Racket

CSC 372, Spring 2023
The University of Arizona
William H. Mitchell
whm@cs

"Far better is it to dare mighty things, to win glorious triumphs, even though checkered by failure, than to take rank with those poor spirits who neither enjoy much nor suffer much because they live in the gray twilight that knows neither victory nor defeat."

—Theodore Roosevelt

What they say about Lisp...

Note: Racket is a dialect of Scheme. Scheme is a dialect of Lisp.

"Lisp is worth learning for a different reason—the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot." —Eric Raymond.

"Most people who graduate with CS degrees don't understand the significance of Lisp. Lisp is the most important idea in computer science." —Alan Kay

"the greatest single programming language ever designed"—Alan Kay, on Lisp

"SQL, Lisp, and Haskell are the only programming languages that I've seen where one spends more time thinking than typing."—Philip Greenspun

paulgraham.com/quotes.html has an interesting collection of quotes about Lisp.

Origins of Lisp

"A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit “common sense” in carrying out its instructions."

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I
John McCarthy, April 1960

Initial ideas for Lisp were formulated in 1956-1958 and some were implemented in FLPL (FORTRAN-based List Processing Language).

Lisp is the second-oldest surviving language, after Fortran.

Assorted history

For many years Lisp was *the* language for artificial intelligence research.

"Lisp machines commercially pioneered many now-commonplace technologies, including effective garbage collection, laser printing, windowing systems, computer mice, high-resolution bit-mapped raster graphics, computer graphic rendering, ..."—W

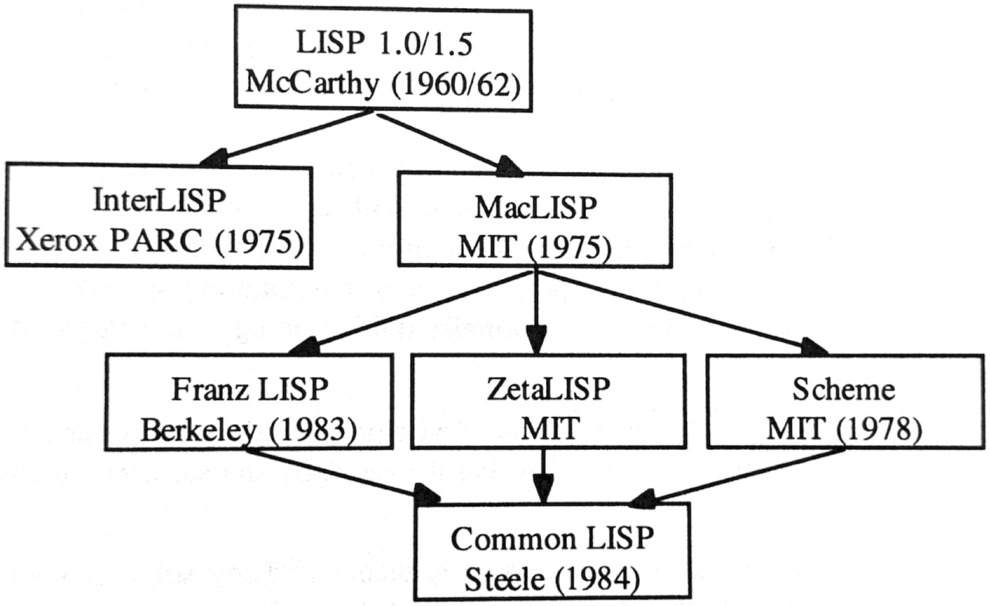
Two of many historically prominent Lisp systems:

> "MYCIN was an early backward chaining expert system that used artificial intelligence to identify bacteria causing severe infections, such as bacteremia and meningitis, and to recommend antibiotics, ..."—W
>
> 'Macsyma "Project MAC's SYmbolic MAnipulator") is one of the oldest general-purpose computer algebra systems still in wide use. It was originally developed from 1968 to 1982 at MIT's Project MAC.'—W

There have been two "AI Winters": 1974–1980 and 1987–1993

Lots of Lisps!

The Wikipedia page on Lisp lists twenty "significant" dialects of Lisp. Jon Pearce's Scheme book has this diagram of ancestry:



The Wikipedia page on GNU Guile has a timeline of Lisp dialects.

Scheme

"Scheme was created during the 1970s at the MIT AI Lab and released by its developers, Guy L. Steele and Gerald Jay Sussman via a series of memos now known as the Lambda Papers."—W

The first Lambda Paper, December 1975:

> **Scheme: An Interpreter For Extended Lambda Calculus**.
>
>> Inspired by ACTORS, we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus, but extended for side effects, multiprocessing, and process synchronization.

Allegedly, they first called it "Schemer", but the ITS operating system limited file names to six-letter components.

Scheme, continued

The *Revised*^{6} *Report on the Algorithmic Language Scheme* [R6RS] says,

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

"The primary Lisp dialects are Common Lisp and Scheme. Scheme and Clojure are from the same family of dialects called lisp-1, and Common Lisp is a lisp-2 dialect." [7L7W]

"Scheme is the UnCommon Lisp."

scheme.org is the Scheme home page

Racket

The Racket Guide says,

Depending on how you look at it, Racket is

- a *programming language*—a dialect of Lisp and a descendant of Scheme;
- a *family* of programming languages—variants of Racket, and more; or
- a set of *tools*—for using a family of programming languages.

Racket was originally called PLT Scheme

"The Racket programming language is the descendant of a long line of languages from the Scheme dialect of Lisp. It's probably the purest contemporary representation of the Lisp family and has amazing community support. Like Haskell, Racket has a relatively small commercial community compared to those who use the language to explore programming language theory."—*Get Programming with Haskell*

racket-lang.org is the Racket home page

Racket Resources

docs.racket-lang.org is Racket documentation and has two primary documents:

- The Racket Guide, docs.racket-lang.org/guide/index.html [RG]
- The Racket Reference, https://docs.racket-lang.org/reference/index.html [RR]

I've learned a lot from these:

The Scheme Programming Language, 4th ed. by R. Kent Dybvig [TSPL]
An Introduction to Scheme by Jerry D. Smith (archive.org)
Teach Yourself Scheme in Fixnum Days by Dorai Sitaram (github.com)
Realm of Racket by Barski, Horn, et al. (learning.orielly.com)
Beautiful Racket by Matthew Butterick [BR]
Good "explainers", like beautifulracket.com/explainer/lang-line.html
Structure and Interpretation of Computer Programs 2nd ed. by Abelson and Sussman
Used in intro CS class at MIT for years. Many study groups of professional programmers have formed to study this book. [SICP]

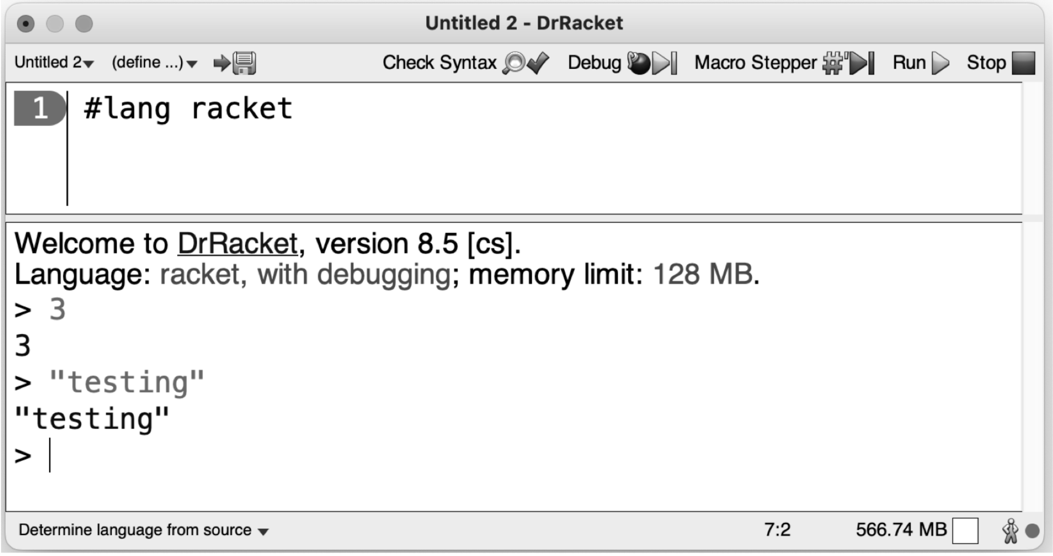
Dr. Collberg really likes *The Little Schemer* by Daniel P. Friedman.
I like it less than he does, but some enjoy its dialog-based approach.

Running Racket

Racket REPLs

One way to bring up a REPL for Racket is to install Racket on your machine and use Dr. Racket there. (download.racket-lang.org)

- Simple IDE with a good editor for Racket code
- Allows use of some graphics procedures
- Be sure to have "**#lang racket**" in the definitions window.
- control-R (cmd-R) to bring up the REPL window
- ESC-p/n to recall previous/next expression



```
#lang racket
```

```
Welcome to DrRacket, version 8.5 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> 3
3
> "testing"
"testing"
> 
```

Racket REPLs, continued

Alternatively, you can run **racket** on the command-line, either on lectura or on your machine, after installing Racket.

- Much like working with **ghci**
- A better REPL (XREPL) than Dr. Racket
- Mostly what I use
- An alias for the Mac: **alias rk="/Applications/Racket\ v8.5/bin/racket"**
- "**it**" is **^**

Here's lectura:

```
% racket
Welcome to Racket v8.5 [cs].
> "hello"
"hello"
> 3.4
3.4
>
```

Working with Racket with VS Code was, and still is, on my TODO list!

Please share what you know/learn about VS Code and Racket on Piazza!

Literals

Let's explore Racket with some expressions that are literals. Here are some numbers:

```
> 7
7

> 3.400
3.4

> 3e3
3000.0

> 25/100          ; No spaces!
1/4

> 3.4-2i
3.4-2.0i          ; Complex number
```

Literals, continued

More:

```
> "abc"
"abc"      ; string

> #\R
#\R        ; character

> 'abc
'abc       ; symbol
```

Literals are considered to be expressions.

Literals

More:

```
> #t
#t              ; boolean

> #F
#f
```

Here's an interesting REPL behavior:

```
> 7 "testing" #\$
7
"testing"
#\$
```

Procedures

Racket has thousands of built-in *procedures* that are bound to identifiers.

```
> string-length
#<procedure:string-length>

> +
#<procedure:+>

> integer->char
#<procedure:integer->char>
```

A curio:

```
> - 3
#<procedure:->
3

> -3
-3
```

More:

```
string-append
zero?
vector-set!
system-language+country
internal-definition-context-splice-binding-identifier
```

Procedure applications

A procedure can be applied to arguments with a *prefix notation*:

```
> (+ 3 4)
7

> (< 3 7)
#t

> (string-length "testing")
7

> (integer->char 65)
#\A

> (string-ci=? "AbC" "aBc")
#t
```

The procedure applications above are expressions, too.

Procedure applications, continued

Many Racket procedures are *variadic*—they can accept any number of arguments:

```
> (+ 3 1 5 7 2)
18

> (= 3 3 5)
#f

> (*)
1

> (> 1/2 1/4 1/8 1/16)
#t

> (string-append "a" "bc" "def")
"abcdef"
```

Procedure applications, continued

Procedure applications can nest:

```
> (+ (* 3 2 4) (/ 5 4.0))
25.25

> (> (+ (* 3 2 4) (/ 5 4.0)) 10)
#t

> (sqrt (+ (* 3 3) (* 4 4)))
5
```

We haven't seen variables yet, but...

```
(play (vector-ref alarm-sounds
           (random (min (vector-length alarm-sounds)
                  (add1 (quotient n 120)))))) ; barzilay.org/misc/alarm
```

A little fun: https://docs.racket-lang.org/oops

No operators!

Effectively, Racket does not have operators in any conventional sense.

Identifiers, some with symbolic names (e.g., + and >=) are bound to procedure values.

Operator precedence and associativity are simply not concepts in Racket's world!

The order of operations is entirely determined by how we nest expressions.

Python:

```
    x*y**3 - 1 / (1.3 * (x - y**z))
```

Racket:

```
    (- (* x (expt y 3)) (/ 1 (* 1.3 (- x (expt y z)))))
```

Is there a gain to compensate for operators being lost?

No superfluousity!

In addition to white space, what characters do most languages let us use superfluously?
Parentheses!

Not so in Racket!

```
> (+ 3 (4))
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 4

> ((+ 3 4))
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 7
```

Extra Credit!

For two assignment points of extra credit:

1. Using either Dr. Racket on your machine, or `racket` on lectura, try ten Racket expressions with some degree of variety, not simply the ones here in the slides.

2. Capture the interaction (both expressions and results) and put it in a plain text file, eca3.txt. No need for your name, NetID, etc. in the file. No need to edit out errors.

3. On lectura, turn in eca3.txt with the following command:

```
% turnin 372-eca3 eca3.txt
```

Due: At the start of the next lecture after the lecture in which this slide is presented.

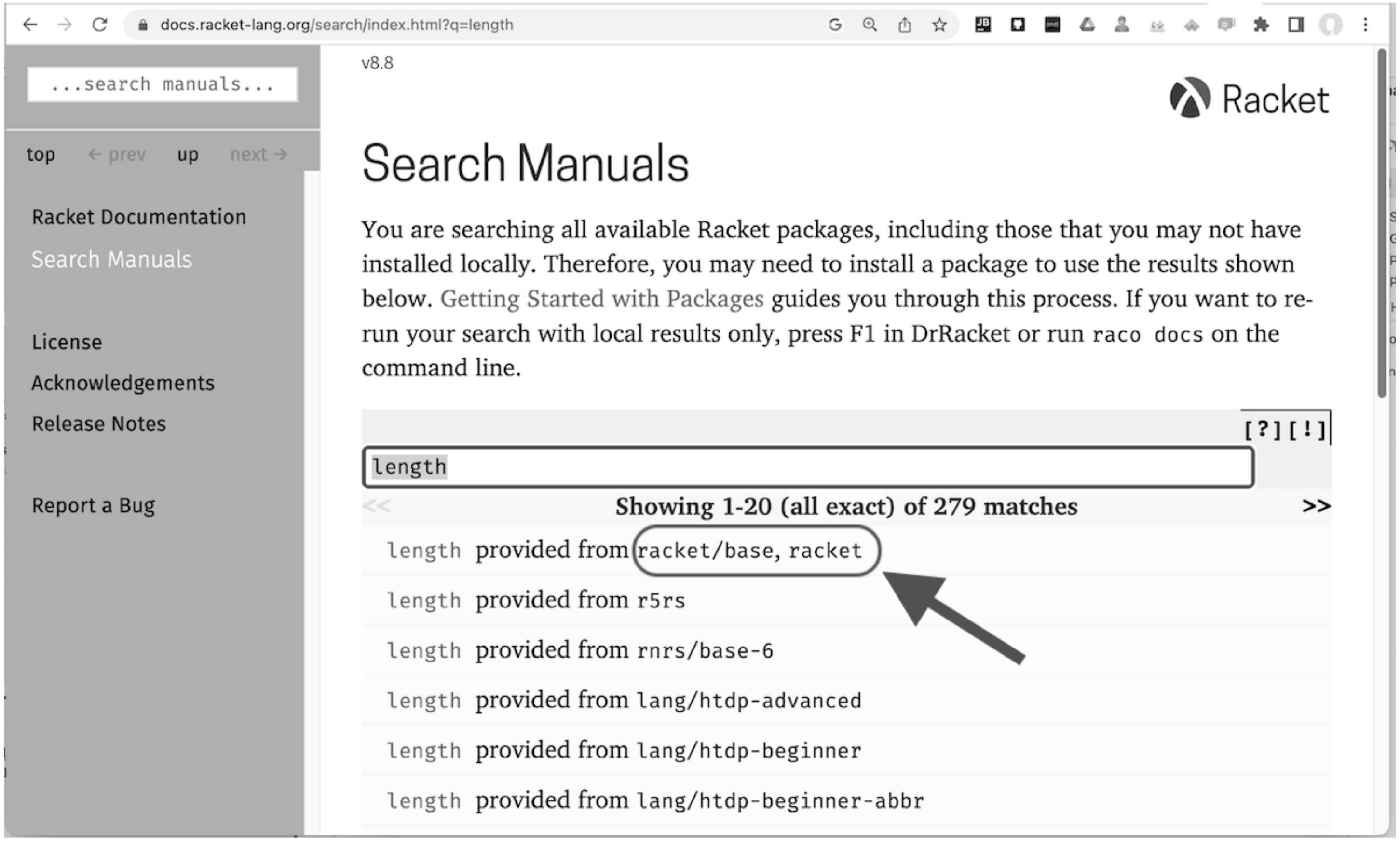
Naming conventions for procedures

What patterns do we see in Racket procedure names?

```
*
<=
char->integer
equal?
hash-clear!
length
let*
procedure?
read-char
string->bytes/utf-8
string->list
string-ref
string?
vector-set!
zero?
```

Searches of Racket docs

Some search results via docs.racket-lang.org/search/index.html:



Search Manuals

You are searching all available Racket packages, including those that you may not have installed locally. Therefore, you may need to install a package to use the results shown below. Getting Started with Packages guides you through this process. If you want to re-run your search with local results only, press F1 in DrRacket or run `raco docs` on the command line.

