Scheme topics related to next assignment

- Characters and strings
- Input, output and string ports
- Popular list operations
- Recursion
- Sequencing
- Variable number of arguments

Characters & character sets

- #\a, #\A, #\(
- #\space, #\newline
- char-set:alphanumeric
- char-set:whitespace
- **(char-set-member?** *char-set char*)

Strings

- "abc", "This is a string"
- (string-length "The length")
- (string=? string1 string2)
- (string-ci=? string1 string2) ; case insensitive
- (string-capitalize string) ; capitalize the first letter
- (string-ref string k)
- (substring string start end)
- (string-trim string [char-set])
- (string-append string ...)

String Trimming

- `(string-trim "max12" char-set:numeric)`
- `(string-trim "max12" char-set:alphabetic)`
- `(string-trim-left "max12" char-set:numeric)`
- `(string-trim-left "max12" char-set:alphabetic)`

str-split: https://gist.github.com/matthewp/2324447

```
(define (str-split str ch)
  (let ((len (string-length str)))
    (letrec
      ((split
        (lambda (a b)
          (cond
            ((>= b len) (if (= a b) '() (cons (substring str a b) '())))
            ((char=? ch (string-ref str b)) (if (= a b)
              (split (+ 1 a) (+ 1 b))
              (cons (substring str a b) (split b b))))
            (else (split a (+ 1 b)))))))
        (split 0 0))))
```

my-str-split.scm

```
(define (str-split-helper line str list)
   (cond
      ((string-null? line)
         (if (string-null? str)
            (reverse list)
            (reverse (cons str list))))
      ((char=? (string-ref line 0) #\space)
         (if (string-null? str)
            (str-split-helper (string-tail line 1) str list)
            (str-split-helper (string-tail line 1) "" (cons str list))))
      (else
         (str-split-helper (string-tail line 1)
                           (string-append str (string-head line 1))
                           list))))
(define (str-split line) (str-split-helper line "" '()))
```

Ports

- A *port* serves as a source or sink for data.
- A port must be open before it can be read from or written to.
- (open-input-file filename)
- (close-input-port port)
- (open-output-file filename)
- (close-output-port port)

Read a character

- (read-char [input-port]) *; port is optional*
- *A semicolon (;) starts a comment*
- (peek-char [input-port])
- (eof-object? object) *; check if it reaches eof*
- Read and run the example, `printfile4.scm`, with

```
scheme --quiet < printfile4.scm
```

Read a line

- (read-line [input-port])
- (eof-object? object)
- Read and run the example, `printfile3.scm`, with

```
scheme --quiet < printfile3.scm
```

Read an object

- (read [input-port])
- Read external representation of next Scheme object (number, symbol, list) and return it
- A symbol, different from a string, will be stored in lowercase
- (eof-object? object)
- Read and run the example, `printfile2.scm`, with

```
scheme --quiet < printfile2.scm
```

Read a string

- (read-string char-set [input-port])
- (eof-object? object)
- `read-string` works only on our Scheme (version 9.2), the most recent (version 11.2) uses `read-delimited-string`
- Read and run the example, `printfile.scm`, with

```
scheme --quiet < printfile.scm
```

Output ports

- **(write-char** *output-port char*)
- **(write-string** *output-port string*)
- **(write-substring** *output-port string start end*)
- **(write-line** *output-port string*)
- **(write** *output-port object*)

String ports

- **(open-input-string** *string [start [end]]*)
- First **(open-output-string),** then write, finally **(get-output-string** *port*) to get the combined string
- Similar to file input/output
- Read and run the example, `parsestring.scm`, with

```
scheme --quiet < parsestring.scm
```

Popular list operations

- Selecting

```
(first '("commission" "Floyd" "Jenkin" 300 3000 .08))
(sixth '("commission" "Floyd" "Jenkin" 300 3000 .08))
```

- Filtering

```
(filter odd? '(1 2 3 4 5)) => (1 3 5)
(filter (lambda (x) (< x value)) '(3 9 5 8 2 4 7))
```

- Mapping

```
(map cadr '((a b) (d e) (g h)))
```

- Applying

```
(apply min '(3 9 5 8 2 4 7))
```

Recursion

```
(define (skip-whitespaces port)
  (let ((ch (peek-char port)))
     (if (and (not (eof-object? ch))
(char-whitespace? ch))
         (begin (read-char port)
                (skip-whitespaces port)))))
```

Recursion

- Recursion instead of iteration in a functional language
- A recursion to skip all the white spaces

```
(define (skip-whitespaces port)
  (let ((ch (peek-char port)))
     (if (and (not (eof-object? ch)) (char-
whitespace? ch))
         (begin (read-char port)
                (skip-whitespaces port)))))
```

Sequencing

- (begin expression expression …)
- The *expression*s are evaluated sequentially from left to right

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

Variable number of arguments

- `(define (perform . args) (display (car args))(newline)(display (cdr args)))`
- `(define (perform action . args) (display action)(newline)(display args))`