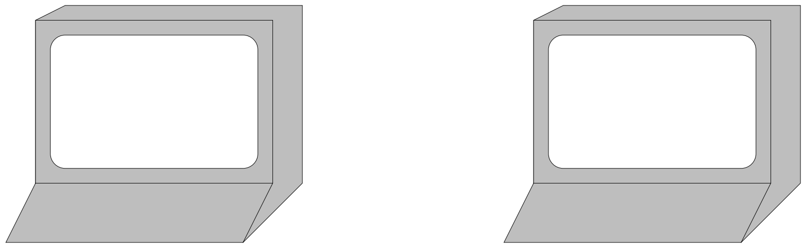
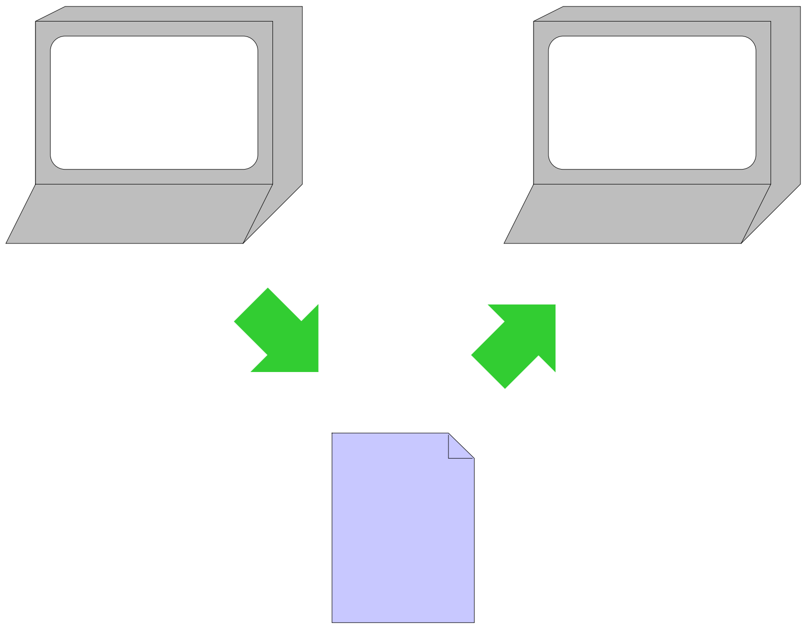
Multiple Programs

How do programs communicate?



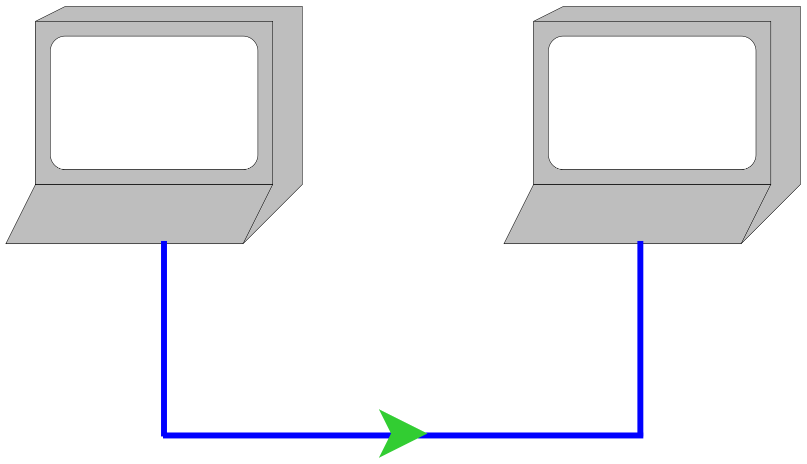
Multiple Programs

How do programs communicate? Files...