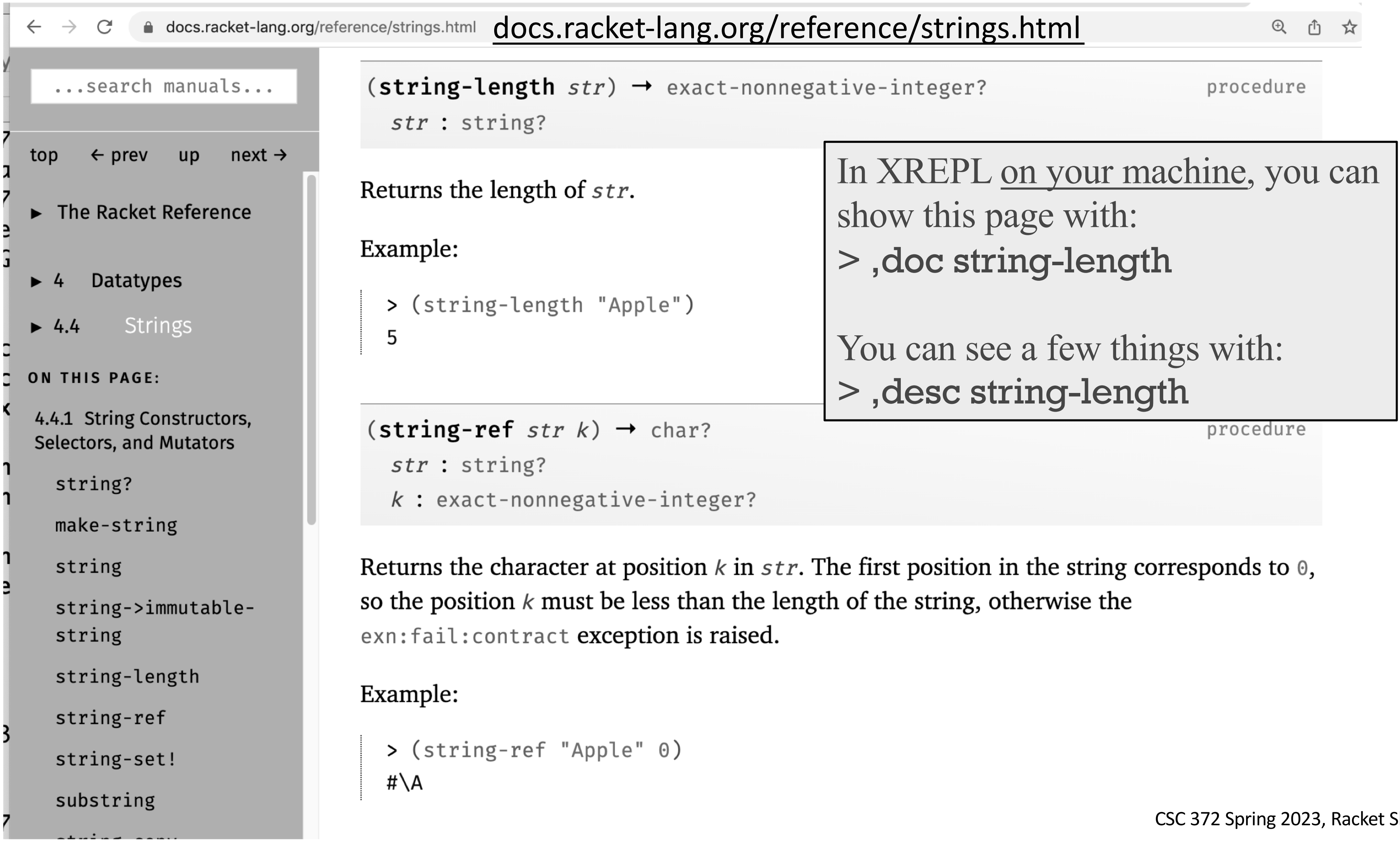
length

<< Showing 1-20 (all exact) of 279 matches >>

```
length provided from racket/base, racket
length provided from r5rs
length provided from rnrs/base-6
length provided from lang/htdp-advanced
length provided from lang/htdp-beginner
length provided from lang/htdp-beginner-abbr
```

Procedure documentation

docs.racket-lang.org/reference/strings.html



```
(string-length str) → exact-nonnegative-integer?        procedure
  str : string?
```

Returns the length of *str*.

Example:

```
> (string-length "Apple")
5
```

```
(string-ref str k) → char?        procedure
  str : string?
  k : exact-nonnegative-integer?
```

Returns the character at position *k* in *str*. The first position in the string corresponds to 0, so the position *k* must be less than the length of the string, otherwise the `exn:fail:contract` exception is raised.

Example:

```
> (string-ref "Apple" 0)
#\A
```

In XREPL on your machine, you can show this page with:

> ,doc string-length

You can see a few things with:

> ,desc string-length

The **rk/fp** script

The directory **spring23/rk** has Racket code and also some Bash scripts.

In your 372 Racket directories (like **csc372/assn7**) make a symlink to **rk**:

```
ln -s /cs/www/classes/cs372/spring23/rk .     # "dot" is optional
```

Once that's done, you can use **rk/fp** ("findproc") to search for procedures whose names contain a specified string:

```
% rk/fp "->list"
...
dict->list
file->list
hash->list
...
```

More:

```
% rk/fp | wc –l
2566

% rk/fp | grep "[A-Z]"
%
```

LHtLaL: **rk/fp** is a Racket program I wrote to help myself explore Racket.

There's also **,ap** (apropos) in XREPL.

More on simple types

Numbers

Numbers in Racket are highly refined. Let's see what **3** is:

```
> (number? 3)
#t
> (integer? 3)
#t
> (rational? 3)
#t
> (real? 3)
#t
> (complex? 3)
#t
> (exact? 3)
#t
```

Racket is said to have a *numerical tower*:

integer?
rational?
real?
complex?
number?

Example: Any value for which **real?** produces **#t** will also get **#t** from **complex?** and **number?**

Numbers, continued

```
> (integer? 1/3)
#f
> (real? 1/3)
#t
> (exact? 1/3)
#t
> (real? 3.4)
#t
> (exact? 3.4)
#f
> (inexact? 3.4)
#t
```

Numbers, continued

More...

```
> (exact? (+ 4 1/5))
#t
> (exact? (+ 4 0.2))
#f
> (exact->inexact 7/3)
2.3333333333333335

> (inexact->exact 0.5)
1/2

> (inexact->exact 0.6)
5404319552844595/9007199254740992

> (< 1/10 0.1)
#t
```

Numbers, continued

A little division:

```
> (/ 3 4)
3/4

> (/ 3 4.0)
0.75

> (quotient 17 5)
3

> (remainder 17 5)
2

> (quotient/remainder 17 5)
3
2
```

(Racket procedures can return multiple values!)

Characters

Racket has a type for individual characters.

```
> (char? #\R)
#t

> #\U1F642
#\🙂     ; character specified with Unicode code point

> (integer->char 38)
#\&

> (char->integer #\ )
32
```

More characters: `#\\ #\" #\space #\tab #\newline #\n`

Characters, continued

Some representative tests and conversions:

```
> (char-upper-case? #\A)
#t

> (char<? #\return #\+ #\7 #\A #\a)
#t

> (char-general-category #\$)
'sc      ; Unicode general category (symbol, currency)

> (char-downcase #\A)
#\a
```

Strings

"A string is a fixed-length array of characters." [RR]

```
 > (string? "testing")
#t

> (string  #\A  #\-  #\U5A)
"A-Z"

> (string->number "1101" 3)
37

 > (make-string 5)
 "\u0000\u0000\u0000\u0000\u0000"

> (make-string 5 (string-ref (string-append "just" "testing") 5))
"eeeee"
```

Strings, continued

A few more procedures:

```
> (substring "abcdefg" 2 5)
"cde"

> (string-contains? "tint" "in")
#t

> (string-suffix? "testing" "ing")
#t

> (string-replace "tethered" "e" "<e>")
"t<e>th<e>r<e>d"
```

Strings, continued

A string can be either mutable or immutable. Various rules and procedures are involved.

```
> (immutable? "abc")
#t

> (immutable? (make-string 7 #\x))
#f

> (immutable? (substring (make-string 7) 3))
#f
```

Speculate: Is there a **mutable?** procedure?

Nope!

Types in Racket

Racket does not provide an analog for Python's **type()** or **ghci**'s **:type**.

Effectively, Racket provides no way to ask "What is X?"
Instead, we are limited to asking "Is X a *whatever*?"

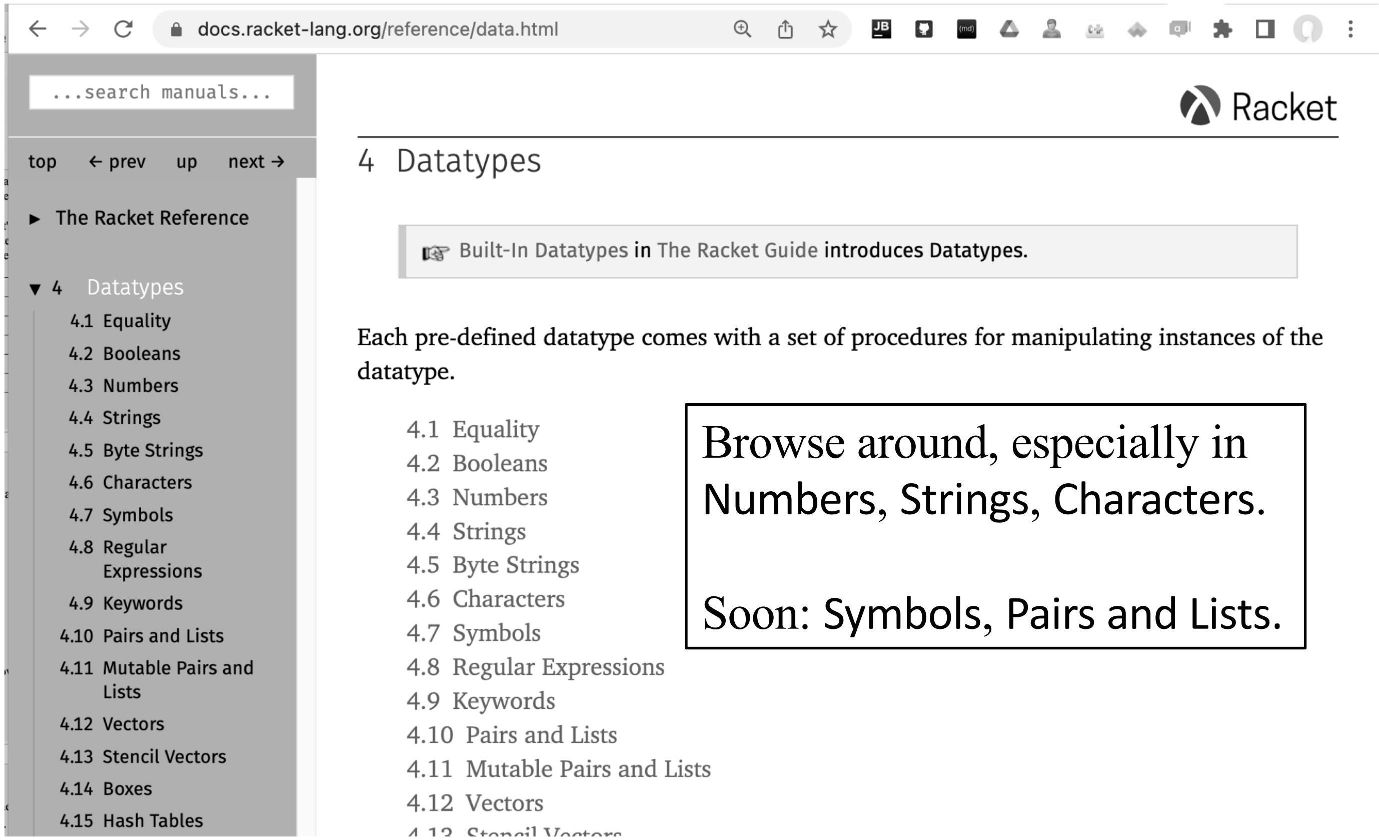
As we've seen...

```
> (string? "testing")
#t
> (number? "23")
#f
> (exact? 10.0)
#f
```

R6RS 11.1 says "No object satisfies more than one of the following predicates: **boolean?, pair?, symbol?, number?, char?, string?, vector?, procedure?, null?**"
(Those predicates *partition* the value/object space.)

Types in Racket, continued

Section 4, Datatypes, in the Racket Reference describes Racket's types and procedures associated with them.



docs.racket-lang.org/reference/data.html

4 Datatypes

Built-In Datatypes in The Racket Guide introduces Datatypes.

Each pre-defined datatype comes with a set of procedures for manipulating instances of the datatype.

4.1 Equality
4.2 Booleans
4.3 Numbers
4.4 Strings
4.5 Byte Strings
4.6 Characters
4.7 Symbols
4.8 Regular Expressions
4.9 Keywords
4.10 Pairs and Lists
4.11 Mutable Pairs and Lists
4.12 Vectors
4.13 Stencil Vectors

Browse around, especially in Numbers, Strings, Characters.

Soon: Symbols, Pairs and Lists.

Equality in Racket

Racket has a complex notion of equality. In general:

- Use `=` for comparison of numbers. **Do not mix** exact and inexact numbers!
- Use `equal?` to compare everything else.
- There are more, like `eq?`, `eqv?`, and lots of *TYPE*`=?` procedures.

Examples with numbers:

```
> (= (+ 3 4) 7 (sub1 8))
#t

> (= 1/4 0.25)
#t

> (= 1/10 0.1)
#f
```

Equality in Racket, continued

equal? on a few things:

```
> (equal? "upper" (string-append "up" "per"))
#t

> (equal? (string-ref "abc" 1) #\b)
#t

> (equal? "a" #\a 'a)
equal?: arity mismatch;...
```

equal? does deep comparison of lists and more, like == in Haskell and Python.

Lots more equalities:

```
$ echo $(rk/fp | grep "[a-z]=?")
arity=? boolean=? bound-identifier=? bytes=? char-ci=? char=? free-identifier=?
free-label-identifier=? free-template-identifier=? free-transformer-identifier=?
member-name-key=? object-or-false=? object=? parameter-procedure=? set=?
string-ci=? string-locale-ci=? string-locale=? string=? symbol=?
```

A little i/o

A little output

The procedure **displayln** prints a value followed by a newline:

```
> (displayln "testing")
testing
>
```

display prints without adding a newline:

```
> (display 3) (display 4) (display 5)
345
>
```

newline outputs a newline

```
> (display "hello") (newline) (display "world")
hello
world
>
```

A little output, continued

The **printf** procedure interpolates its arguments into a format string:

```
> (printf "~a + ~a is ~a\n" 3 4 (+ 3 4))
3 + 4 is 7
```

The **~a** *escape* directs that the corresponding value be output using **display**.

