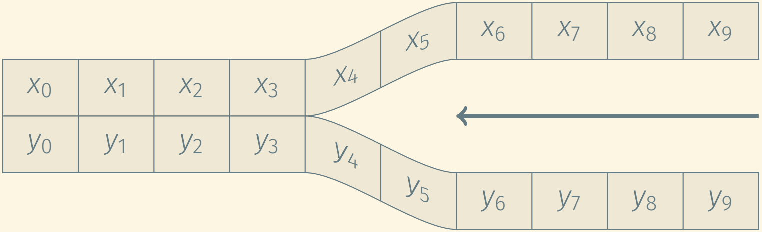


Sometimes, we have a pair (or more) lists where the *i*th elements in these lists are logically associated with each other. We may want to combine them into a list of pairs (or tuples) of associated elements.



`zip` to the rescue …

Sometimes, we have a pair (or more) lists where the *i*th elements in these lists are logically associated with each other. We may want to combine them into a list of pairs (or tuples) of associated elements.

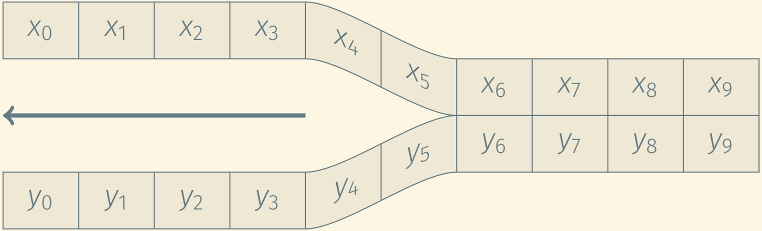


`zip` to the rescue … except that it does not exist in Scheme.

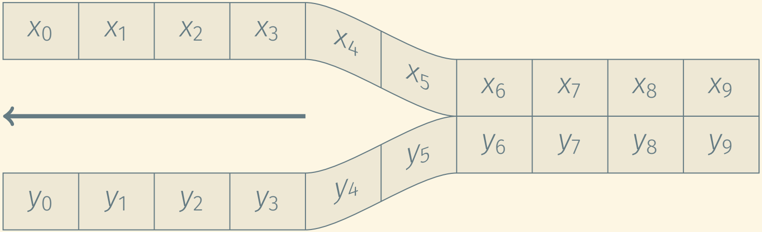
`zip` can be implemented easily enough using `map` and `list`:

```
> (map list (list  1      2      3       )
            (list  "one"  "two"  "three" )
            (list  #f     #t     #f      ))
((1 "one" #f) (2 "two" #t) (3 "three" #f))
```

Functional languages that have an explicit **`zip`** function usually also have an **`unzip`** function, which takes a list of pairs or a list of tuples and turns it into a tuple of lists:



Functional languages that have an explicit `zip` function usually also have an `unzip` function, which takes a list of pairs or a list of tuples and turns it into a tuple of lists:



Scheme does not have this one either, but it is implemented easily enough using `map`, `list`, and `apply`.

APPLY

`apply` allows us to apply a function to a list of arguments not given as part of the source code but in an actual list.

`apply` allows us to apply a function to a list of arguments not given as part of the source code but in an actual list.

Normal application of `+`:

```
> (+ 1 2 3 4 5)
15
```

APPLY

`apply` allows us to apply a function to a list of arguments not given as part of the source code but in an actual list.

Normal application of `+`:

```
> (+ 1 2 3 4 5)
15
```

If `1 2 3 4 5` are given in a list, we can use `apply` to sum them:

```
> (define lst (list 1 2 3 4 5))
> (apply + lst)
15
```

In general,

```
(apply fun arg1 arg2 ... (list larg1 larg2 ...))
```

does the same as

```
(fun arg1 arg2 ... larg1 larg2 ...)
```

In general,

```
(apply fun arg1 arg2 ... (list larg1 larg2 ...))
```

does the same as

```
(fun arg1 arg2 ... larg1 larg2 ...)
```

So, we could add some additional terms to the sum of the elements in the list:

```
> (define lst (list 1 2 3 4 5))
> (apply + 6 7 lst)
28
```

BACK TO UNZIP

An implementation of `unzip` using `map`, `apply`, and `list`:

```
> (define lst
    (list (list 1 "one"   #f)
          (list 2 "two"   #t)
          (list 3 "three" #f)))
> (apply map list lst)
((1 2 3)
 ("one" "two" "three")
 (#f #t #f))
```

IMPLEMENTING MAP

`map`, `filter`, `fold-left`, `fold-right` can all be implemented easily using recursion.