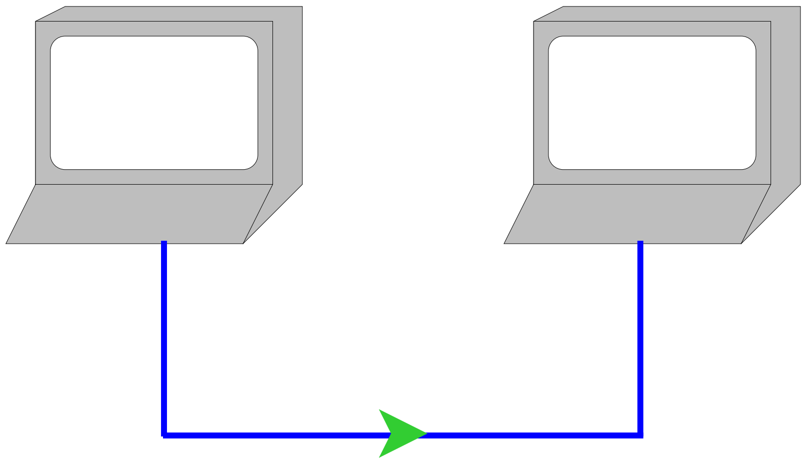
Multiple Programs

How do programs communicate? Files... Network...



Multiple Programs

How do programs communicate? Files... Network... Etc.



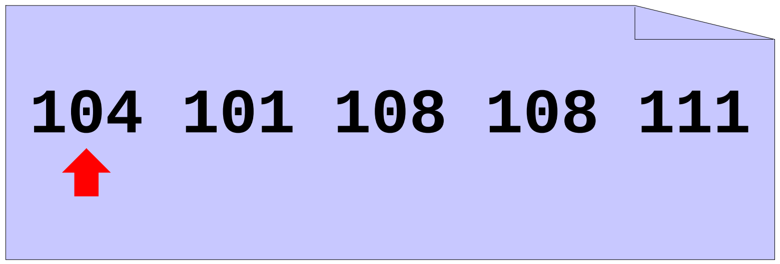
But what's in a file or sent over the network?

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between **`0`** and **`255`**

A ***stream*** is a sequence with a pointer and an operation: **`read`** or **`write`**

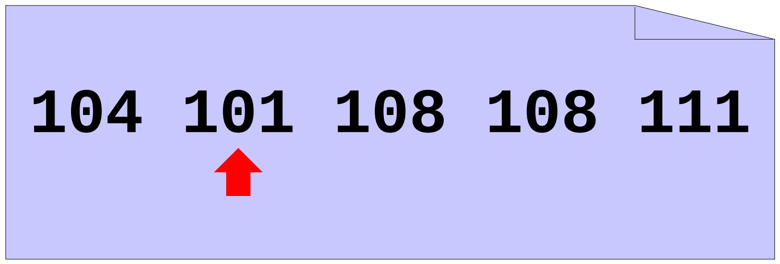


Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a pointer and an operation: `read` or `write`