What's the difference between Racket's **printf** and **format**?

```
> (format "~a + ~a is ~a\n" 3 4 (+ 3 4))
"3 + 4 is 7\n"
```

There are many other escapes (see **fprintf**) but **~a** is mostly all that we'll need.

~~docs.racket-lang.org/srfi/srfi-std/srfi-48.html provides for more elaborate formatting, including field widths, fixed point precision, and more.~~

There's also a family of procedures (**~a**, **~v**, and more) for converting values to strings. [RR 4.4.7]

Input

read-line and **read-char** read from standard input

```
> (read-line)
testing
"testing"

> (read-char) (read-char) (read-line)
mudge
#\m
#\u
"dge"
```

Input, continued

The name **eof** is bound to "a value (distinct from all other values) that represents an end-of-file." [RR]

```
> eof
#<eof>

> (read-line)
^D  (control-D, not echoed)
#<eof>

> (define line (read-line))
^D
> line
#<eof>
> (equal? line eof)
#t
```

Variables

Forms

Most generally we'll say that the following are all *forms*:

```
7
"abc"
#\c
+
length
```

The syntax of a procedure application is simply this:

(*form1 form2 ... formN*)

It, too, is a form. Forms that produce a value are considered to be expressions.

Forms inside a form are often called *subforms*. The following form has three subforms, each of which has subforms itself:

```
((identity +) (* 3 (- 5 a)) (string->number "20"))
```

define

One way to create a variable is with **define**:

```
> (define i 3)
> (define s "Racket")
> (string-ref s i)
#\k
```

What's a little odd about **define**?

Above, the forms **i** and **s** are not evaluated by **define**!

define is not an identifier with a procedure as its value!

define is a *syntactic keyword*. **(define ...)** is said to be a *special form*.

Special forms can handle/treat their subforms in any way they want.

The **define**s above treat **i** and **s** not as expressions but as variable names.

define, continued

At the REPL prompt, **(define x ...)** creates a *top-level* variable.

We can change the value of a variable with the **set!** special form:

```
> (define x 5)
> (set! x (* x 3))
> x
15
```

Speculate: Can we create a variable with **set!** ?

```
> (set! y (+ x 1))
set!: assignment disallowed;
 cannot set variable before its definition
  variable: y
```

Sidebar: identifiers

It's simplest to say what's *not* valid for Racket identifiers:

- The characters `( ) [ ] { } " , ' ` ; # | \` can't appear in an identifier.
- A sequence of characters that's a *valid* numeric literal can't be an identifier.

All other sequences of non-whitespace characters are valid identifiers! Examples:

```
a-b+c*d
3..
2.3.4
3.4e5.3
>10&&<x
is-D>C++?
🙂+   ->😐
```

We've traded away operators but we have gained in identifier naming.

How do we feel about that transaction?

A simple program

Here's a file that contains a whole program:

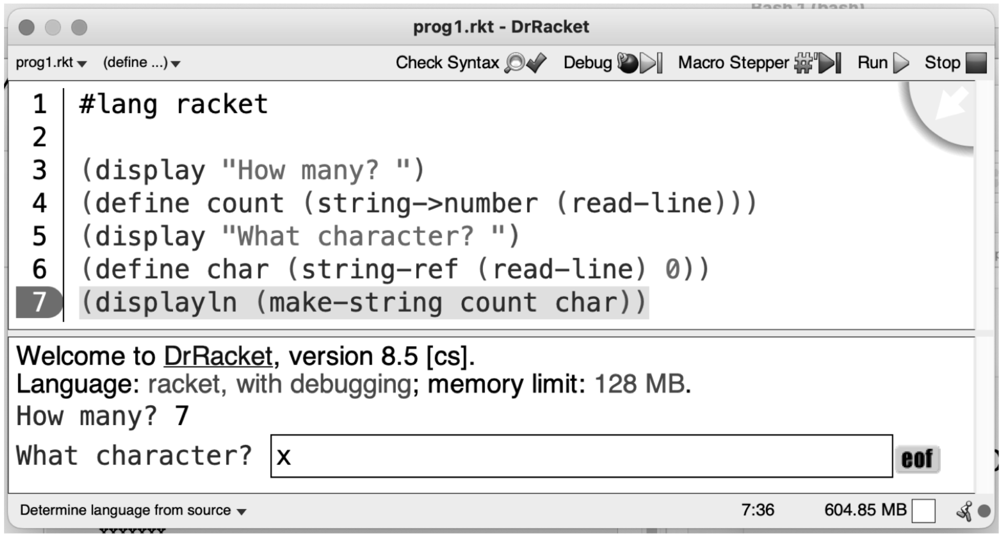
```
% cat rk/howmany.rkt          # assumes rk symlink made on slide 27
#lang racket ; must be first non-comment line

(displayln "How many? ")
(define count (string->number (read-line)))
(displayln "What character? ")
(define char (string-ref (read-line) 0))
(displayln (make-string count char))
```

Let's run it on lectura:

```
% racket rk/howmany.rkt
How many? 7
What character? x
xxxxxxx
```

A simple program, continued

After copying **rk/howmany.rkt** from lectura to a file on my machine, I can open it with Dr. Racket (File>Open... or click...) and run it with ctrl-R or cmd-R:



```
#lang racket

(display "How many? ")
(define count (string->number (read-line)))
(display "What character? ")
(define char (string-ref (read-line) 0))
(displayln (make-string count char))
```

```
Welcome to DrRacket, version 8.5 [cs].
Language: racket, with debugging; memory limit: 128 MB.
How many? 7
What character? x
```

Notice the **eof** button!

Dr. Racket shows lots of "informative" arrows by default. You can turn them off with (File>Edit>)Preferences>Background Expansion tab, deselect "Show binding and tail-position arrows..." and then close/open any source files to make it take effect.

Don't forget #lang racket

At hand:

```
% cat rk/howmany.rkt
#lang racket

(display "How many? ")
(define count (string->number (read-line)))
...
```

If we forget the "hash-lang" line (#lang racket), we'll see something like this:

```
% racket rk/howmany.rkt
default-load-handler: expected a `module' declaration, but found
something else
```

Experiment!

What does the following experiment tell us?

```
% cat rk/topexprs.rkt
#lang racket
3
"testing"
(displayln "Here's a sum...")
(+ 5 1 3)

% racket rk/topexprs.rkt
3
"testing"
Here's a sum...
9
```

Defining procedures

Defining procedures

We can also use **define** to create procedures:

```
> (define (double x)
    (printf "doubling ~a\n" x)
    (* x 2))

> double
#<procedure:double>

> (double 7)
doubling 7
14

> (double (double 7))
doubling 7
doubling 14
28
```

Defining procedures, continued

Here's the syntax of **define** for creating a procedure:

```
(define (name param1 ... paramN)
    form1
    ...
    formN)
```

Each form is evaluated in turn. The return value is the value of the last form.

Is the following valid?

```
(define (f) 1 2 3)
```

How does procedure definition in Racket compare to Haskell? Python?

What aspect of the imperative paradigm do Racket procedure definitions embody?

Which?

At hand:

```
(define (double x) (printf "doubling ~a\n" x) (* x 2))
(define (f) 1 2 3)

> f
#<procedure:f>

> double
#<procedure:double>

> ,desc double
; `double' is a bound identifier,
;   defined in /Users/whm/372/rw/define1.rkt
...
```

Does Racket appear to be statically or dynamically typed?

Practice!

Problem:

Write a procedure **first-last-same?** that tests whether the first and last characters of a string are the same. Assume the string is not empty.

```
> (first-last-same? "testing")
#f

> (first-last-same? "giggling")
#t
```

Handy:

(string-ref *s index*)
(string-length *s*)
(equal? *a b*)

Practice, continued

Desired:

```
> (first-last-same? "testing")
#f
> (first-last-same? "giggling")
#t
```

You can (often) find code from the slides by doing, for example,

```
% grep first-last rk/*.rkt
```

Solution:

```
(define (first-last-same? s)
   (equal? (string-ref s 0)
               (string-ref s (sub1 (string-length s)))))
```

Python:

```
def first_last_same(s):
     return s[0] == s[-1]
```

Java:

```
static boolean firstLastSame(String s)
{
   return s.charAt(0) == s.charAt(s.length()-1);
}
```

More practice

Problem: Write a procedure **dollars** that returns how many dollars some number of pennies, nickels, and dimes represents.

Usage:

```
> (dollars 1 2 3)
0.41

> (dollars 7 20 0)
1.07
```

Solution:

```
(define (dollars pennies nickels dimes)
   (/ (+ pennies (* 5 nickels) (* 10 dimes)) 100.0))
```

First version of **dollars**

My first version of **dollars**:

```
(define (dollars pennies nickels dimes)
   (/ (+ pennies (* 5 nickels) (* 10 dimes))) 100.0)
```

Results:

```
> (dollars 1 2 3)
100.0

> (dollars 5 10 20)
100.0
```

What's wrong?

Misplaced parentheses!

The body has only two forms: `(/ (+ ...))` and `100.0`

What's different?

```
% cat rk/define2.py
def f(xval, str):
    return xval * 3 + length(s)

print("hi!")

% python define2.py
hi!
```

```
% cat rk/define2.rkt
#lang racket
(define (f xval str)
    (+ (* xval 3) (string-len s)))

(displayln "hi!")

% racket  rk/define2.rkt
rk/define2.rkt:3:19: string-len: unbound
identifier
  in: string-len
  location...:
   rk/define2.rkt:3:19
```

The **let** special form

Imagine a procedure that puts a box of asterisks around a string:

```
> (boxstr "Testing!")
************
* Testing! *
************
```

Here's an almost-fine version:

```
(define (boxstr s)
   (define bar (make-string (+ 4 (string-length s)) #\*))
   (displayln bar)
   (displayln (string-append "* " s " *"))
   (displayln bar))
```

let, continued

A better practice is to use **let**, another special form, to creating a binding for **bar**:

```
(define (boxstr s)
   (let ((bar (make-string (+ 4 (string-length s)) #\*)))
      (displayln bar)
      (displayln (string-append "* " s " *"))
      (displayln bar)))
```

Here's the syntax of **let**:

```
(let ((var1 init-expr1) (var2 init-expr2) ... (varN init-exprN))
     (body-expr1)
     ...
     (body-exprN))
```

- Each ***varN*** is created and assigned the value of ***init-exprN***.
- Each ***body-expr*** is evaluated in turn.
- The scope of the variables is the **body-expr**s.

let, continued

A simple example:

```
> (let ((x 2) (y 3) (z (+ 4 5))) (+ x (* y z)))
29
```

What does the above demonstrate about **let**?

The value of a **let** is the value of the last expression.

What's the shortest valid **let**?

A **let** example from SICP, p.64:

```
(define (f x y)
   (let ((a (+ 1 (* x y)))
           (b (- 1 y)))
          (+ (* x (* a a))
              (* y b)
              (* a b))))
```

let, continued

What's wrong here?

```
> (let ((a 5) (b (* a a)) (c (+ a b))) (+ a b c))
a: undefined;
 cannot reference an identifier before its definition
```

The scope of variables in a **let** is only the body forms!

let* is a slight variant of **let** that allows use of earlier bindings in later bindings:

```
> (let* ((a 5) (b (* a a)) (c (+ a b))) (+ a b c))
60
```

Challenge: Rewrite the above using nested **let**s instead of **let***.

In general, an asterisk suffix indicates a variant of a procedure or special form.

Sidebar: A readability aid

Racket allows both [...] and { ... } to be used in place of (...):

```
> {+ [* 3 4] 5}
17
```

Which of the following two do you like better?

```
(let [(x 2) (y 3) (z (+ 4 5))] (+ x (* y z)))
```

or

```
(let ([x 2] [y 3] [z (+ 4 5)]) (+ x (* y z)))
```

My understanding is that the latter is preferred because it accentuates that `[x 2]` et al. are not expressions.

You are free to use whatever style you want for code in this class!

Comments

In Racket, a semicolon is comment to end of line, like // in Java and # in Python:

```
(define max-delta 0.001) ; whatever max-delta represents
```

#| and |# are block comments, just like /* ... */ in Java

There's also a *datum comment.* What does it seem to do?

```
> (+ 1 2 #; 3 4)
7

> (define max #; 10000 10)
> max
10

> (+ 1 2 #; (* 3 4) 5)
8
```

It comments the next *form*, whatever it is, no matter how long/how many lines.

if and more

The **if** special form

Here's the syntax of **if**:

(if *test-expr true-expr false-expr*)

Example:

```
> (if (< 3 4) "less" "greater")
"less"
```

What does the above tell us about **if**?

It produces a value that we can use in an enclosing expression.

Note the error that this common mistake produces:

```
> (if (3 < 4) "less" "greater")
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 3
```

Everything except **#f** is true

Java's **if** statement requires the control expression to have type **boolean**.

```
if (i > 4) ...
if (checkStr(s)) ...
if (verboseMode) ...
```

Racket's **if** allows an expression of any type to be used as the control expression. Any value is that is not **#f** is considered to be true.

Examples:

```
> (if 0 "true" "false")
"true"
> (if "false" "true" "false")
"true"
> (if 5 > 3)
#<procedure:>>
```

Problem: **->boolean**

Racket doesn't seem to have any counterpart to Python's **bool(x)**.

Problem: Using **if**, write **->boolean**. Examples:

```
> (->boolean 5)
#t

> (->boolean "")
#t

> (->boolean #f)
#f
```

Solution:

```
(define (->boolean x)
   (if x #t #f))
```

if, continued

Does **if** have to be a special form?

Consider this procedure:

```
(define (print-average sum n)
   (if (> n 0)
      (printf "average: ~a\n" (/ sum n))
      (printf "No items\n")))
```

For any case, we only want one of those **printf**s to be evaluated!

The implementation of **if** must first evaluate the test **(> n 0)** to decide which of the two clauses to evaluate.

if, continued

Problem: Make the following a little less repetitive:

```
(if (< a b)
    (+ a b)
    (- a b))
```

Solution:

```
((if (< a b) + -) a b)
```

Questions:

Which do you like better?

What does this tell us about Racket?

Credit: Aaron Christianson on Quora

Example

Let's use much of what we've seen to write this procedure:

```
> (print-pattern 4 #\*)
************
*********
******
***
*
***
******
*********
************
```

Note the pattern for the line lengths: `12, 9, 6, 3, 1, 3, 6, 9, 12`

It will be recursive! Don't peek at the next slide!

Example, continued

Solution:

```
(define (print-pattern n c)
   (let ([line (make-string (* n 3) c)])
      (displayln line)
      (if (> n 1)
         (print-pattern (sub1 n) c)
         (displayln c))
      (displayln line)))
```

if, continued

What does the following **if** do?

```
(if (< a b) 7 (displayln "calling f") (f a b))
```

```
if: bad syntax
```

(f a b) is not expected! **if** expects exactly three forms.

We can use **(begin *form1* ... *formN*)** to group forms:

```
(if (< a b)
   7
   (begin
      (displayln "calling f")
      (f a b)))
```

The value of the last form is the value of the **begin** form.

and and or

and and or in Racket are variadic:

```
> (and (< 3 4) (zero? (- 3 3)) (string-ci=? "ok" "OK"))
#t

> (or (> 1 2) (= 3 4) (zero? 1) (zero? 0))
#t

> (or (< 3 4))
#t

> (and 1 2 3 4)
4

> (or 1 2 3 4)
1
```

and and **or**, continued

Two more examples:

```
> (and (= 2 3) (= 4 (/ 1 0)))
#f

> (or (= 3 3) (= 0 (/ 1 0)))
#t
```

What's interesting about those examples?

Neither produced a division by zero error!

and and **or** are special forms, and *use short-circuit evaluation.*

not

Here's **not**:

```
> (not #t)
#f

> (not (= 3 3))
#f

> (not (not 1))
#t
```

Is **not** a special form?

Can we write a simpler version of **->boolean** using **not**?

```
(define (->boolean x)
   (not (not x)))
```

leap-year?