They are simply common enough abstractions that they are provided in the standard library.

`map`, `filter`, `fold-left`, `fold-right` can all be implemented easily using recursion.

They are simply common enough abstractions that they are provided in the standard library.

Implementation of `map`

```
(define (map fun lst)
  (cond [(null? lst)] '()]
        [else         (cons (fun (car lst))
                            (map fun (cdr lst)))]))
```

Implementation of `filter`

```
(define (filter pred lst)
  (cond [(null? lst)]     '()]
        [(pred (car lst)) (cons (car lst)
                                (filter pred (cdr lst)))]
        [else             (filter pred (cdr lst))]))
```

Implementation of `filter`

```
(define (filter pred lst)
  (cond [(null? lst)]      '()]
        [(pred (car lst)) (cons (car lst)
                                (filter pred (cdr lst)))]
        [else             (filter pred (cdr lst))]))
```

More efficient implementations exist, but they are less pretty!

Vectors in Scheme are like C arrays (and unlike vectors in C++, Python, Java, …):
Their length is fixed!

Advantage over lists: Items can be accessed by index in constant time.

CREATING VECTORS

We create a vector containing `count` copies of `item` using

```
> (make-vector count item)
```

Example: A Boolean vector of all false values

```
> (make-vector 10 #f)
#(#f #f #f #f #f #f #f #f #f #f)
```

CREATING VECTORS

We create a vector containing `count` copies of `item` using

```
> (make-vector count item)
```

Example: A Boolean vector of all false values

```
> (make-vector 10 #f)
#(#f #f #f #f #f #f #f #f #f #f)
```

The analog of `list` is `vector`:

Example: A vector containing the elements 1, ..., 5

```
> (vector 1 2 3 4 5)
#(1 2 3 4 5)
```

CONVERSION TO AND FROM LISTS

Conversion between lists and vectors is useful in functional programming:

- Lists are recursive, good for functional programming.
- Vectors provide fast element-wise access.

CONVERSION TO AND FROM LISTS

Conversion between lists and vectors is useful in functional programming:

- Lists are recursive, good for functional programming.
- Vectors provide fast element-wise access.

Conversion from vector to list

```
> (vector->list (vector 1 2 3 4 5))
(1 2 3 4 5)
```

CONVERSION TO AND FROM LISTS

Conversion between lists and vectors is useful in functional programming:

- Lists are recursive, good for functional programming.
- Vectors provide fast element-wise access.

Conversion from vector to list

```
> (vector->list (vector 1 2 3 4 5))
(1 2 3 4 5)
```

Conversion from list to vector

```
> (list->vector (list 1 2 3 4 5))
#(1 2 3 4 5)
```

Read a vector element

```
> (vector-ref (vector 2 4 6 8 10) 2)
6
```

ELEMENT ACCESS AND LENGTH

Read a vector element

```
> (vector-ref (vector 2 4 6 8 10) 2)
6
```

Update a vector element

```
> (define vec (vector 2 4 6 8 10))
> (vector-set! vec 2 7)
> vec
#(2 4 7 8 10)
```

ELEMENT ACCESS AND LENGTH

Read a vector element

```
> (vector-ref (vector 2 4 6 8 10) 2)
6
```

Update a vector element

```
> (define vec (vector 2 4 6 8 10))
> (vector-set! vec 2 7)
> vec
#(2 4 7 8 10)
```

The length of a vector

```
> (vector-length (vector 2 4 6 8 10))
5
```

VECTOR-MAP AND VECTOR-FOREACH

`vector-map` is the equivalent of `map`:

```
> (vector-map + (vector 1 2 3 4 5) (vector 10 9 8 7 6))
#(11 11 11 11 11)
```