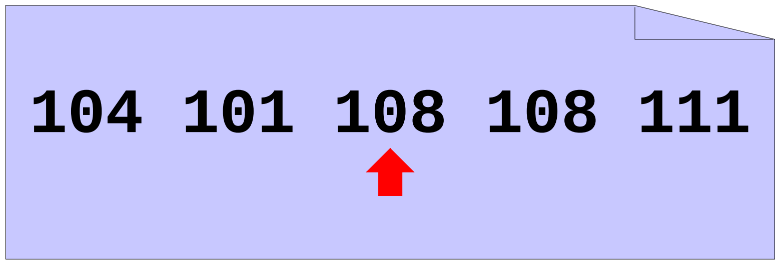
`(read i)` $\rightarrow$ `104`

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

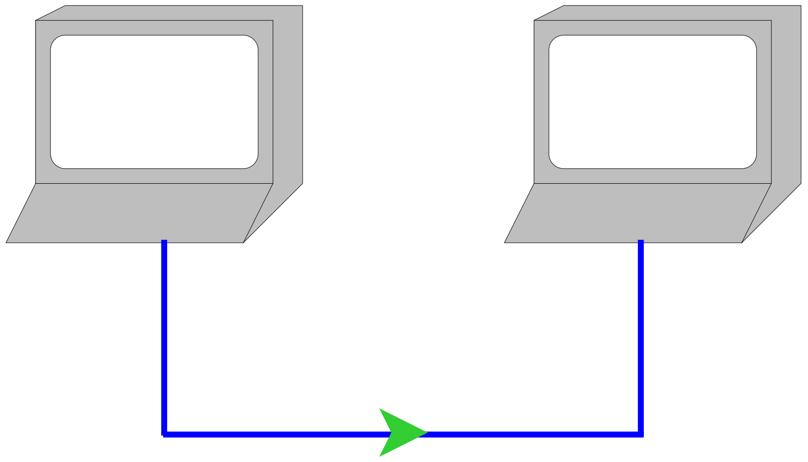
A ***stream*** is a sequence with a pointer and an operation: `read` or `write`



`(read i)` $\rightarrow$ `104`

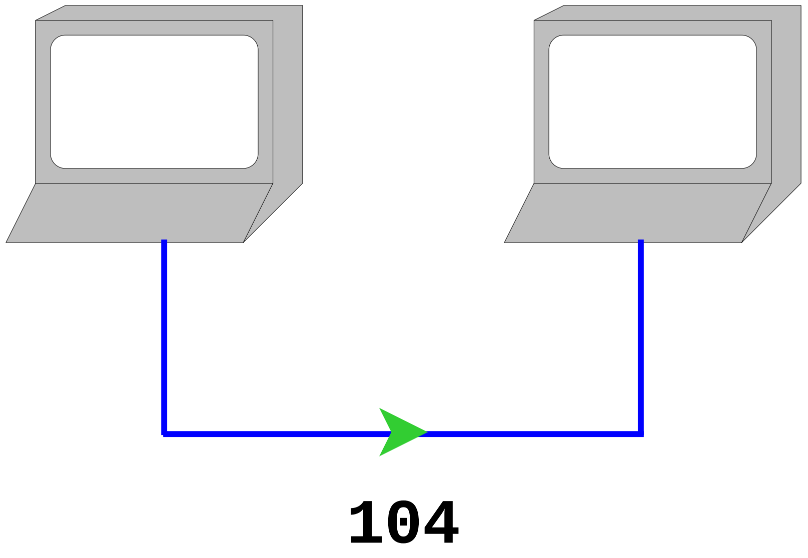
`(read i)` $\rightarrow$ `101`

Byte Streams and Networks



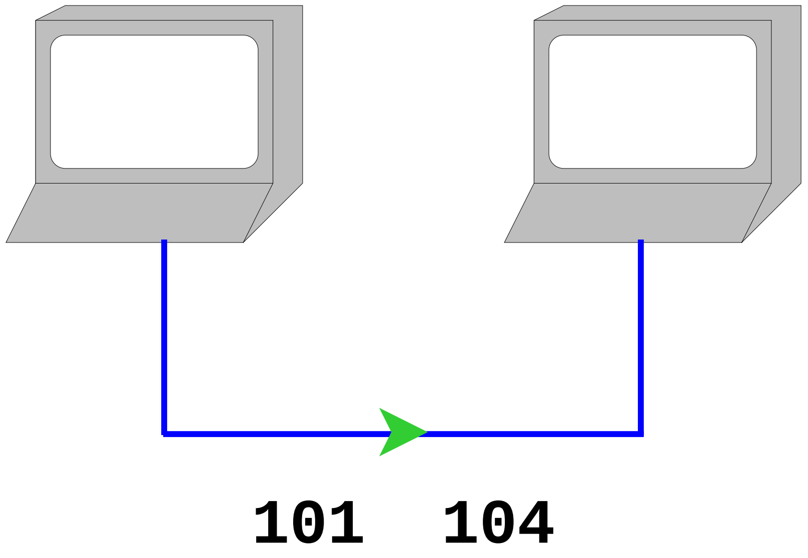
Byte Streams and Networks

```
(write 104 o)
 → (void)
```