A year is a leap year if it is divisible by 4 and not by 100, or if it is divisible by 400.

Here's my first version of a leap year predicate:

```
(define (leap-year? year)
  (or
    (and (= (modulo year 4) 0)
         (not (= (modulo year 100))))
    (= (modulo year 400) 0)))
```

Execution:

```
> (leap-year? 2020)
#f
> (leap-year? 2021)
#f
> (leap-year? 2000)
#t
```

What's the bug?

Curiosities

Here are two interesting examples from *Realm of Racket*. What does each do?

```
(or (odd? x) (set! is-it-even #t))

(and file-modified (ask-user-about-saving) (save-file))
```

Some "real" code

Here's some slightly simplified Scheme code from a structure editor by Dybvig. (**edit.ss**)

```
(define (f p i b)
   (let ([e (list-ref p i)])
      (if (atom? e)
         (let next ([p p] [i i] [b b]) ; this is a "named let"
            (let ([n (maxref p)])
               (if (or (not n) (< i n))
                   (check p (+ i 1) b)
                   (if (null? b)
                       (search-failed s0 p0 i0 b0)
                       (apply next b)))))
         (check e 0 (list p i b))))))
```

Note the multiple **let**s, and their nesting. Can any **let**s be combined?

Racket doesn't have **atom?**, which tests for a non-null *pair*. We'll learn about pairs and **null?** later.

cond

The original, and more general, conditional special form is **cond**.

Syntax:

```
    (cond
        [test-form1 form ...]
        [test-form2 form...]
        ...
        [else form ...]) ; Literally "else"
```

Each ***test-formN*** is evaluated in turn until one is true.

- The form**(s)** that follow that **test-form** are then evaluated.
- The value of the **cond** is the value of the last form.

There can be as little as one clause.

cond, continued

Example:

```
(define (f x)
   (cond
      [(number? x) (displayln "number") (* x 3)]
      [(string? x) (displayln "string") (string-length x)]
      [else #f]))
```

Usage:

```
> (f "test")
string
4
> (f 5)
number
15
> (f f)
#f
```

Practice

Problem: Write a procedure to determine the sign of a number.

```
> (sign 7)
1
> (sign -3/4)
-1
> (sign (+))
0
```

Solution:

```
(define (sign n)
   (cond
      [(< n 0) -1]
      [(zero? n) 0]
      [else 1]))
```

Exploration

What is **else**?

```
> else
else: not allowed as an expression
```

How about these?

```
> (cond [0 1])
1

> (cond [0])
0

> (cond 0)
cond: bad syntax (clause is not a test-value pair)
```

Intentionally blank...

The **for** special form

The **for** special form

Racket's **for** special form can be used to iterate through a *sequence*. A string is one of many things considered to be a sequence.

```
> (for ([c "abc"])
     (printf "~a is ~a" c (char->integer c))
     (newline))
a is 97
b is 98
c is 99
```

What can we observe about **for**?

for, continued

in-range is much like Python's **range**, and produces a *sequence*. Let's use it in an imperative factorial computation:

```
(define (f n)
   (let ([product 1])
      (for ([i (in-range 2 n)])
         (set! product (* product i)))
      (string-length (number->string product))))
```

Experimentation:

```
> (f 6)
3
> (f 1000)
2565
> (time (f 50000))
cpu time: 982 real time: 1007 gc time: 89
213232
```

for, continued

What's interesting about the following example?

```
> (for ([1s (in-range 1 30)]
        [3s (in-range 1 30 2)]
        [7s (in-range 1 30 7)])
    (printf "~a  ~a  ~a\n" 1s 3s 7s))
1  1  1
2  3  8
3  5  15
4  7  22
5  9  29
```

for can iterate through multiple sequences in parallel, stopping when one runs out!

for, continued

in-lines returns a sequence of the lines on standard input.

Here's a Racket program that numbers lines on standard input:

```
% cat numlines.rkt
#lang racket
(for ([line-num (in-naturals 1)] [line (in-lines)])
   (printf "~a: ~a\n" line-num line))
```

Let's feed **numlines.rkt** to itself, using the command line:

```
% racket numlines.rkt < numlines.rkt
1: #lang racket
2: (for ([line-num (in-naturals 1)] [line (in-lines)])
3:    (printf "~a: ~a\n" line-num line))
```

docs.racket-lang.org/reference/sequences.html talks about sequences.

for, continued

Of course, **for**s can nest:

```
> (for ([row "abcd"])
       (for ([seat (in-range 1 7)])
           (printf "~a-~a  " row seat))
       (newline))
a-1  a-2  a-3  a-4  a-5  a-6
b-1  b-2  b-3  b-4  b-5  b-6
c-1  c-2  c-3  c-4  c-5  c-6
d-1  d-2  d-3  d-4  d-5  d-6
```

Racket has 50+ variants of **for**. Some of them:

for, for/first, for/foldr, for/foldr/derived, for/hash, for/last, for/list, for/lists, for/or, for/product, for/sum, for/vector, for/weak-set, for*,
(and **for*/...** counterparts).

See them in Bash: **rk/fp | grep -w ^for**, or with **,ap for** in **XREPL**

wc.rkt

wc(1) reports the number of lines, "words", and characters on standard input:

```
% wc < rk/numlines.rkt
 3  13 104
```

Let's write a simple, imperative **wc** in Racket!

```
(define (wc)
   (let ([lines 0] [words 0] [chars 0])
      (for ([line (in-lines)])
         (set! lines (add1 lines))
         (set! words (+ words (length (string-split line))))
         (set! chars (+ 1 chars (string-length line))))
      (printf "~a ~a ~a\n" lines words chars)))
(wc)
```

Usage:

```
% racket rk/wc.rkt < rk/numlines.rkt
3 13 104
```

Will **wc.rkt** work in Dr. Racket?

There's also a "**do**", slide 154+/-.

Symbols

Symbols

We can create a *symbol* by putting an apostrophe immediately before any valid identifier.

```
> 'abc
'abc

> (symbol? 'abc)
#t

> (define x '<==oOo!)
> x
'<==oOo!

> (symbol? x)
#t
```

We pronounce `'x` as "quote x".

Just like the double-quotes in `"abc"` are not part of the string, the apostrophe in `'abc` is not part of the symbol!

Symbols, continued

A string is not a symbol; a symbol is not a string!

```
> (symbol? "abc")
#f

> (string? 'abc)
#f
```

But we can convert between symbols and strings:

```
> (string->symbol "abc")
'abc

> (symbol->string 'green)
"green"
```

Symbols, continued

Various procedures use symbols a bit like **Enum**s in Java:

```
> (system-type 'os)
'macosx
> (system-type 'arch)
'x86_64
```

open-output-file recognizes '**append**, '**replace**, and '**error** and others.

find-system-path recognizes '**pref-file, 'init-dir, 'host-config-dir** and others.

Symbols are often used to represent members of small sets:

```
'red  'green  'blue
'up  'down  'left  'right
'A  'K  'Q  'J
```

Unlike **Enum**s in Java, symbols are subject to misspellings.

Symbols, continued

"The symbols are atomic in the sense that any substructure they may have as sequences of characters is ignored." [JMC1]

"A *symbol* is like an immutable string, but symbols are normally *interned* so that two symbols with the same character content are normally `eq?`." [RR]

```
> (eq? 'test (string->symbol "test"))
#t

> (eq? "upper" (string-append "up" "per"))
#f

> (eq? "test" (symbol->string 'test))
#f
```

An essential quality is that comparing two symbols is an *O(1)* operation.

Sidebar: quote

In the early days of Lisp, symbols were created with the **quote** special form.

It still exists:

```
> (quote abc)
'abc

> (symbol? (quote abc))
#t

> (quote quote)
'quote

> (quote 3)
3

> (quote "x")
"x"
```

Lists

List basics

Here's one way to make a list in Racket:

```
> (define L (cons 10 (cons 20 (cons 30 empty))))
> L
'(10 20 30)        ; shown as a "quoted list"
```

car and **cdr** ("could-er"), respectively, get the head and tail of a list:

```
> (car L)
10
> (cdr L)
'(20 30)
```

There are a number of *list accessor shorthands*:

```
> (cadr L)         ; (car (cdr L))
20
> (caddr L)        ; (car (cdr (cdr L)))
30
```

List basics, continued

More:

```
> (define L (cons 10 (cons 20 (cons 30 empty))))
> L
'(10 20 30)

> (list? L)
#t

> (length L)
3

> (last L)
30

> (empty? L)
#f
```

```
> empty
'()

> (empty? empty)
#t

> (cdddr L)
'()

> (empty? (cdddr L))
#t

> (empty? (cdr (cons 5 empty)))
#t
```

List basics, continued

A quoted list is commonly used to make a list from literals:

```
> (define L '(10 a "b" 30 more...))
> L
'(10 a "b" 30 more...)

> (cadr L)
'a                    ; symbol!

> (third L) ; for the caddr-challenged
"b"

> (last L)
'more...

> (symbol->string ^)       ; caret is last value, like "it" in ghci.  (Only in XREPL.)
"more..."
```

We'll frequently use quoted lists to make a list to experiment with:

```
> (define words '(just a test here))
> words
'(just a test here)

> (define nums '(3 1 5 7 2))
> nums
'(3 1 5 7 2)
```

Sidebar: **`car`** and **`cdr`**

The names **`car`** and **`cdr`** are said to have originated with the initial Lisp implementation, on an IBM 7090.

- "CAR" stood for Contents of Address part of Register.
- "CDR" stood for Contents of Decrement part of Register.
- There's more to the story...

Intentionally blank

Problem: len

Let's write our own version of the **length** procedure:

```
> (len '(3 1 5 7))
4

> (len empty)
0
```

Solution:

```
(define (len L)
   (if (empty? L)
      0
      (add1 (len (cdr L)))))
```

Problem: **from-through**

Problem: Write a procedure **(from-through first last)** that returns a list of the integers from **first** through **last**.

```
> (from-through -5 5)
'(-5 -4 -3 -2 -1 0 1 2 3 4 5)

> (from-through 5 1)
'()
```

Solution, in **rk/lists.rkt**:

```
(define (from-through first last)
   (if (> first last)
      empty
      (cons first (from-through (add1 first) last))))
```

from-through, continued

Let's write a **char-from-through**:

```
> (char-from-through #\a #\e)
'(#\a #\b #\c #\d #\e)
```

Sadly, we haven't seen **map** yet, so ...

```
(define (char-from-through first last)
   (define (ints-to-chars ints) ; visible only inside char-from-through
      (if (empty? ints)
         empty
         (cons (integer->char (car ints))
                  (ints-to-chars (cdr ints)))))

   (ints-to-chars
      (from-through (char->integer first)
                          (char->integer last))))
```

Problem: **sum-nums**

Problem: Write a procedure **sum-nums** that returns the sum of the numbers in a list, ignoring non-numbers. Example:

```
> (sum-nums '(5 x y 3 z 10 a b 3.4 1/16))
21.4625
```

Solution:

```
(define (sum-nums L)
   (cond
      [(empty? L) 0]
      [(number? (car L)) (+ (car L) (sum-nums (cdr L)))]
      [else (sum-nums (cdr L))]))
```

How could we make it skip the **3.4**, too?

```
[(and (number? (car L)) (exact? (car L))) ...]
```

Sidebar: Tracing

Racket has an excellent procedure-call tracing facility. To use it:

1. Add **(require racket/trace)** after the **#lang line**.
2. Add **(trace *procedure*)** after your definition for ***procedure***.

Example:

```
#lang racket
(require racket/trace)
(define (sum-nums L)
    (cond ...lots...) )
(trace sum-nums)
```

Usage:

```
> (sum-nums '(7 9 2 1))
>(sum-nums '(7 9 2 1))
> (sum-nums '(9 2 1))
> >(sum-nums '(2 1))
> > (sum-nums '(1))
> > >(sum-nums '())
< < <0
< < 1
< <3
< 12
<19
19
```

Sidebar, continued

It seems that in Dr. Racket, you can't use **trace** at the REPL prompt:

```
> (trace sum-nums)
.../racket/trace.rkt:294:41: sum-nums: cannot modify a constant
```

(Ok with **racket** from the command line.)

It seems that built-in procedures can't be traced:

```
> (trace string-split)
... set!: cannot mutate module-required identifier
```

A bit from the documentation for **(trace *id* ...)**

> Each *id* is **set!**ed to a new procedure that traces procedure calls and returns by printing the arguments and results of the call ...

More list procedures

define-values provides an easy way to create and initialize variables for experimentation:

```
> (define-values (a b c) (values 10 20 30))
```

(But why not just use **let**?)

The procedure **list** evaluates its arguments and makes a list from the values:

```
> (list 3 a (+ b c) 'test (* a b c) 'the 'end)
'(3 10 50 test 6000 the end)
```

What's the difference between **list** and **list***, below?

```
> (list* 3 a (+ b c) 'test (* a b c) '(the end))
'(3 10 50 test 6000 the end)
```

List procedures, continued

More procedures:

```
> (append (range 3) (make-list 3 'a) (string->list "xyz"))
'(0 1 2 a a a #\x #\y #\z)

> (equal? '(1 2 3) (cdr (range 4)))
#t

> (list-ref '(a b c d e) 3)
'd

> (range 10 20 3.3)
'(10 13.3 16.6 19.900000000000002)

> (range 1 10 (+ 1 1/3))
'(1 7/3 11/3 5 19/3 23/3 9)
```

Even more list procedures

```
> (member 7 '(3 1 7 5 3))
'(7 5 3)

> (remove 'be '(to be or not to be))
'(to or not to be)

> (remove* '(3 8) '(8 6 3 4 2 3 9 7 8))
'(6 4 2 9 7)

> (take (drop (range 10) 5) 3)
'(5 6 7)

> (drop-right '(a b c d e) 2)
'(a b c)

> (list-tail (string->list "abcdef") 3)
'(#\d #\e #\f)
```

And still more...

```
> (define L '(we will go up the stairs!))

> (list-set L (index-of L 'up) 'down)
'(we will go down the stairs!)

> L
'(we will go up the stairs!)

> (shuffle '(a b c d e))
'(a c e d b)

> (list->string (take-common-prefix
                   (string->list "testing")
                   (string->list "tester")))
"test"
```

Some higher-order procedures for lists

A number of list-related procedures are higher order. Here two simple ones:

sort requires a comparison predicate for its third argument:

```
> (sort '(3 1 9 7 5) <)
'(1 3 5 7 9)

> (sort (string-split "how will they sort?") string>?)
'("will" "they" "sort?" "how")
```

What's the following doing?

```
> (indexes-where '(9 0 2 1 0) zero?)
'(1 4)
```

Lots more list procedures

Lots more: docs.racket-lang.org/reference/pairs.html (~135 procedures)

Unlike other types, list-oriented procedures tend to not have "list" in their names, especially "classic" procedures like **length** and **append**.

Source code on lectura: **/usr/local/racket/collects/racket/list.rkt**

- See if you think Racket library source code is more readable than Haskell's Prelude code.

Lists of lists of lists...

As you'd hope, lists can be nested.

```
> (define L '((1 a 2 b) (+ 3 4) (a < b)))
> L
'((1 a 2 b) (+ 3 4) (a < b))

> (first L)
'(1 a 2 b)

> (second L)
'(+ 3 4)

> (third L)
'(a < b)

> (second (third L))
'<
```

car and **cdr**, continued

Given **L**...

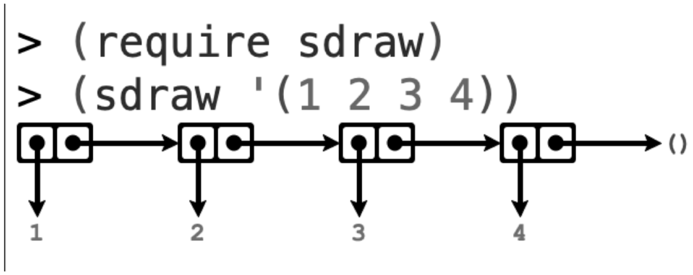
```
> L
'((a b) c ((d e) f))
```

How can we get the...