VECTOR-MAP AND VECTOR-FOREACH

`vector-map` is the equivalent of `map`:

```
> (vector-map + (vector 1 2 3 4 5) (vector 10 9 8 7 6))
#(11 11 11 11 11)
```

Chez Scheme parallelizes `map` and `vector-map`, so do not count on the evaluation order:

```
> (vector-map display (vector 1 2 3 4 5))
34521#(#<void> #<void> #<void> #<void> #<void>)
```

VECTOR-MAP AND VECTOR-FOREACH

`vector-map` is the equivalent of `map`:

```
> (vector-map + (vector 1 2 3 4 5) (vector 10 9 8 7 6))
#(11 11 11 11 11)
```

Chez Scheme parallelizes `map` and `vector-map`, so do not count on the evaluation order:

```
> (vector-map display (vector 1 2 3 4 5))
34521#(#<void> #<void> #<void> #<void> #<void>)
```

If you care about the evaluation order and only the side effects matter, use `for-each` or `vector-for-each`:

```
> (vector-for-each display (vector 1 2 3 4 5))
12345
```

Records are like `structs` in C (classes without methods).

Define a record

```
> (define-record-type point (fields x y))
```

There are lots of things that can be customized using more detailed arguments:

- A constructor function
- Mutability of fields (default = immutable!)
- ...

CREATE AND ACCESS A RECORD

Create an object of the defined record type

```
> (define p (make-point 1 2))
```

CREATE AND ACCESS A RECORD

Create an object of the defined record type

```
> (define p (make-point 1 2))
```

Test whether an object is of a particular record type

```
> (point? p)
#t
> (point? 1)
#f
```

CREATE AND ACCESS A RECORD

Create an object of the defined record type

```
> (define p (make-point 1 2))
```

Test whether an object is of a particular record type

```
> (point? p)
#t
> (point? 1)
#f
```

Access the fields of a record

```
> (point-x p)
1
> (point-y p)
2
```

MUTABLE FIELDS

If we want the fields to be mutable, we need to say so:

```
> (define-record-type point (fields x (mutable y)))
> (define p (make-point 1 2))
> (point-y-set! p 3)
> (point-y p)
3
> (point-x-set! p 2)
Exception: variable point-x-set! is not bound
Type (debug) to enter the debugger.
```

CODE ORGANIZATION

Single-file programs in Scheme are easy:

fibs.ss

```
#! env scheme-script

(import (rnrs (6))) ; The import statement is required

; Compute the first n+1 Fibonacci numbers F0, ..., Fn
(define (fibs n)
  (let loop ([i 0]
             [cur 1]
             [prev 0])
    (cond [(> i n) '()]
          [else    (cons cur (loop (+ i 1) (+ cur prev) cur))])))

; (Continued on next page)
```

fibs.ss (Continued)

```
; Print a sequence of numbers
(define (print-seq seq)
  (let loop ([seq seq])
    (cond [(null? seq) (newline)]
          [else       (display (car seq))
                      (display " ")
                      (loop (cdr seq))])))

; No safety checks of any kind, for brevity!
(define n (string->number (cadr (command-line))))

(print-seq (fibs n))
```

MULTI-FILE PROJECTS

Larger projects should be broken up into separate source code files.

fibs.ss

```
#! env scheme-script

(import (rnrs (6))
        (fibs generator)
        (only (fibs printer) print-seq))

; No safety checks of any kind, for brevity!
(define n (string->number (cadr (command-line))))

(print-seq (fibs n))
```

fibs/generator.ss

```
(library (fibs generator (1))
  (export fibs)
  (import (rnrs (6)))

  ; Compute the first n+1 Fibonacci numbers F0, ..., Fn
  (define (fibs n)
    (let loop ([i 0]
               [cur 1]
               [prev 0])
      (cond [(> i n) '()]
            [else    (cons cur (loop (+ i 1) (+ cur prev) cur))]))))
```

fibs/printer.ss

```
(library (fibs printer (1))
  (export print-seq)
  (import (rnrs (6)))

  ; Print a sequence of numbers
  (define (print-seq seq)
    (let loop ([seq seq])
      (cond [(null? seq) (newline)]
            [else        (display (car seq))
                         (display " ")
                         (loop (cdr seq))]))))
```