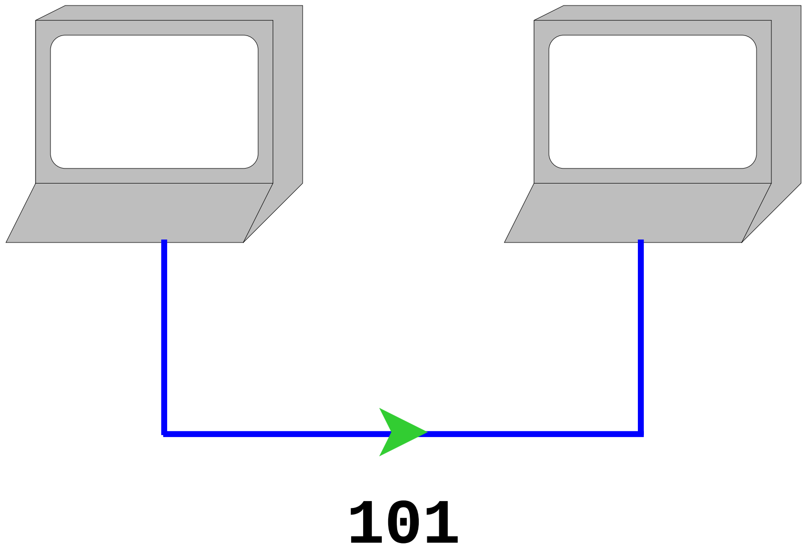
Byte Streams and Networks

```
(write 104 o)
 → (void)

(write 101 o)
 → (void)
```



Byte Streams and Networks

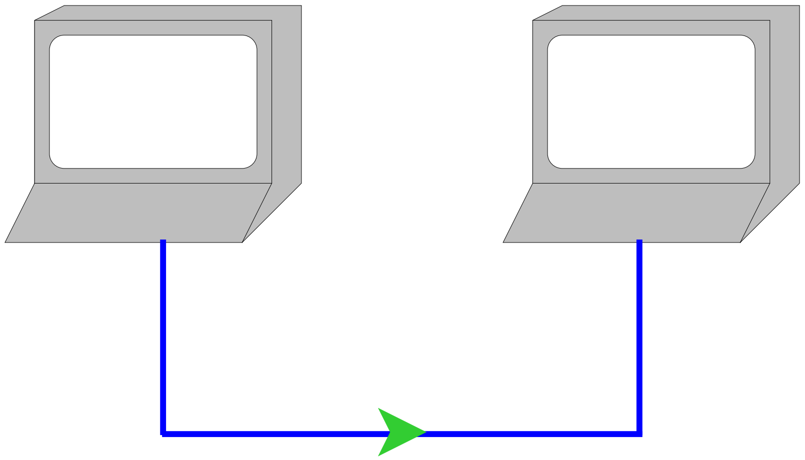
```
(write 104 o)
 → (void)

(write 101 o)
 → (void)
```



```
(read i)
 → 104
```

Byte Streams and Networks

```
(write 104 o)
 → (void)

(write 101 o)
 → (void)
```



```
(read i)
 → 104

(read i)
 → 101
```

Encoding

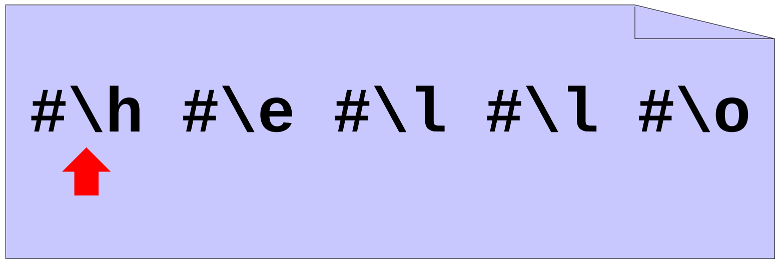
To communicate information other than small numbers, it must be ***encoded***

To encode English text, map each ***character*** to a byte

`#\a`	$\Rightarrow$	`97`
`#\b`	$\Rightarrow$	`98`
`#\c`	$\Rightarrow$	`99`
	...	
`#\A`	$\Rightarrow$	`65`
	...	
`#\(`	$\Rightarrow$	`40`
`#\)`	$\Rightarrow$	`41`
`#\1`	$\Rightarrow$	`48`
	...	