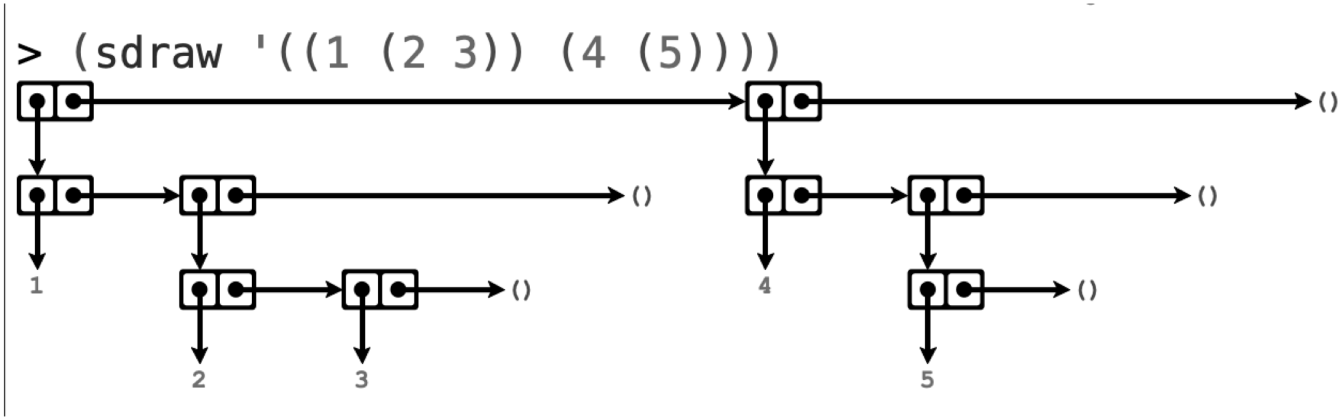
```
'a ?
(caar L)

'b ?
(cadar L)

'(d e) ?
> (caaddr L)

'f ?
(last (last L))
```

Sidebar: **sdraw**

The **sdraw** package draws cons-cell diagrams. After installing it with **File>Install** in Dr. Racket, you can do this:



```
> (require sdraw)
> (sdraw '(1 2 3 4))
```



```
> (sdraw '((1 (2 3)) (4 (5))))
```

docs.racket-lang.org/sdraw/index.html

sum-nums2

Let's extend **sum-nums** so that it can handle nested lists, summing all the numbers found within any of the lists:

```
> (sum-nums2 '(3 4 (7 3) (5 (6 (7)))))
35
```

Here's a copy of **sum-nums** renamed to **sum-nums2**:

```
(define (sum-nums2 L)
   (cond
      [(empty? L) 0]
      [(number? (car L)) (+ (car L) (sum-nums2 (cdr L)))]
      [else (sum-nums2 (cdr L))]))
```

As-is, what happens with a list like '(3 (4 5) 6)?

sum-nums2, continued

Revised:

```
(define (sum-nums2 L)
   (cond
      [(empty? L) 0]
      [(number? (car L)) (+ (car L) (sum-nums2 (cdr L)))]
      [(list? (car L))   ; if the head is a list, recurse on both head and tail
                  (+  (sum-nums2 (car L))
                       (sum-nums2 (cdr L)))]
      [else (sum-nums2 (cdr L))]))
```

Could/should we introduce a **let**?

sum-nums3

Here's a version with a **let**:

```
(define (sum-nums3 L)
   (if (empty? L)
      0
      (let ([head (car L)] [tail (cdr L)])
         (cond
            [(number? head) (+ head (sum-nums3 tail))]
            [(list? head) (+ (sum-nums3 head) (sum-nums3 tail))]
            [else (sum-nums3 tail)]))))
```

Why was the **if** introduced?

Could we improve it further?

flatten

The built-in **flatten** procedure "flattens" a possibly-nested list:

```
> (flatten '(1 (2 (3 4) 5)))
'(1 2 3 4 5)

> (flatten '((1 (2 3)) (4 (5 (6)))))
'(1 2 3 4 5 6)
```

Let's write our own version! How can we approach it?

```
> (define L '((1 (2 3)) (4 (5 (6)))))

> (car L)
'(1 (2 3))

> (cdr L)
'((4 (5 (6))))
```

flatten, continued

Here's a simple solution, with a shortcut:

```
(define (my-flatten L)
  (cond
    [(empty? L) empty]
    [(list? L) (append
                 (my-flatten (car L))
                 (my-flatten (cdr L)))]
    [else (list L)]))
```

What's the shortcut?

We're flattening non-lists into lists!

Could we write flatten in Haskell?

```
> flatten [1,2,[3,[4,5]]]
```

No—the list above isn't homogenous!

A trace on a simple case:

```
> (my-flatten '(3 4 5))
>(my-flatten '(3 4 5))
> (my-flatten 3)
< '(3)
> (my-flatten '(4 5))
> >(my-flatten 4)
< <'(4)
> >(my-flatten '(5))
> > (my-flatten 5)
< < '(5)
> > (my-flatten '())
< < '()
< <'(5)
< '(4 5)
<'(3 4 5)
'(3 4 5)
```

flatten, continued

Here's a revision:

```
(define (my-flatten2 L)
  (if (empty? L)
      empty
      (let ([head (car L)] [tail (cdr L)])
         (if (list? head)
             (append (my-flatten2 head)
                             (my-flatten2 tail))
             (cons head (my-flatten2 tail))))))
```

But...

```
> (flatten 3)     ; built-in
'(3)
> (my-flatten2 3)
car: contract violation
  expected: pair?
```

A trace of the same call:

```
> (my-flatten2 '(3 4 5))
>(my-flatten2 '(3 4 5))
> (my-flatten2 '(4 5))
> >(my-flatten2 '(5))
> > (my-flatten2 '())
< < '()
< <'(5)
< '(4 5)
<'(3 4 5)
'(3 4 5)
```

Procedures involving lists of lists

Generally speaking, lists "work" as values:

```
> (member '(a b) '(a b (b a) (a b) (b c)))
'((a b) (b c))

> (indexes-of '(a b (b a) (a b) (b c) (a b c) (a b)) '(a b))
'(3 6)
```

Sometimes handy...

```
> (permutations '(a b c))
'((a b c) (b a c) (a c b) (c a b) (b c a) (c b a))

> (combinations '(a b c d) 2)
'((a b) (a c) (a d) (b c) (b d) (c d))

> (cartesian-product '(big little) '(x y z))
'((big x) (big y) (big z) (little x) (little y) (little z))
```

Sidebar: quote with lists

Originally, quoted lists were created with the quote special form. It still works:

```
> (quote (a b c 1 2 3))
'(a b c 1 2 3)

> (quote (a b c (d (e f) g) h))
'(a b c (d (e f) g) h)
```

Note that '(a (b c)) is not the same as '(a '(b c)). Flattening shows it a bit:

```
> (flatten '(a (b c)))
'(a b c)

> (flatten '(a '(b c)))
'(a quote b c)
```

Try (sdraw '(a (b c))), (sdraw '(a '(b c))), and (sdraw '(a '(b 'c))).

Intentionally blank

Pairs

Pairs

What will this do?

```
> (define x (cons 3 4))
> x
'(3 . 4)
> (pair? x)
#t
> (car x)
3
> (cdr x)
4
```

"(cons *a d*) Returns a newly allocated *pair* whose first element is *a* and second element is *d*." –RR

'(3 . 4) is commonly called a *dotted pair*. Dotted pairs are often used as 2-tuples.

How can a dotted pair be drawn with boxes and arrows?

Pairs, continued

Problem: Write an expression that produces `'(a b . c)`

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

How about `'((a . b) . c)` ?

```
(cons (cons 'a 'b) 'c)
```

~~How about '((a . b) . (c . d)) ?~~

~~[and more...]~~

Formally, we say that a list is a pair whose second element is a list.

Proper lists end with an empty list. `'(a . b)` and `'(a b . c)` are *improper lists*.

Pairs, continued

Again, a list is a pair whose second element is a list.

Are pairs lists? Are lists pairs?

```
> (pair? '(1 2 3))
#t
> (pair? '(1))
#t
> (pair? empty)
#f
> (list? '(3 . 4))            ; (cons 3 4)
#f
> (list? '(3 4 . 5))          ; (cons 3 (cons 4 5))
#f
> (list? '(3 4 5 . 6))        ; (cons 3 (cons 4 (cons 5 6)))
#f
```

What's a performance implication of pair? vs. list?

A curiosity

```
> (define x (range 10000000))
> (time (pair? x))
cpu time: 0 real time: 0 gc time: 0
#t
> (time (list? x))
cpu time: 29 real time: 30 gc time: 0
> (time (list? x))
cpu time: 15 real time: 16 gc time: 0
> (time (list? x))
cpu time: 6 real time: 6 gc time: 0
> (time (list? x))
cpu time: 3 real time: 3 gc time: 0
> (time (list? x))
cpu time: 2 real time: 2 gc time: 0
> (time (list? x))
cpu time: 1 real time: 2 gc time: 0
> (time (list? x))
cpu time: 0 real time: 0 gc time: 0
```

Eli Barzilay wrote,

> Re list?, you’re actually getting some very nice timings that demonstrate what Racket is doing… This starts with first second … rest which are a bit different than car cadr … cdr. The latter just want the value to exist, so they’re constant-time operations. But with first et al, they require an input which is a proper list, and that requires testing the value with list?, which in turn must chase the pointers all the way to the tail of the list. (It actually uses that hare/tortoise algorithm, since lists can have pointer cycles.)
>
> So the solution is that every time that you verify that something is a list?, you mark the half-way point as a pair which is known to start a proper list, and the second test needs to reach only to *that* point, marking the half-way to it, etc. This makes the time go down in half every time you try it with the same input list, so the amortized time is still constant.

Sidebar: A minimalist view

With only symbols, integers, arithmetic, **if**, comparisons, and **cons**, **car**, and **cdr**, what kinds of data structures can be we build and access?

Imagine a set of procedures that implement a string type that holds its length, too:

```
> (define s (str-make '(t e s t)))
> s
'(:s: 4 t e s t)     ; Design decisions: (1) Lists that represent strings start with :s:
                                         (2) Second element is the length of the string
```

```
> (str-length s)
4

> (str-concat s s)
'(:s: 8 t e s t t e s t)
```

```
> (str-concat s (str-make '(- m e !)))
'(:s: 8 t e s t - m e !)

> (str-print (str-concat s (str-make '(- m e !))))
test-me!

> (str-length '(a b c))
not a str
```

Sidebar, continued

Using only pairs for storage could we build...

- A binary tree?
- A two-dimensional array?
- A hash table?
- An address book entry?

"The single compound-data primitive pair, implemented by the procedures **`cons`**, **`car`**, and **`cdr`**, is the only glue we need."—SICP

Sidebar, continued

Here's a start on a simple Lisp written in Java:

```
abstract class Value {}

class Symbol extends Value
{
   String name;
}

class Int extends Value
{
   int value;
}

class Pair extends Value
{
   Value car, cdr;
}
```

GOTO 156!

Keyword arguments (was Intentionally Blank)

Some built-in procedures have *keyword arguments*. Here's an excerpt from the string-join documentation:

```
(string-join strs
   [sep #:before-first before-first #:before-last before-last
        #:after-last after-last])
```

Let's experiment:

```
> (define parts (string-split "one two three four"))
> parts
'("one" "two" "three" "four")
> (string-join parts)
"one two three four"
> (string-join parts "-")
"one-two-three-four"
> (string-join parts "," #:before-first ">" #:after-last "<")
">one,two,three,four<"
> (string-join parts ", " #:before-first "Parts: " #:before-last " and " #:after-last "." )
"Parts: one, two, three and four."
```

Defining variadic procedures

Defining variadic procedures

Here's a procedure that accepts two or more arguments:

```
(define (show-args a1 a2 . more-args)
   (printf "a1: ~a, a2: ~a, more-args: ~a" a1 a2 more-args))
```

Usage:

```
> (show-args 10 20 30 40)
a1: 10, a2: 20, more-args: (30 40)

> (show-args 10 20)
a1: 10, a2: 20, more-args: ()

> (show-args 10)
show-args: arity mismatch;  expected: at least 2;  given: 1
```

What's interesting about the syntax used to describe this variadic procedure?

An existing syntactic element was used; nothing new was devised.

Variadic procedures, continued

Contrast with Python:

```
def show_args(a1, a2, *more_args):
   print(f"a1: {a1}, a2: {a2}, more_args: {more_args}")

>>> show_args(10,20,30,40)
a1: 10, a2: 20, more_args: (30, 40)
```

And Java:

```
 static void showargs(int a1, int a2, int ...more_args) {
       System.out.printf("a1: %d, a2: %d, more_args: %s\n",
          a1, a2, Arrays.toString(more_args));
       }

jshell> varargs.showargs(10, 20, 30, 40);
a1: 10, a2: 20, more_args: [30, 40]
```

Variadic procedures, continued

Recall that we can make a string like this:

```
> (string #\T #\e #\s #\t)
"Test"
```

What's the variadic aspect of **string**?

Let's write **my-string**, with the same behavior:

```
> (my-string #\R #\a #\c #\k #\e #\t)
"Racket"
```

Two handy procedures:

```
> (make-string 3)
"\u0000\u0000\u0000"
```

(mutable...)

```
(string-set! s 0 #\x)
```

Variadic procedures, continued

Wanted:

```
> (my-string #\R #\a #\c #\k #\e #\t)
"Racket"
```

Solution:

```
(define (my-string . chars)
   (let [(result (make-string (length chars)))]
      (for ((i (in-naturals)) (c chars))
         (string-set! result i c))
      result))
```

Will just (my-string) work?

Would a recursive solution be better?

Code as data

Code as data

What does the following create?

```
> (define (add a b) (let ([sum (+ a b)]) (printf "sum is ~a\n" sum) sum))
```

What does the following create?

```
> '(define (add a b) (let ([sum (+ a b)]) (printf "sum is ~a\n" sum) sum))
```

Let's give that list a name and explore it!

```
> (define code ^)
> (car code)
'define

> (cadr code)
'(add a b)

> (caddr code)
'(let ((sum (+ a b))) (printf "sum is ~a\n" sum) sum)  ;  note no [ ]!
```

Code as data

At hand:

```
> code
'(define (add a b) (let ((sum (+ a b))) (printf "sum is ~a\n" sum) sum))
```

An apostrophe transforms the source code for a procedure into an easily manipulated data structure!

How could we get the procedure's name?

```
> (caadr code)
'add
```

How could we get the parameter names?

```
> (cdr (second code))
'(a b)
```

Given a `let` like `'(let ([x 2] [y 3] [z (+ 4 5)]) (+ x (* y z)))`, how could we get a list of the names that are bound?

S-expressions

Most generally, we can say that the following are *symbolic expressions*, or *S-expressions*:

```
'xyz
3/4
(10 20 30 40)
(+ 3 4)
(if (< x y) (f x) (g y))
((course 372)
    (name "Comparative Programming Languages) (location "GS 906"))
```

"S-expression: The essential building block of Racket programs. An S-expression can be either a) an atomic value (like a string, number, or symbol) or b) a parenthesized list of values." [*Beautiful Racket* [BR] glossary]

Recall the title of McCarthy's first paper on Lisp:

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

The **read** procedure

In essence, the **read** procedure reads an s-expression:

```
> (read)
just    ; input in bold
'just   ; read returned the symbol 'just

> (read)
(just some

  values:
3.4 #f (a b
)
        )
'(just some values: 3.4 #f (a b))
```

We have now seen three ways to read data: **read-line**, **read-char**, and **read**.

RR: "**(read [*in*])** reads and returns a single *datum* from **in**." (*in* is a *port*.)