LIBRARY SEARCH PATH

A library with name `(part1 part2 part3)` is located as one of the following:

- `$SCHEMELIBDIR/part1/part2/part3.ss`
- `$SCHEMELIBDIR/part1/part2/part3.sls`
- `./part1/part2/part3.ss`
- `./part1/part2/part3.sls`

So the project above should be structured as:

```
$ tree .
.
├── fibs
│   ├── generator.ss
│   └── printer.ss
└── fibs.ss
```

A BIGGER EXAMPLE: MERGE SORT

MERGE SORT

```
;;; A simple sorting library

(library (sorting (1))
  (export merge merge-sort)
  (import (rnrs (6))
          (only (chezscheme) list-head))

  ;; Sort the list `lst` by the comparison function `cmp`
  (define (merge-sort cmp lst)
    (define (recurse lst)
      (let ([n (length lst)])
        (if (< n 2)
            lst
            (apply merge cmp (map recurse (split-list n lst))))))
    (recurse lst))

  ; (Continued on next page)
```

MERGE

```
;; Merge two sorted lists by a comparison function `cmp`
(define (merge cmp left right)
  (let loop ([left left]
             [right right]
             [merged '()])
    (cond [(null? left)  (fold-left (flip cons) right merged)]
          [(null? right) (fold-left (flip cons) left  merged)]
          [(cmp (car right) (car left))
           (loop left (cdr right) (cons (car right) merged))]
          [else
           (loop (cdr left) right (cons (car left) merged))])))

; (Continued on next page)
```

```
;; Split a list into two halves
(define (split-list n lst)
  (let ([l (div n 2)])
    (list (list-head lst l)
          (list-tail lst l))))

;; Helper function every Haskell-lover needs
;; Swaps the arguments of a two-argument function `fun`
(define (flip fun)
  (lambda (x y) (fun y x)))
```

TEST HARNESS

```
#! env scheme-script

(import (rnrs (6))
        (sequences)
        (sorting))

; Get an input size
(define n    (string->number (cadr   (command-line))))
(define low  (string->number (caddr  (command-line))))
(define high (string->number (cadddr (command-line))))

(let* ([seq        (random-seq n low high)]
       [sorted-seq (merge-sort < seq)])
  (display "--- INPUT SEQUENCE ---") (newline)
  (print-seq seq)
  (display "--- OUTPUT SEQUENCE ---") (newline)
  (print-seq sorted-seq))
```

EQUALITY AND ASSOCIATION LISTS

Scheme has three notions of equality of objects:

- `eq?`: The two objects are identical.
- `eqv?`: As `eq?` but slightly coarser.
- `equal?`: The two objects are structurally the same.

Most of the time, you want `equal?`.

However, `eq?` and `eqv?` are faster.

Historically, Scheme did not have hashtables ... but it had lists.

We can store some elements in a list to represent a set:

A list of elements

```
> (define set (list 4 5 6))
```

Historically, Scheme did not have hashtables ... but it had lists.

We can store some elements in a list to represent a set:

A list of elements

```
> (define set (list 4 5 6))
```

and then ask whether an element is a member:

Membership queries over this list

```
> (member 5 set)
(5 6)                   ; This returns the tail of the list
                        ; after the first match

> (member 2 set)
#f
```

ASSOCIATION LISTS

Similarly, we can use lists as (not very efficient) dictionaries:

An association list

```
> (define alist '((1 . "one") (2 . "two") (3 . "three")))
```

Similarly, we can use lists as (not very efficient) dictionaries:

An association list

```
> (define alist '((1 . "one") (2 . "two") (3 . "three")))
```

and then ask for the first pair whose key (first element) matches a given value:

Lookup queries on this association list

```
> (assoc 2 alist)
(2 . "two")
> (assoc 4 alist)
#f
```

HOW ARE MATCHES CHOSEN?