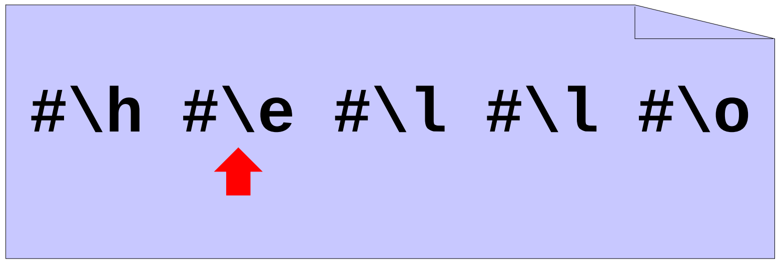
Character Streams

This character encoding is so popular that byte streams are sometimes viewed as ***character streams***



Character Streams

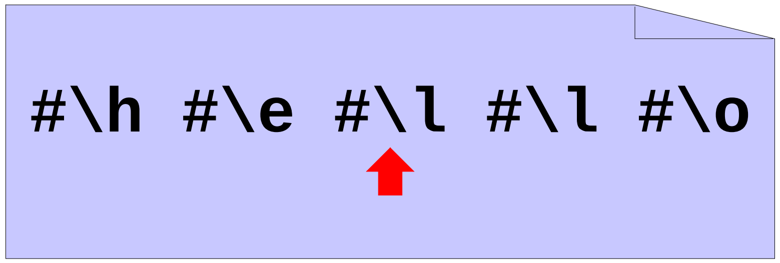
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***



`(read-char i)` $\rightarrow$ `#\h`

Character Streams

This character encoding is so popular that byte streams are sometimes viewed as ***character streams***



`(read-char i)` $\rightarrow$ `#\h`

`(read-char i)` $\rightarrow$ `#\e`

Character Streams in Scheme

```
(define o (open-output-file "ex1"))
(write-char #\h o)
(write-char #\e o)
...
(close-output-port o)

(define i (open-input-file "ex1"))
(read-char i) "should be" #\h
(read-char i) "should be" #\e
...
(close-input-port i)
```

Note: Scheme term for ***stream*** is ***port***

Communicating More Than Characters

`read-char` and **`write-char`** are sufficient for communicating character sequences (or small-number sequences)

To read and write aquariums, we need to communicate lists of (large) numbers

One again, we must encode:

```
empty       ⇒   #\.
'(10000)    ⇒   #\1 #\0 #\0 #\0 #\space #\.
'(1 2)      ⇒   #\1 #\space #\2 #\space #\.
                ...
```

Number List Example

A **`<numlist>`** is

```
    #\.
    <num> #\space <numlist>
```

A **`<num>`** is

```
    <digit>
    <num> <digit>
```

A **`<digit>`** is

```
    #\0
    #\1
    ...
    #\9
```

Number List Writer

```
; write-numlist : list-of-num output-port -> void
(define (write-numlist l p)
  (cond
    [(empty? l) (write-char #\. p)]
    [else (begin
            (write-num (first l) p)
            (write-char #\space p)
            (write-numlist (rest l) p))]))

; write-num : num output-port -> void
(define (write-num n p)
  (cond
    [(< n 10) (write-digit n p)]
    [else (begin
            (write-num (quotient n 10) p)
            (write-digit (remainder n 10) p))]))

; write-digit : num (0-9) output-port -> void
(define (write-digit n p)
  (cond
    [(= n 0) (write-char #\0 p)]
    ...
    [(= n 9) (write-char #\9 p)]))
```