Here's a procedure that prints the *s-expressions* (forms) contained in a file:

```
(define (print-exprs filename)
   (define (read-and-print port)
      (let ([expr (read port)])
         (cond
            [(equal? expr eof) (void)]
            [else (printf "Expression: ~a\n" expr)
                (read-and-print port)])))
   (read-and-print (open-input-file filename)))
```

What's the type of each expression below?

```
% cat exprs.1
(10 20 30 40)
(+ 3 4)
; Here's a comment!
(if (< x y)
   (f x)
   (g y))
9/54
```

Execution:

```
> (print-exprs "exprs.1")
Expression: (10 20 30 40)
Expression: (+ 3 4)
Expression: (if (< x y) (f x) (g y))
Expression: 1/6
```

"Homoiconic"

"Languages in which program code is represented as the language's fundamental data type are called 'homoiconic'."—wiki.c2.com

"A language is homoiconic if a program written in it can be manipulated as data using the language, and thus the program's internal representation can be inferred just by reading the program itself. This property is often summarized by saying that the language treats code as data."—W

- All Lisps are fundamentally homoiconic.
- Is Java homoiconic?
- Is Python homoiconic?
- Is Prolog homoiconic?

Sidebar: Is Prolog homoiconic?

Experiments:

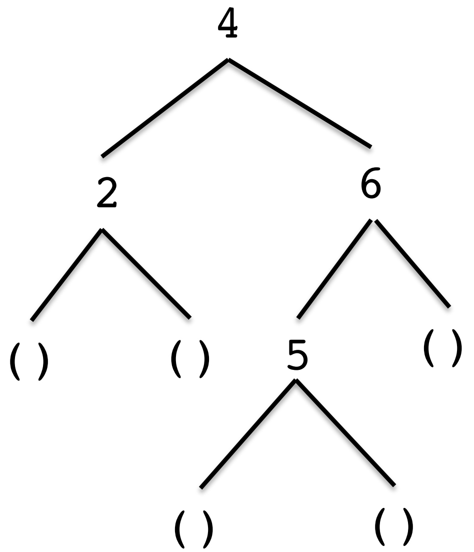
```
?- display(member(X,[_|T]) :- member(X,T)).
:-(member(_4052,[_4046|_4048]),member(_4052,_4048))
true.

?- atom_chars('test',C),member(E,C),writeln(E),fail.
t
e
s
t
false.

?- display((atom_chars('test',C),member(E,C),writeln(E),fail)).
','(atom_chars(test,_4164),','(member(_4168,_4164),','(writeln(_4168),fail)))
true.
```

REPLACEMENTS!
Discard 156-157 in prior set

Binary trees
(adapted from Dr. Collberg's slides)

Binary trees with lists

Here's a binary tree:



A little more readable:

```
(4
    (2
        ()
        ())
    (6
        (5
            ()
            ())
        ()))
```

Can we represent it with a Racket list?

```
    (4 (2 () ()) (6 (5 () ()) ()))
```

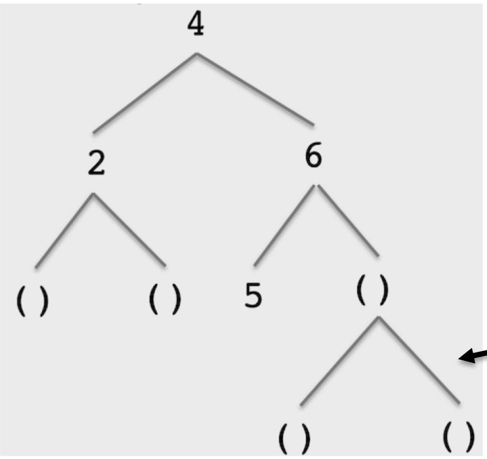
How about with a Python list?

```
    [4, [2, [], []], [6, [5, [], []], []]]
```
(more text than Racket...)

Accessors for our tree

Our tree:

```
> t
'(4 (2 () ()) (6 (5 () ()) ()))
```



Bug

What do these values represent?

```
> (car t)
4
> (cadr t)
'(2 () ())
> (caddr t)
'(6 (5 () ()) ())
```

Let's write some accessors:

```
(define (key tree) (car tree))
(define (left tree) (cadr tree))
(define (right tree) (caddr tree))
```

Let's use them...

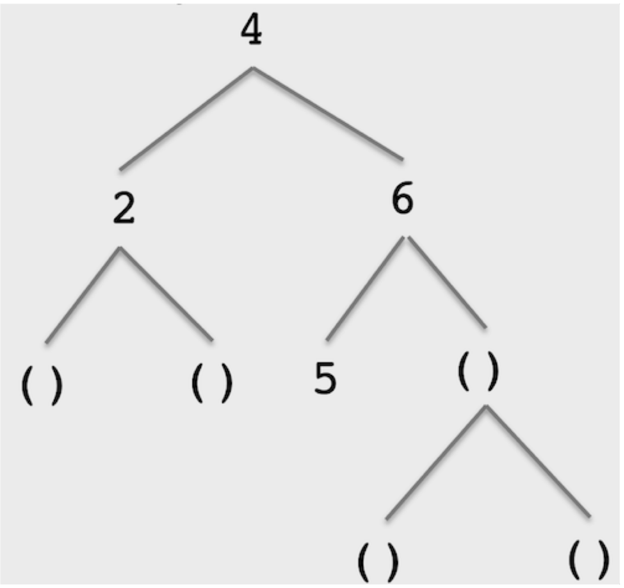
```
> (key (left t))
2
> (right t)
'(6 (5 () ()) ())
```

How can we get the 5?

```
> (key (left (right t)))
5
```

Let's write an in-order traversal procedure:

```
> (in-order empty)
> (in-order '(7 () ()))
7
> (in-order '(4 (2 () ()) (6 (5 () ()) ())))
2 4 5 6
```



Solution:

```
(define (in-order tree)
  (cond
    ((null? tree) (void))
      (else
         (in-order (left tree))
         (printf "~a " (key tree))
         (in-order (right tree)))))
```

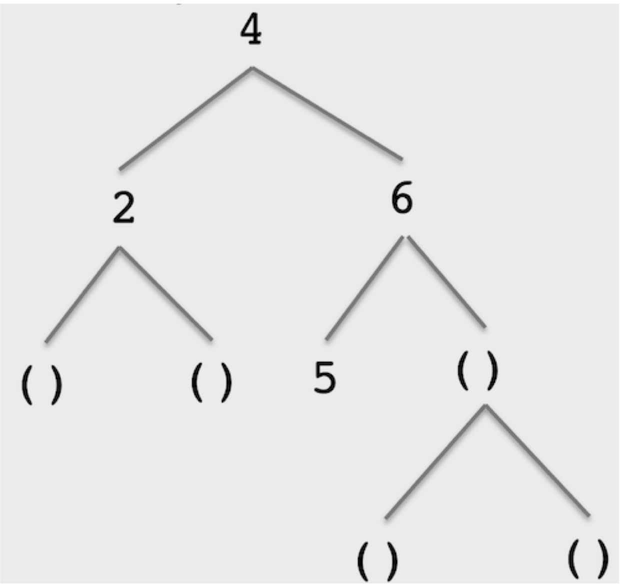
A pretty-printer for trees

Let's write a "pretty-printer" for our trees:

```
> (print-tree t)
4
   2
   6
      5
```

Here it is; how does it work?

```
(define (print-tree tree [depth 0])
  (cond
    [(null? tree) (void)]
    [else
        (display (make-string depth #\ ))
        (display (key tree))
        (newline)
        (print-tree (left tree) (+ depth 3))
        (print-tree (right tree) (+ depth 3))]))
```



Insertion into trees

Let's write **insert** for our trees:

```
> (insert 5 '())
'(5 () ())

> (insert 3 ^)
'(5 (3 () ()) ())

> (insert 10 ^)
'(5 (3 () ()) (10 () ()))

> (insert 1 ^)
'(5 (3 (1 () ()) ()) (10 () ()))

> (print-tree ^)
5
  3
    1
  10
```

Let's walk through a solution:

```
(define (insert value tree)
  (cond
    [(empty? tree) (void)
       (list value '() '())]
    [(< value (key tree))
       (list (key tree)
             (insert value (left tree))
             (right tree))]
    [(> value (key tree))
       (list (key tree)
             (left tree)
             (insert value (right tree)))]
    [else
       (printf "ignored: ~a\n" value) tree]))
```

Association lists

Association lists

A commonly used Lisp data structure is an *association list:*

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

The procedure **(assoc *value list*)** searches **list** for a pair whose **car** is **equal?** to ***value***.

```
> (assoc 'one A)
'(one . 1)

> (assoc 'two A)
'(two . 2)

> (assoc 'four A)
#f
```

Association lists, continued

Here's another association list:

```
> denoms
'((pennies . 1) (nickels . 5) (dimes . 10) (quarters . 25))
```

Problem: Write a procedure that will compute the total number of cents for some coins. Assume **denoms** is a global variable.

```
> (total '(2 nickels 3 pennies 2 quarters))
63
> (total '(7 pennies 3 dimes))
37
```

We'll live with awkward wording:

```
> (total '(1 quarters 1 pennies 1 dimes))
36
```

Association lists, continued

Given:

```
> denoms
'((pennies . 1) (nickels . 5) (dimes . 10) (quarters . 25))
```

Desired:

```
> (total '(2 nickels 3 pennies 2 quarters))
63
```

Solution:

```
(define (total L)
  (if (empty? L) 0
    (let* [(count (car L))
           (coin (cadr L))
           (info (assoc coin denoms))]
      (+ (* count (cdr info)) (total (cddr L))))))
```

Association lists, continued

The pairs in an association list need not be be dotted pairs:

```
> items
'((antfarm "Ant Farm" 24.95) (twinkies "Twinkies" 2.75) (hamster "Hamster" 9.25))
```

Problem: Write a **lookup** procedure that assumes **items** is a global.

```
> (lookup 'antfarm)
"Ant Farm: $24.95"

> (lookup 'tiger)
"Not found"
```

```
(define (lookup item)
  (let ([match (assoc item items)])
    (if match
        (format "~a: $~a"
                (cadr match)
                (caddr match))
        "Not found")))
```

Fill in the binding(s) for the **let**!

More examples

More association lists:

```
(define bands
   '(('black . 0) ('brown . 1) ('red . 2) ('orange . 3) ('yellow . 4)
     ('green . 5) ('blue . 6) ('violet . 7) ('grey . 8) ('white . 9)))

(define dir-to-offset ; from travel.hs, the robot moving around on a grid
   '((north . (0 .  1)) (south . (0 .  -1)) (east . (1 .  0)) (west . (-1 .  0))))
```

Racket has a **roomba** package. Here's one of its procedures:

```
> (decode-roomba-manufacturing-code "JEN04110061006012345")
 '((manufacturer . jetta)
   (status . new)
   (model . 4110)
   (date . #(2006 10 6))     ; A vector!  Coming soon!
   (rev . 0)
   (serial . 12345))
```

Sidebar: Minimalism in library design

assoc has these arguments: **(assoc *value lst* [*is-equal?*])** where **is-equal?** is a predicate that defaults to **equal?**.

Racket has a number of *"indelicately named"* built-in procedures related to association lists:

- **(assw *value lst*)** uses **equal-always?** for comparison. (Related to mutability...)

- **(assv *value lst*)** uses **eqv?** for comparison.

- **(assq *value lst*)** uses **eq?** for comparison.

- **(assf *proc lst*)** matches when **(*proc* (car *pair*))** returns true.

How many of these do we really need? Why so many?

Lists and mutability

Racket lists are immutable, but...

In Scheme, lists are mutable!

```
% chezscheme
Chez Scheme Version 9.5
> (define L '(a b c d e))
> L
(a b c d e)

> (set-car! L 'x)  ;  "list surgery"
> L
(x b c d e)

> (set-cdr! L '(3 4 5))
> L
(x 3 4 5)

> (set-cdr! (cdddr L) '(10 20 30))
> L
(x 3 4 5 10 20 30)
```

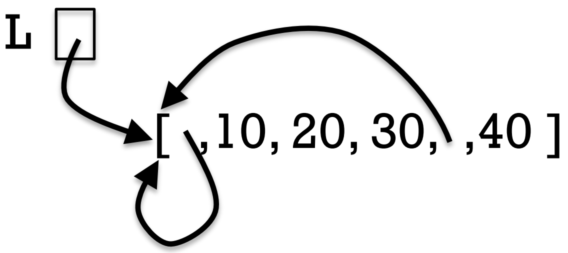
```
> (set-cdr! (cddr L) L)
> L
Warning in pretty-print: cycle detected;
 proceeding with (print-graph #t)
#0=(x 3 4 . #0#)

> (list-ref L 2)
4
> (list-ref L 3)
x
> (list-ref L 4)
3
> (list-ref L 5)
4
> (list-ref L 6)
x
```

blog.racket-lang.org/2007/11/getting-rid-of-set-car-and-set-cdr.html

Sidebar: Python lists can be cyclic, too!

```
>>> L = [10, 20, 30]

>>> L.append(L)
>>> L
[10, 20, 30, [...]]

>>> len(L)
4

>>> L[-1]
[10, 20, 30, [...]]

>>> L[-1][1]
20
```

```
>>> L.append(40)

>>> L
[10, 20, 30, [...], 40]

>>> L[0] = L
>>> L
[[...], 20, 30, [...], 40]
```



Vectors

Vector basics

R6RS says,

> "Vectors, like lists, are linear data structures, representing finite sequences of arbitrary objects. Whereas the elements of a list are accessed sequentially through the chain of pairs representing it, the elements of a vector are addressed by integer indices. Thus, vectors are more appropriate than lists for random access to elements."

```
> (define v (make-vector 3 7))  v
'#(7 7 7)

> (vector-set! v 0 '(a b))  v
'#((a b) 7 7)

> (vector-length v)
3

> (vector-ref v 2) (car (vector-ref v 0))
7
'a
```

```
> (vector->list v)
'((a b) 7 7)
```

What does the following tell us?

```
   > (immutable? #(5 3))
   #t

   > (immutable? (vector 5 3))
   #f
```

Association list + vectors

Imagine a game that uses an association list to represent players, with each player having two numeric attributes, "attack" and "health" that can change over time:

```
> (define players (list (cons 'a (vector 3 12)) (cons 'b (vector 2 9))))
> players
'((a . #(3 12)) (b . #(2 9)))
```

Let's imagine an accessor and a mutator for a player's health:

```
> (get-health 'b)
9

> (adjust-health 'b -5)
> (get-health 'b)
4
```

Implementation:

```
(define (get-health player)
   (let ([stats (cdr (assoc player players))])
      (vector-ref stats 1)))

(define (adjust-health player adj)
   (let ([stats (cdr (assoc player players))])
      (vector-set!
         stats 1 (+ (vector-ref stats 1) adj))))
```

Key point: Vectors can be put in immutable objects to provide fields we can change.

Example: by-length

Imagine a procedure that splits up a string and groups its words by length into a vector of lists:

```
> (by-length "now a test of it with this" 5)
'#(() ("a") ("it" "of") ("now") ("this" "with" "test") ())
   0    1        2         3              4            5   (vector indices/word lengths)
```

The second argument specifies the maximum word length. Any words exceeding the maximum go into element zero:

```
> (by-length "now a test of it with this" 3)
'#(("this" "with" "test") ("a") ("it" "of") ("now"))
```

How can we approach it?

by-length, continued

Let's make a vector of five empty lists:

```
> (define v (make-vector 5 empty))
> v
'#(() () () () ())
```

Let's add "**test**" to the fourth list:

```
> (vector-set! v 4 (cons "test" (vector-ref v 4)))
> v
'#(() () () () ("test"))
```

How can we add "**this**" to the fourth list?

```
> (vector-set! v 4 (cons "this" (vector-ref v 4)))
> v
'#(() () () () ("this" "test"))
```

What if we didn't want the order reversed?

```
> (vector-set! v 4 (append (vector-ref v 4) (list "this")))
```

by-length, continued

Wanted:

```
> (by-length "now a test of it with this" 5)
'#(() ("a") ("it" "of") ("now") ("this" "with" "test") ())

> (by-length "now a test of it with this" 3)
'#(("this" "with" "test") ("a") ("it" "of") ("now"))
```

Solution:

```
(define (by-length s maxlen)
   (define words-by-len (make-vector (add1 maxlen) empty))
   (for ([w (string-split s)])
      (let* ([len (string-length w)]
             [pos (if (<= len maxlen) len 0)]
             [current (vector-ref words-by-len pos)])
          (vector-set! words-by-len pos (cons w current))))
   words-by-len)
```

Key point: We started with a vector of empty lists and repeatedly replaced, with **vector-set!**, a given list **L** with **(cons *new-element* L)**.