Membership queries:

- `member` uses `equal?`
- `memv` uses `eqv?`
- `memq` uses `eq?`

Dictionary lookups:

- `assoc` uses `equal?`
- `assv` uses `eqv?`
- `assq` uses `eq?`

MUTATION

MUTABLE VARIABLES

Mutable variables are the source of a large number of software bugs.
Use them sparingly.

MUTABLE VARIABLES

Mutable variables are the source of a large number of software bugs.
Use them sparingly.

Some things cannot be done as efficiently in a purely functional fashion as using mutable state.

MUTABLE VARIABLES

Mutable variables are the source of a large number of software bugs.
Use them sparingly.

Some things cannot be done as efficiently in a purely functional fashion as using mutable state.

Scheme supports the mutation of variables to support this style of stateful programming:

- We have seen `vector-set!` to update a vector.
- `(set! var val)` replaces the value in the variable `var` with `val`.

ADVANCED TOPICS

MULTIPLE RETURN VALUES

Python creates the illusion of multiple return values by automatically creating and unpacking lists:

```
def fun():
  return 1, 2

x, y = fun()
print("{}, {}".format(x, y))
```

This is equivalent to:

```
def fun():
  return [1, 2]

[x, y] = fun()
print("{}, {}".format(x, y))
```

The scheme version of this is:

```
(define (fun)
  (list 1 2))

(define lst (fun))
(display (format "~A, ~A~%" (car lst) (cadr lst)))
```

To avoid manually unpacking these values, Scheme allows us to explicitly return multiple values from a function:

```
(define (fun)
  (values 1 2))

(define-values (x y) (fun))
(display (format "~A, ~A~%" x y))
```

In general,

```
(values expr1 expr2 ...)
```

is a function with multiple return values `expr1`, `expr2`, …

In general,

```
(values expr1 expr2 ...)
```

is a function with multiple return values `expr1`, `expr2`, ...

Using this as the last expression in a function definition `fun` results in `fun` having return values `expr`, `expr2`, ...

In general,

```
(values expr1 expr2 ...)
```

is a function with multiple return values `expr1`, `expr2`, ...

Using this as the last expression in a function definition `fun` results in `fun` having return values `expr`, `expr2`, ...

```
(define-values (var1 var2 ...) fun)
```

then assigns the values returned by `fun` to variables `var1`, `var2`, ...

VALUES, DEFINE-VALUES, AND LET-VALUES

In general,

```
(values expr1 expr2 ...)
```

is a function with multiple return values `expr1`, `expr2`, ...

Using this as the last expression in a function definition `fun` results in `fun` having return values `expr`, `expr2`, ...

```
(define-values (var1 var2 ...) fun)
```

then assigns the values returned by `fun` to variables `var1`, `var2`, ...

There also exists a version of `let` that assigns multiple values:

```
(let-values ([(var1 var2 ...) fun])
  ...)
```

QUOTING AND EVALUATING THINGS

`(+ 1 2)` computes $1 + 2$.

What if we want to store the expression `(+ 1 2)` in a variable without evaluating it?

Then we need to quote it:

```
> (define expr (quote (+ 1 2)))
```

`(+ 1 2)` computes $1 + 2$.

What if we want to store the expression `(+ 1 2)` in a variable without evaluating it?

Then we need to quote it:

```
> (define expr (quote (+ 1 2)))
```

If we want to know the value of the expression later, we can evaluate it:

```
> (eval expr)
3
```

QUOTING AND EVALUATING THINGS

`(+ 1 2)` computes $1 + 2$.

What if we want to store the expression `(+ 1 2)` in a variable without evaluating it?

Then we need to quote it:

```
> (define expr (quote (+ 1 2)))
```

If we want to know the value of the expression later, we can evaluate it:

```
> (eval expr)
3
```

This works with arbitrarily complex Scheme expression!

A SHORTER NOTATION FOR QUOTING THINGS

Quoting things is common and writing `(quote ...)` quickly becomes tedious.

A SHORTER NOTATION FOR QUOTING THINGS

Quoting things is common and writing `(quote ...)` quickly becomes tedious.