Copy

Number List Example

A **`<numlist>`** is

```
    #\.
    <num> #\space <numlist>
```

A **`<num>`** is

```
    <digit>
    <num> <digit>
```

A **`<digit>`** is

```
    #\0
    #\1
    ...
    #\9
```

Number List Example

Parsing algorithms ⇒ use the following equivalent form:

A **`<numlist>`** is

```
    #\.
    #\0 <num> <numlist>
    ...
    #\9 <num> <numlist>
```

A **`<num>`** is

```
    #\space
    #\0 <num>
    ...
    #\9 <num>
```

Number List Reader

```
; read-numlist : input-port -> list-of-num
(define (read-numlist p)
  (local [(define c (read-char p))]
    (cond
      [(char=? #\. c) empty]
      [(char-digit? c) (cons (read-number p (digit-val c))
                             (read-numlist p))])))

; read-number : input-port num -> num
(define (read-number p n)
  (local [(define c (read-char p))]
    (cond
      [(char=? #\space c) n]
      [(char-digit? c)
       (read-number p (+ (* n 10) (digit-val c)))])))

; char-digit? : char -> bool
...

; digit-val : char -> num
...
```

Copy

read and write

That's the idea, but you usually don't have to start from scratch

- Built into Scheme: **`read`** and **`write`**
  - Like **`read-from-string`**, but handles strings, chars, etc.

- Next time: **`read-xml`** and **`write-xml`**
  - A generalization of HTML

Using read/write libraries means easier encoding

Family Trees

Copy

```
; A family-tree is either
;  - empty
;  - (make-child family-tree family-tree sym)
(define-struct child (father mother name))

(define MY-FAMILY (make-child empty empty 'Matthew))

; add-mother! : sym sym -> void
(define (add-mother! c-name m-name)
  (set! MY-FAMILY (add-mother MY-FAMILY c-name m-name)))

; add-mother : family-tree sym sym -> family-tree
...

; find-relative : sym -> family-tree-or-false
(define (find-relative c-name)
  (find-person MY-FAMILY c-name))

; find-person : family-tree sym -> family-tree-or-false
...
```

Writing Family Trees

```
; family-tree->sexp : family-tree -> sexp
(define (family-tree->sexp ft)
  (cond
    [(empty? ft) '()]
    [else (list (family-tree->sexp (child-father ft))
                (family-tree->sexp (child-mother ft))
                (child-name ft))]))

(family-tree->sexp empty) "should be" '()
(family-tree->sexp (make-child empty empty 'Matthew))
"should be" '(() () Matthew)
(family-tree->sexp
 (make-child (make-child empty empty 'Raymond) empty 'Matthew))
"should be" '((() () Raymond) () Matthew)

; write-family-tree : family-tree output-port -> void
(define (write-family-tree ft p)
  (write (family-tree->sexp ft) p))

(define o (open-output-port "my tree"))
(write-family-tree MY-FAMILY o)
(close-output-port o)
```

Reading Family Trees

```
; sexp->family-tree : sexp -> family-tree
(define (sexp->family-tree sexp)
  (cond
    [(empty? sexp) empty]
    [else (make-child
           (sexp->family-tree (first sexp))
           (sexp->family-tree (second sexp))
           (third sexp))]))

(sexp->family-tree '()) "should be" empty
(sexp->family-tree '(() () Matthew))
"should be" (make-child empty empty 'Matthew)

; read-family-tree : input-port -> family-tree
(define (read-family-tree i)
  (sexp->family-tree (read i)))

(define i (open-input-port "my tree"))
(set! MY-FAMILY (read-family-tree i))
(close-input-port i)
```

Summary

Input/output (or ***I/O*** for short): files, network, and more

- Output **–** choose a representation in terms of an existing writer
- Input **–** parse representation from an existing reader

Base reader/writer (practically all operating systems): bytes

... but there are always better libraries