Simple utility: Character sorting

Imagine a program that sorts its input character by character:

```
% echo anthropomorphologically | racket csort.rkt

aacghhilllmnoooopprrty%
% echo to be or not to be | racket csort.rkt

     bbeenoooorttt%
%
```

First, where are those blank lines coming from? And why is my Bash prompt appearing at the end of two lines?

How can we approach it?

Character sorting, continued

Here's the solution. How does it work?

```
(define (build-char-counts)
   (let ([counts (make-vector #x10FFFF)])
      (for ([c (in-port read-char)])
         (let ([i (char->integer c)])
            (vector-set! counts i (add1 (vector-ref counts i)))))
      counts))

(define (show-chars counts)
   (for ([pos (vector-length counts)])
      (for ([n (vector-ref counts pos)])
         (write-char (integer->char pos)))))

(show-chars (build-char-counts))
```

What's big-O for this sort?

Experiment: vector vs. list access time

Recall R6RS:

> "Vectors, like lists, are linear data structures, representing finite sequences of arbitrary objects. Whereas the elements of a list are accessed sequentially through the chain of pairs representing it, the elements of a vector are addressed by integer indices. Thus, vectors are more appropriate than lists for random access to elements."

A simple experiment...

```
> (define L (range 50000000)) ; 50 million
> (time (list-ref L 49999999))
cpu time: 269 real time: 274 gc time: 0
49999999

> (define v (list->vector L))
> (time (vector-ref v 49999999))
cpu time: 0 real time: 0 gc time: 0
49999999
```

Two catch-ups!
("named **`let`**", and a little **`lambda`**)

Loops with "named let"

"named **let**"

This is an example of a "named **let**":

```
(let loop ([n 3])
   (if (= n 0)
      (displayln "Done!")
      (begin
         (printf "~a...\n" n)
         (loop (sub1 n)))))
```

Execution:

```
3...
2...
1...
Done!
```

What's the idea of it? (Don't peek ahead!)

named let, continued

At hand:

```
(let loop ([n 3])
    (if (zero? n)
        (displayln "Done!")
        (begin
            (printf "~a...\n" n)
            (loop (sub1 n)))))
```

A named let is essentially a let that's a loop implemented with an implicit recursive procedure.

Operation:

- loop gets bound to a procedure with one parameter, n
- That procedure is initially called with 3
- (loop (sub1 n)) is a recursive call to that procedure
- if n is zero, (displayln "Done!") is evaluated and becomes the value of the let

Problem

Problem: Write a named **let** that will print the elements in a list **values**, numbering them:

```
> values
'(some symbols here)
> (let loop ...your code here...)
1: some
2: symbols
3: here
```

Solution:

```
(let loop ([pos 1][lst values])
   (if (empty? lst) (void)
      (begin
         (printf "~a: ~a\n" pos (car lst))
         (loop (add1 pos) (cdr lst)))))
```

Could we have just used a **for** instead?

In essence...

In essence, a named let gives us a way to put a loop into the middle of some code.

Here's a procedure from /usr/local/racket/collects/racket/format.rkt:

```
(define (number->string* N base upper?)
 (cond
    [(memv base '(2 8 10 16))
     (let ([s (number->string N base)])
      (if (and (= base 16) upper?) (string-upcase s) s))]
    [(zero? N) (string #\0)]
    [else
     (let loop ([N N] [digits null])
      (cond [(zero? N) (apply string digits)]
            [else (let-values ([(q r) (quotient/remainder N base)])
                     (loop q (cons (get-digit r upper?) digits)))]))]))
```

Would a helper function be better or worse than the named let?

(a little) `lambda`

lambda

The **lambda** special form creates an anonymous procedure:

```
> (lambda (n) (* n 2))
#<procedure>

> (^ 7)
14
```

In general, everything we learned about anonymous functions/lambda expressions in Haskell and Python carries over to **lambda** in Racket.

Haskell: `\n -> n * 2`
Python: `lambda n: n * 2`

What's the following code doing?

```
((lambda (a b c) (+ a (* b c))) 3 4 5)
```

lambda, continued

Did I really need to show you how to create procedures with **define**?

```
(define (add a b)
     (+ a b))
```

Nope!

```
(define add
    (lambda (a b)
        (+ a b)))
```

This works, too:

```
(define add
    (λ (a b)
        (+ a b)))
```

lambda, continued

Recall that **sort** requires a comparison procedure:

```
> (sort '(b a c d e) symbol<?)
'(a b c d e)
```

Problem: Using a **lambda**, sort a list of words by decreasing length.

```
> (sort (string-split "the words to sort are these")
            (lambda ...))
```

Solution:

```
> (sort (string-split "the words to sort are these")
      (lambda (s1 s2) (> (string-length s1) (string-length s2))))
```

Macros

Something familiar to many: The C preprocessor

The C preprocessor provides a *macro* facility.

```
$ cat macros.c
#include <stdio.h>
#include <limits.h>

#define abs(x) ((x) < 0 ? -(x) : (x))
#define iprint(e) printf(#e " = %d\n", (e))

int main()
{
   iprint(3+4);
   iprint(INT_MAX);
   iprint(abs('a' - 'z'));
}
```

Execution:

```
$ gcc macros.c && ./a.out
3+4 = 7
INT_MAX = 2147483647
abs('a' - 'z') = 25
```

Try `gcc –E macros.c`, too!

A very simple *macro*

I'd like to be able to create a variable and initialize it to zero like this:

```
> (zero x)
> x
0
```

Would **zero** need to be a special form? Why or why not?

It needs to be a special form. It doesn't evaluate the subform **x**.

zero, continued

Wanted—create a variable and initialize it to zero:

```
> (zero x)
> x
0
```

We can define a special form **zero** like this:

```
(define-syntax-rule
    (zero var) (define var 0))
```

We've now defined **zero** as a new syntactic keyword and specified its *expansion*.

We say that **zero** is a *macro*.

zero, continued

At hand:

```
(define-syntax-rule
   (zero var) (define var 0))
```

- The first argument of **define-syntax-rule** specifies a *pattern*: **(zero var)**
- **var** is a *pattern variable*.
- The second argument specifies a *template:* **(define var 0).**

When the compiler sees

```
(zero x)
```

the pattern is matched, **var** is given the value **x** and **(zero x)** is replaced with

```
(define x 0)
```

zero, continued

zero is not a procedure! **zero** is new keyword, on equal footing with **let**, **if**, **cond** and more.

```
> zero
string:1:0: zero: bad syntax

> (zero)
string:1:0: zero: bad syntax

> (zero x y)
string:1:0: zero: bad syntax

> (zero 7)
string:1:6: define: bad syntax
in: (define 7 0)
```

zero, continued

At hand:

```
(define-syntax-rule
   (zero var) (define var 0))
```

With XREPL we can see the expansion of a macro using the **,stx** command:

```
> ,stx (zero total) !
```

Note: *Does not work in Dr. Racket!*

```
; Syntax set
; expand ->
; (define-values (total) '0)
```

Any surprises?

zero, continued

define is itself a macro!

```
> ,stx (define somevar 25) !
; Syntax set
; expand ->
; (define-values (somevar) '25)
```

Whenever we use the special form **define**, Racket is transforming that code into new code that uses **define-values**, a more general special form!

Three steps:

```
(zero x)
(define x 0)
(define-values (x) '0)
```

define-values is said to be a *core form*.

zero, continued

We can add an asterisk to the **,stx** command to see the intermediate steps

```
> ,stx (zero total) * !
; Syntax set
; Stepper:
; ---- Macro transformation ----
; (zero total)
;   ==>
; (define:1 total 0)
; expand ->
; (define-values (total) '0)
```

(Try it with two asterisks, too: **,stx (zero total) ** !**)

Sidebar: Macros among us!

Let's see if **if** is a macro:

```
> ,stx (if 1 2 3) !
; expand ->
; (if '1 '2 '3)
```

How about **and**?

```
> ,stx (and (< 1 2) (> 3 4) (= 5 6)) !
; Syntax set
; expand ->
; (if (< '1 '2) (if (> '3 '4) (= '5 '6) '#f) '#f)
```

I lied! In fact, the final result is this:

```
; (if (#%app < '1 '2) (if (#%app > '3 '4) (#%app = '5 '6) '#f) '#f)
```

I'll elide #%apps in these examples.

Note:

```
> '3
3
> (number? '3)
#t
> (symbol? '3)
#f
```

Macros among us, continued!

Here's **or**:

```
> ,stx (or (< 1 2) (> 3 4) (= 5 6) (zero? 7)) !
; Syntax set
; expand ->
; (let-values (((or-part) (< '1 '2)))
  (if or-part
   or-part
   (let-values (((or-part) (> '3 '4)))
    (if or-part
     or-part
     (let-values (((or-part) (= '5 '6)))
      (if or-part or-part (zero? '7)))))))
```

Macros among us, continued!

Let's try our **sign** procedure:

```
> ,stx (define (sign n)
            (cond
             ((< n 0) -1)
             ((zero? n) 0)
             (else 1))) !

(define-values
  (sign)
   (lambda (n)       ; an anonymous function...
    (if (< n '0)
      (let-values () '-1)
      (if (zero? n) (let-values () '0) (let-values () '1)))))
```

Speculate: How many lines does the seven-line **wc** procedure on slide 97 expand into?

78 lines

Homesick!

April 15, 2023

Dear Mom and Dad,

I'm not sure if I like it here at Camp Racket. Everywhere I look there are parentheses! (And I mean EVERYWHERE!!)

I really miss some things, too, like the += operator we have at home in Javatown. I sure wish I could do something like this:

```
> (define a 10)
> (+= a 5)
> a
15
> (+= a (* 3 4))
> a
27
```

Your son,
Cuthbert

Homesick, continued

Let's help, by writing a += procedure to send to Cuthbert!

```
% cat plus-equal.rkt
#lang racket

(define (+= var value)
   (set! var (+ var value)))

% rk/i plus-equal.rkt
Welcome to Racket v8.5 [cs].
"plus-equal.rkt"> (define a 10)
"plus-equal.rkt"> (+= a 5)
"plus-equal.rkt"> a
10
```

Are we ready to mail it to Camp Racket?

Homesick, continued

Could we use a macro?

With a macro, what should (+= a 5) expand into?

```
(set! a (+ a 5))
```

A += macro:

```
(define-syntax-rule
   (+= var value) (set! var (+ var value)))
```

What's the *pattern*?

```
(+= var value)
```

What are the *pattern variables*?

var and **value**

What's the *template*?

```
(set! var (+ var value))
```

Homesick, continued

At hand:

```
(define-syntax-rule
   (+= var value) (set! var (+ var value)))
```

Usage:

```
> (define a 10)
> (+= a 5)
> a
15

> ,stx (+= a 5) !
; expand ->
; (set! a (+ a '5))
```

Homesick, continued

Does our += behave exactly like Java's += ?

```
jshell> int a = 10

jshell> System.out.println(a += 5)
15

jshell> int b = a += 3

jshell> /var
|   int a = 18
|   int b = 18
```

Ours...

```
> (define a 10)
> (displayln (+= a 5))
#<void>

> (define b (+= a 3))
> a b
18

> (cons a b)
'(18 . #<void>)
```

For reference:

```
(define-syntax-rule
   (+= var value) (set! var (+ var value)))
```

+=, continued

At hand:

```
(define-syntax-rule
    (+= var value) (set! var (+ var value)))

> a
7
> (displayln (+= a 5))
#<void>
```

How does the behavior of our **+=** differ from Java and C?

It produces **#<void>** instead of the value assigned.

How can we fix it? (Don't peek ahead!)

+=, continued

Version 1:

```
(define-syntax-rule
   (+= var value) (set! var (+ var value)))
```

Version 2:

```
(define-syntax-rule
   (+= var value)
   (let ([result (+ var value)])
       (set! var result)
       result))
```

What's the pattern? What's the template?

"swap" in six languages

Consider the task of swapping the value of two variables.

```
Python:     x,y = y,x

Icon:       x :=: y          // :=: is the "swap" operator

C:          swap(&x, &y)     // Pointers...

C++:        swap(x, y)       // Uses "references"

Java:       x = x + y;       // Assuming integers...
            y = x - y;
            x = x - y;

JavaScript: [x, y] = [y, x]
```

Problem: Write **(swap x y)** for Racket

swap

Wanted: A **swap** special form.

```
> x y
'xylophone
25
> (swap x y)
> x y
25
'xylophone
```

Solution, from *The Racket Guide*:

```
(define-syntax-rule
   (swap v1 v2) (let ([tmp v1])
                       (set! v1 v2)
                       (set! v2 tmp)))
```

while

Imagine a **while** loop for Racket:

```
> (zero a)
> (while (< a 10) (displayln a) (+= a 3))
0
3
6
9
```

How could we build it?

while, continued

Wanted:

```
(while (< a 10) (displayln a) (+= a 3))
```

First, let's just write out the desired expansion. We'll use a named-**let**:

```
(let loop ()
    (if (< a 10)
        (begin
            (displayln a)
            (+= a 3)
            (loop))
         (void)))
```

while, continued

We'll let our **while** have zero or more forms after the condition:

```
(while (< a 10) (displayln a) (+= a 3))
(while (f))
(while (> x y) (f x y) (g x) (x += 5) (h x y))
```

Our **while** requires another element of pattern and template syntax, an ellipsis (**...**).

```
(define-syntax-rule
     (while condition expr ...)
       (let loop ()
          (if condition
             (begin
                expr ...
                (loop))
            (void))))
```

Desired expansion for first case above...

```
(let loop ()
 (if (< a 10)
        (begin
           (displayln a)
           (+= a 3)
           (loop))
        (void)))
```

Above, **expr** and **...** are inseparable!

A **show** macro

Have you ever done this?

```
System.out.println("x: " + x);
```

Let's write a **show** macro that will show both an expression and its value:

```
> (show x)
x: 7

> (show (+ x (* x 3)))
(+ x (* x 3)): 28
```

Solution:

```
(define-syntax-rule (show expr)
   (printf "~a: ~a\n" 'expr  expr))
```

Key point: we only need to specify the expression once, not twice!

Does Eclipse or IntelliJ give us a way to generate code like the **println** at top?

show, continued

Let's generalize **show** to allow any number of expressions:

```
> (show x (* x 2) (/ x 2))
x: 7
(* x 2): 14
(/ x 2): 7/2
```

The solution has a couple of new elements, but here it is:

```
(define-syntax show
   (syntax-rules ()
      [(show expr) (printf "~a: ~a\n" 'expr expr)]
      [(show expr exprs ...) (begin
                                          (show expr)
                                          (show exprs ...))]))
```

What can we understand about it?

and and **or**

Here are **and** and **or** from Dybvig. Note the multiple cases and recursive patterns. What do the underscores mean?

```
(define-syntax and
 (syntax-rules ()
  [(_) #t]
  [(_ e) e]
  [(_ e1 e2 e3 ...)
   (if e1 (and e2 e3 ...) #f)]))

(define-syntax or
 (syntax-rules ()
  [(_) #f]
  [(_ e) e]
  [(_ e1 e2 e3 ...)
   (let ([t e1]) (if t t (or e2 e3 ...)))]))
```

Industrial strength...

Here is **let** defined as a **lambda**:

```
(define-syntax let
 (lambda (x)
  (define ids?
   (lambda (ls)
    (or (null? ls)
        (and (identifier? (car ls))
             (ids? (cdr ls))))))
  (syntax-case x ()
   [(_ ((i e) ...) b1 b2 ...)
    (ids? #'(i ...))
    #'((lambda (i ...) b1 b2 ...) e ...)])))
```

```
(let ((x 3) (y 4)) (+ x y))
```

Notes:

Lists of identifiers and expressions are referenced separately with **i ...** and **b ...**

The procedure **ids?** ensures that all would-be identifiers are so.

second, third, ... tenth

SKIP!