There exists a shorthand: `'expr` is the same as `(quote expr)`.

Quoting things is common and writing `(quote ...)` quickly becomes tedious.

There exists a shorthand: `'expr` is the same as `(quote expr)`.

And suddenly the notation for symbols makes sense:

- `name` refers to the value stored in the variable `name`.
- `'name` (or `(quote name)`) refers to the name `name` itself, a symbol.

We have written `(list 1 2 3 4 5)` for the list `(1 2 3 4 5)` so far.

We have written `(list 1 2 3 4 5)` for the list `(1 2 3 4 5)` so far.

By quoting the expression `(1 2 3 4 5)`, we obtain a shorter notation for lists:

```
> (define lst '(1 2 3 4 5))
> (cadr lst)
2
```

We have written `(list 1 2 3 4 5)` for the list `(1 2 3 4 5)` so far.

By quoting the expression `(1 2 3 4 5)`, we obtain a shorter notation for lists:

```
> (define lst '(1 2 3 4 5))
> (cadr lst)
2
```

The same works for vectors:

```
> (define vec '#(1 2 3 4 5))
> (vector-ref 3)
4
```

QUASI-QUOTATION

Can we avoid the tedious `(list ...)` notation if the elements in the list aren't constants?

Can we avoid the tedious `(list ...)` notation if the elements in the list aren't constants?

```
> (define var 3)
> '(1 2 var 4 5)
(1 2 var 4 5)
```

Can we avoid the tedious `(list ...)` notation if the elements in the list aren't constants?

```
> (define var 3)
> '(1 2 var 4 5)
(1 2 var 4 5)
```

Quotation stores the *entire* expression unevaluated.

QUASI-QUOTATION

Can we avoid the tedious `(list ...)` notation if the elements in the list aren't constants?

```
> (define var 3)
> '(1 2 var 4 5)
(1 2 var 4 5)
```

Quotation stores the *entire* expression unevaluated.

Quasi-quotation combined with the **`unquote`** special form, we can choose to substitute the results of evaluating an expression into a quoted expression:

```
> (define var 3)
> (quasiquote (1 2 (unquote var) 4 5)
(1 2 3 4 5)
> (quote (1 2 (unquote var) 4 5)
(1 2 ,var 4 5)
```

UNQUOTE-SPLICING

`unquote-splicing` lets us insert a list into a quasi-quoted list:

```
> (define lst '(3 4))
> (quasiquote (1 2 (unquote lst) 5))
(1 2 (3 4) 5)
> (quasiquote (1 2 (unquote-splicing lst) 5))
(1 2 3 4 5)
> (quote (1 2 (unquote-splicing lst) 5))
(1 2 ,@lst 5)
```

SHORTHANDS FOR QUASIQUOTE, UNQUOTE, UNQUOTE-SPLICING

`quasiquote`, `unquote`, and `unquote-splicing` are very useful for building lists but are tedious to write.

SHORTHANDS FOR QUASIQUOTE, UNQUOTE, UNQUOTE-SPLICING

`quasiquote`, `unquote`, and `unquote-splicing` are very useful for building lists but are tedious to write.

Again, we have shorthands for these expressions:

- `` `expr `` is the same as `(quasiquote expr)`.
- `,expr` is the same as `(unquote expr)`.
- `,@expr` is the same as `(unquote-splicing expr)`.

SHORTHANDS FOR QUASIQUOTE, UNQUOTE, UNQUOTE-SPLICING

`quasiquote`, `unquote`, and `unquote-splicing` are very useful for building lists but are tedious to write.

Again, we have shorthands for these expressions:

- `` `expr `` is the same as `(quasiquote expr)`.
- `,expr` is the same as `(unquote expr)`.
- `,@expr` is the same as `(unquote-splicing expr)`.

```
> (define lst '(3 4))
> `(1 2 ,lst 5)
(1 2 (3 4) 5)
> `(1 2 ,@lst 5)
(1 2 3 4 5)
> '(1 2 ,@lst 5)
(1 2 ,@lst 5)
```

MACROS

C has a preprocessor and, as a result, macros that can rewrite the program text.

These macros are not hygienic: temporary variable names inside the macro can clash with variables outside the macro.

Consider:

```
#define swap(x, y) int tmp = x; x = y; y = tmp;

int foo() {
  int x = 1;
  int tmp = 2;
  swap(x, tmp);
}
```

C has a preprocessor and, as a result, macros that can rewrite the program text.

These macros are not hygienic: temporary variable names inside the macro can clash with variables outside the macro.

Consider:

```
#define swap(x, y) int tmp = x; x = y; y = tmp;

int foo() {
  int x = 1;
  int tmp = 2;
  swap(x, tmp);
}
```

The C preprocessor is also not a very powerful language, so the complexity of macros that can (sanely) be written is limited.

This is only a brief introduction. For a deeper discussion, see

- The Scheme Programming Language, Chapter 8
  `https://www.scheme.com/tspl4/syntax.html#./syntax:h0`
- Fear of Macros
  `https://www.greghendershott.com/fear-of-macros/all.html`

General form of a macro definition

```
(define-syntax macro
(syntax-rules (<keywords>)
  [(<pattern>) <template>]
  ...
  [(<pattern>) <template>)])
```

A WHILE-LOOP

We would like to add a construct

```
(while condition
  body ...)
```

to our language.

A WHILE-LOOP

We would like to add a construct

```
(while condition
  body ...)
```

to our language.

Here is how we do this:

```
(define-syntax while
  (syntax-rules ()
    [(while condition body ...)
     (let loop ()
       (if condition
           (begin body ... (loop))
           (void)))]))
```

A PYTHON-LIKE FOR-LOOP

How about a for-loop as in Python:

```
(for elem in lst
  body ...)
```

to our language.

A PYTHON-LIKE FOR-LOOP

How about a for-loop as in Python:

```
(for elem in lst
  body ...)
```

to our language.

The following works but is a bit too flexible:

```
(define-syntax for
  (syntax-rules ()
    [(for elem in lst body ...)
     (for-each (lambda (elem)
                 body ...)
               lst)]))
```

TOO MUCH FLEXIBILITY

We can now write

```
> (for i in '(1 2 3 4 5)
    (display i) (display " "))
1 2 3 4 5
```

but also

```
> (for i as '(1 2 3 4 5)
    (display i) (display " "))
1 2 3 4 5
```

or

```
> (for i doodledidoo '(1 2 3 4 5)
    (display i) (display " "))
1 2 3 4 5
```

INTRODUCING LITERAL KEYWORDS

```
(define-syntax for
  (syntax-rules (in as)
    [(for elem in lst body ...)
     (for-each (lambda (elem)
                 body ...)
               lst)]
    [(for lst as elem body ...)
     (for elem in lst body ...)]))
```

INTRODUCING LITERAL KEYWORDS

```
(define-syntax for
  (syntax-rules (in as)
    [(for elem in lst body ...)
     (for-each (lambda (elem)
                 body ...)
                 lst)]
    [(for lst as elem body ...)
     (for elem in lst body ...)]))
```

Now, only the following two forms are permissible:

```
> (for i in '(1 2 3 4 5) (display i) (display " "))
1 2 3 4 5
```

```
> (for '(1 2 3 4 5) as i (display i) (display " "))
1 2 3 4 5
```

`define-syntax` defines a syntax transformer function that is used at load time to rewrite the source code.

A SLIGHTLY DEEPER LOOK AT SYNTAX TRANSFORMERS

`define-syntax` defines a syntax transformer function that is used at load time to rewrite the source code.

This is indeed a full-blown Scheme function!

`syntax-rules` is itself a macro that makes it easier to write such functions.

`define-syntax` defines a syntax transformer function that is used at load time to rewrite the source code.

This is indeed a full-blown Scheme function!

`syntax-rules` is itself a macro that makes it easier to write such functions.

If we use such macros during load time, we need a macro expansion phase for the macro expansion code itself.

Scheme allows us to layer an arbitrary number of such macro expansion phases on top of each other.

A MACRO DEFINITION WITHOUT SYNTAX-RULES

```
(define-syntax (ten-times stx)
  (let* ([body (cdr (syntax->list stx))]
         [repeated (let loop ([i 0]
                              [rep '()])
                     (if (< i 10)
                         (loop (+ i 1) `(,@body ,@rep))
                         rep))])
    #`(begin #,@repeated)))
```

You are familiar with breakpoints in debuggers.

This suspended computation has a future, what will happen if we continue from the breakpoint.

CONTINUATIONS

You are familiar with breakpoints in debuggers.

This suspended computation has a future, what will happen if we continue from the breakpoint.

The formal term for this future is “continuation”.

CONTINUATIONS

You are familiar with breakpoints in debuggers.

This suspended computation has a future, what will happen if we continue from the breakpoint.

The formal term for this future is “continuation”.

Continuations exist in all languages.

Scheme allows us to capture continuations as objects, pass them between functions, and store them in variables.

`(call-with-current-continuation fun)` or `(call/cc fun)` calls `fun` with one argument, the current continuation.

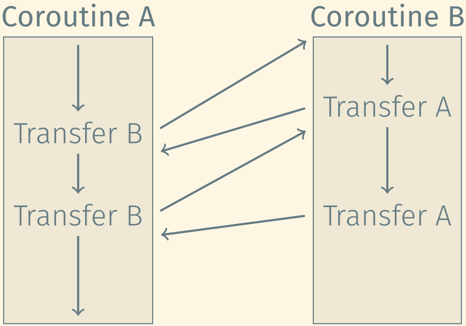
Example

```
> (define (find-first-odd lst)
    (call/cc (lambda (found)
               `(failure
                 ,(let search [(lst lst)]
                    (cond [(null? lst) #f]
                    [(even? (car lst)) (display (car lst))
                                       (display " ")
                                       (search (cdr lst))]
                    [else (found `(success ,(car lst)))]))))))
> (find-first-odd '(4 8 7 2 3)
4 8 (success 7)
> (find-first-odd '(2 4 6 8 10))
2 4 6 8 10 (failure #f)
```

EXCEPTION HANDLING USING CONTINUATIONS

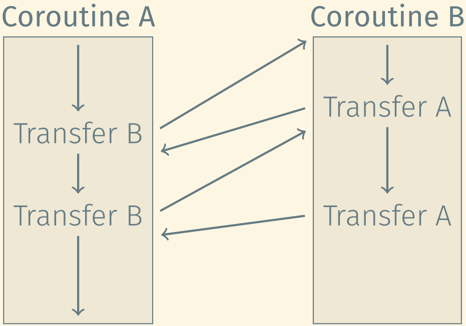
```
(define (double-odds lst)
  (let ([result
         (call/cc
           (lambda (throw)
             `(succ ,(let loop ([lst lst])
                       (cond [(null? lst) '()]
                             [(even? (car lst))
                              (throw '(err "Found an even number"))]
                             [else
                              (cons (* 2 (car lst))
                                    (loop (cdr lst)))])))))])
    (if (eq? (car result) 'succ)
        (cadr result)
        (begin (display (cadr result))
               (newline)
               (exit 1)))))
```

Coroutines are separate threads of execution that voluntarily transfer control to each other. (Contrast this with threads.)



COROUTINES

Coroutines are separate threads of execution that voluntarily transfer control to each other. (Contrast this with threads.)



Useful to implement generators, e.g., in Python

```
(define (range yield start end)
  (let* [(cur start)
         (resume (call/cc (lambda (r) r)))]
    (if (< cur end)
        (begin (set! cur (+ cur 1))
               (yield (- cur 1) resume))
        (yield #f resume))))

(define (print-range start end)
  (let-values ([(val resume) (call/cc (lambda (yield)
                                        (range yield start end)))])
    (if val
        (begin (display val)
               (newline)
               (resume resume)))))
```

EXCEPTIONS

Scheme has a fairly powerful exception handling mechanism.

For details, read The Scheme Programming Language, Chapter 11.
`https://www.scheme.com/tspl4/exceptions.html#./exceptions:h0`.