From /usr/local/racket/collects/racket/list.rkt:

```
(define-syntax define-lgetter
  (syntax-rules ()
    [(_ name npos)
     (define (name L0)
       (if (list? L0)
         (let loop ([L L0] [pos npos])
           (if (pair? L)
             (if (eq? pos 1) (car L) (loop (cdr L) (sub1 pos)))
             (raise-arguments-error 'name
                                    "list contains too few elements"
                                    "list" L0)))
         (raise-argument-error 'name "list?" L0)))]))
```

```
(define-lgetter second  2)
(define-lgetter third   3)
(define-lgetter fourth  4)
(define-lgetter fifth   5)
(define-lgetter sixth   6)
(define-lgetter seventh 7)
(define-lgetter eighth  8)
(define-lgetter ninth   9)
(define-lgetter tenth   10)
```

```
> (define-lgetter fifty-third   53)
> (fifty-third (range 100))
52
```

The "capture" problem with macros

Imagine a repeat macro that repeatedly evaluates one or more forms some number of times:

```
> (repeat 3 (displayln 'hello))
hello
hello
hello

> (define x 0)
> (repeat 3 (displayln 'adding...) (+= x 5))
adding...
adding...
adding...
> x
15
```

Implementation:

```
(define-syntax-rule
   (repeat n expr ...)
      (let ([i 1])
         (while (<= i n)
            expr ...
            (+= i 1))))
```

"capture", continued

Consider the following usage,

```
> (define i 0)
> (repeat 3 (+= i 5))
```

and an expansion of the repeat:

```
(let ([i 1])
    (while (<= i 3)
      (+= i 5))
      (+= i 1)))
```

For reference:

```
(define-syntax-rule
   (repeat n expr ...)
      (let ([i 1])
         (while (<= i n)
            expr ...
            (+= i 1))))
```

Which uses of i refer to which definitions of i?!

The binding for i established by the let "captures" the i used in (+= i 5).

How could we avoid this capturing in the above case? (Don't peek!)

"capture", continued

We could use a less-common name than **i** for our loop counter:

```
(define-syntax-rule
  (repeat n expr ...)
    (let ([repeat-macro-loop-ctr 1])
      (while (<= repeat-macro-loop-ctr n)
        expr ...
        (+= repeat-macro-loop-ctr 1))))
```

Expansion of (repeat 3 (+= i 5)):

```
(let ([repeat-macro-loop-ctr 1])
      (while (<= repeat-macro-loop-ctr n)
        (+= i 5)
        (+= repeat-macro-loop-ctr 1))))
```

Ugly, but little chance of capture!

"capture", continued

Let's observe the bug that the "capture" of i creates:

```
> (define-syntax-rule
      (repeat n expr ...)
         (let ([i 1])
             (while (<= i n)
                 expr ...          ; becomes (+= i 5)
                 (+= i 1))))

> (define i 0)
> (repeat 3 (+= i 5))
> i
15
```

Your thoughts?

Hygienic macros

Scheme (and Racket) have "hygienic" macros!

There's considerable machinery involved but hygienic macros guarantee that names in expansions are processed in the scope of their origin.

For the expansion of **(repeat 3 (+= i 5))**, we had this naive view:

```
(let ([i 1])
   (while (<= i 3)
     (+= i 5))
     (+= i 1)))
```

In fact, when Racket expands a macro, it maintains information about the scope (and more) in which the macro call appeared. It knows that the **i** in **(+= i 5))** is not the same i as the others!

The bottom line is that in Racket, our **repeat** macro works as-is!

Hygienic macros, continued

Racket's macro system is said to be its crown jewel.

Two good papers:

Hygienic Macro Expansion by Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*

prl.khoury.northeastern.edu/img/kffd-tr-1986.pdf

Writing Hygienic Macros in Scheme with Syntax-Case by Kent Dybvig (1992)

legacy.cs.indiana.edu/ftp/techreports/TR356.pdf

Borrowing from Richard Feynman's "cataclysm question"...

Q: "If you could only tell me one thing about learning Racket, what would it be?"

A: "Understand hygienic macros."—Andrew Maurer-Oats

(Feynman's question: huffpost.com/entry/if-i-could-pass-on-one-se_b_3462335 and thecomplexityproject.com/richard-feynman-and-the-cataclysmic-question.)

Sidebar: A profusion of control structures

Here's an example of using the **do** special form. How would you describe its behavior?

```
> L
'(31 17 23 99)
> (do ([i 1 (add1 i)] [lst L (cdr lst)])
       ((empty? lst) (displayln "Done!") "do's value")
     (printf "~a: ~a\n" i (car lst)))
1: 31
2: 17
3: 23
4: 99
Done!
"do's value"
```

SKIP!

Could we implement it with a macro?

Too many control structures?

- For iteration we've got recursion, named **let**, 50+ variants of **for**, we wrote **while** ourselves, and there's a **do**-loop, too!

- For conditionals we've seen **if** and **cond** but two more are these:

  ```
  (when (zero? x) (displayln "It's zero!") ...)
  (unless (empty? L) (process L) ...)
  ```

- Cooking up a new control structure is no big deal in Lisp!

  What facilitates this?

  Syntax is simply forms enclosed in parentheses.

  The idea of special forms and implementation with *macros*.

- But there's little inertia against proliferation!

  Good or bad?

Macros in general

In general, programming language macros are source code rewriting rules, with varying levels of complexity and implementation approaches.

Macros originated with assembly language programming.

Many languages have macro facilities:

- C, C++, PL/I, Prolog, Julia, Rust, Scala, many others, and most Lisps.
- C is on the low end; Racket is on the high end
- Tools like `m4` can be used with any language (but line number issues...)
- Popularity waned with language designers but seems to be coming back.

Lots of pros and cons with macros!

- A syntactic sugar machine!
- Provide great flexibility
- Easy to get carried away
- Complicates support in debuggers, IDEs, etc.—just about every tool, in fact!

Higher-order Procedures

apply

The **apply** procedure can be used to apply a procedure to a list of values:

```
> (apply + '(7 2 3)) ; like (+ 7 2 3)
12
```

Problem: Write a procedure to compute the mean of a non-empty list of numbers.

```
> (mean '(95.3 90 100 92.8 73))
90.22
```

Solution:

```
(define (mean values)
   (/ (apply + values) (length values) 1.0))
```

apply, continued

There is a built-in max procedure:

```
> ,desc max
...
;    (max x ...+) -> real?
;      x : real?
```

Let's try it:

```
> (max '(3 1 5))
max: contract violation
  expected: real?
```

What's wrong?

max finds the maximum value of its argument**s**, not the maximum value in a list.

```
> (max 3 1 5)
5
```

apply, continued

How can we use **max** to find the largest value in the list **L**?

```
> L
'(10/9 7/5 9/2 1/4 14/3 5/13 3/2 13/6 2/7 1/11 11/12 6/5 3/7)
```

Solution:

```
> (apply max L)
14/3
```

Consider the contrast between the Racket and Haskell mechanisms (below) for finding a maximum value:

```
> :t max
max :: Ord a => a -> a -> a

> :t maximum
maximum :: (Foldable t, Ord a) => t a -> a    (like Ord a => [a] -> a)
```

apply, continued

Problem: Write a procedure to see if a list of numbers is in sorted order, either ascending or descending.

```
> (ordered? '(3 1 2))
#f
> (ordered? '(3 2 1))
#t
> (ordered? '(1 2 2))
#t
```

Solution:

```
(define (ordered? values)
   (or (apply <= values) (apply >= values)))
```

apply, continued

What does the following demonstrate about **apply**?

```
> (apply + 1 2 '(3))
6

> (apply + 1 2 3 empty)
6
```

apply can be called with individual values followed by a list of values.

map

Racket has a **map** procedure. Let's try it!

```
> (map string-length (string-split "a few words here"))
'(1 3 5 4)

> (map / '(2 3 4 5))
'(1/2 1/3 1/4 1/5)

> (map string (string->list "test"))
'("t" "e" "s" "t")

> (apply + (map (lambda (n) (* n n)) (range 1 6)))
55

(string-join (map string (map integer->char (range #x2654 #x265f))))
"♔♕♖♗♘♙♚♛♜♝♞"
```

map, continued

Problem: Write a procedure to see if all lists in a list of lists are the same length.

```
> (same-lengths? '((3 1) (5 8) (3 5)))
#t
> (same-lengths? '((3 1) (5 8) (3 5 2)))
#f
```

Solution:

```
(define (same-lengths? L)
   (apply = (map length L)))
```

map, continued

What do these examples tell us about map?

```
> (map + '(2 5 7) '(3 1 3))
'(5 6 10)

> (map cons '(a b c) (map list '(10 20 30)))
'((a 10) (b 20) (c 30))
```

We can map an N-ary procedure onto N lists!

What happens if lists are of unequal length?

```
> (map * '(1 2 3) '(4 5 6 7))
map: all lists must have same size
```

Challenge: Implement map!

andmap

What does **andmap** seem to do?

```
> (andmap char-numeric? (string->list "91571"))
#t

> (andmap char-numeric? (string->list "555-1212"))
#f
```

Do we really need **andmap**? Could we achieve the same result with **apply**?

```
> (apply and (map char-numeric? (string->list "91571")))
string:1:7: and: bad syntax
```

There's **ormap**, too.

More maps

Mapping isn't limited to lists!

```
> (define v (vector 7 1 3 5))
> (vector-map! (lambda (n) (* n n)) v)
'#(49 1 9 25)
> v
'#(49 1 9 25)

> (sequence-map (lambda (n) (* n n)) (in-naturals))
#<stream>
> (sequence-ref ^ 16)
256

> (time (sequence-ref (sequence-map (lambda (n) (* n n)) (in-naturals))
111111111))
cpu time: 115486 real time: 115567 gc time: 755
12345678987654321
```

Folding

Here's some documentation for **foldl**:

```
> ,desc foldl
...
;    (foldl proc init lst ...+) -> any/c
;      proc : procedure?
;      init : any/c
;      lst : list?
```

How does Racket's **foldl** differ from Haskell's?

It operates on one or more lists!

Speculate: What are the arguments for **proc**, the folding procedure?

Folding, continued

For reference:

```
;    (foldl proc init lst ...+)
;      proc : procedure?
;      init : any/c
;      lst : list?
```

Here's a procedure to help us explore folding:

```
> (define (show-args . args) (println args))
> (show-args 10 20 30)
'(10 20 30)
```

Let's try it with **foldl** and **foldr**:

```
> (foldl show-args '(1 10) '(2 20) '(3 30) '(4 40))
'(2 3 4 (1 10))
'(20 30 40 #<void>)

> (foldr show-args '(1 10) '(2 20) '(3 30) '(4 40))
'(20 30 40 (1 10))
'(2 3 4 #<void>)
```

What does it show?

The second argument for both **foldl** and **foldr** is the initial accumulator.

The folding procedure's last argument is the accumulator.

Folding, continued

For reference:

```
(foldl folding-proc init lst ...+)
(foldr folding-proc init lst ...+)
```

The folding procedure's last argument is accumulator.

Some simple folds:

```
> (foldl (lambda (val acm) (+ acm val)) 0 '(3 1 5 7))
16

> (foldr (lambda (val acm) (list* val val acm)) empty '(a b c))
'(a a b b c c)

> (foldr (lambda (val acm)
             (string-append acm (make-string val #\*)))
         "Stars: " '(3 1 5))
"Stars: *********"
```

Recall our binary tree insertion procedure:

```
> (print-tree (insert 1 (insert 3 (insert 7 (insert 5 empty)))))
5
  3
    1
  7
```

Let's use a fold to more easily test it:

```
> (print-tree (foldl (lambda (v tree) (insert v tree)) empty '(1 5 3 7 9 4 2)))
1
  5
    3
      2
      4
    7
      9
```

Would a `foldr` produce the same result? Would a `map` work?

A little fun...

What does the following fold produce?

```
> (foldr (lambda (a b acm) (cons (/ a b) acm)) '() '(1 2 3) '(4 5 6))
'(1/4 2/5 1/2)
```

Let's randomly generate a list of fractions for a **max** example.

```
> (let ([nums (range 1 15)])
          (foldr (lambda (a b acm) (cons (/ a b) acm))
              empty (shuffle nums) (shuffle nums)))
'(10/7 2/3 14/11 1 11/2 9/10 13/4 1/3 1 12/13 3/8 8 7/9 1/14)
```

Let's see if we like a second batch better:

```
> (let ([nums (range 1 15)])
          (foldr (lambda (a b acm) (cons (/ a b) acm))
              empty (shuffle nums) (shuffle nums)))
'(6/13 8/11 12/7 5/7 1/2 9/10 1/6 11/9 13 14/3 7/6 5/4 3/5 1/2)
```

group.rkt

Recall group.hs from assignment 4:

```
$ cat a4/group.1
able
academia
algae
carton
fairway
hex
hockshop
```

Execution:

```
% runghc group.hs a4/group.1
1 able
2 academia
3 algae
------
4 carton
------
5 fairway
------
6 hex
7 hockshop
```

Lets write a pure functional version in Racket using higher-order procedures, with no recursive code.

group.rkt, continued

First, let's write a procedure to read a file and produce dotted-pairs with line numbers and line contents:

```
> (make-pairs "a4/group.1")
'((1 . "able")  (2 . "academia") (3 . "algae") (4 . "carton")  (5 . "fairway")
  (6 . "hex") (7 . "hockshop"))
```

Here is make-pairs:

```
(define (make-pairs fname)
   (let* ([all-lines    (port->lines (open-input-file fname))]
           [lines          (filter (lambda (s) (> (string-length s) 0)) all-lines)]
           [line-nums  (range 1 (add1 (length lines)))]
           [pairs          (map (lambda (n line) (cons n line)) line-nums lines)])
       pairs))
```

At hand:

```
> (make-pairs "a4/group.1")
'((1 . "able")  (2 . "academia") (3 . "algae") (4 . "carton")  (5 . "fairway")
  (6 . "hex") (7 . "hockshop"))
```

Let's envision how a folding procedure, fp, might work:

```
> (fp '(7 . "hockshop") empty)
'((7 . "hockshop"))

> (fp '(6 . "hex") ^)
'((6 . "hex") (7 . "hockshop"))

> (fp '(5 . "fairway") ^)
'((5 . "fairway") "--------" (6 . "hex") (7 . "hockshop"))

> (fp '(4 . "carton") ^)
'((4 . "carton") "--------" (5 . "fairway") "--------" (6 . "hex") (7 . "hockshop"))
```

We want a folding procedure fp that behaves like this:

```
> (fp '(7 . "hockshop") empty)
'((7 . "hockshop"))

> (fp '(6 . "hex") ^)
'((6 . "hex") (7 . "hockshop"))

> (fp '(5 . "fairway") ^)
'((5 . "fairway") "--------" (6 . "hex") (7 . "hockshop"))

> (fp '(4 . "carton") ^)
'((4 . "carton") "--------" (5 . "fairway") "--------" (6 . "hex") (7 . "hockshop"))
```

Here's fp:

```
(define (fp pair acm)
   (define (first-same? s1 s2)
      (char=? (string-ref s1 0) (string-ref s2 0)))
   (if (empty? acm)
      (list pair)
      (if (first-same? (cdr pair) (cdar acm))
         (cons pair acm)
         (list* pair "--------" acm))))
```

Let's write a function to print an "entry":

```
> (print-entry '(7 . "hockshop"))
7 hockshop

> (print-entry "--------")
--------
```

```
(define (print-entry entry)
   (if (pair? entry)
      (printf "~a ~a\n" (car entry) (cdr entry))
      (displayln entry)))
```

And finally, a top-level procedure:

```
> (group "a4/group.1")
1 able
2 academia
3 algae
--------
4 carton
--------
5 fairway
...
```

```
(define (group fname)
   (let ([pairs (make-pairs fname)])
      (map print-entry (foldr fp empty pairs))
      (void)))
```

Procedures that produce procedures

negate

What does **negate** seem to do? (Not to be confused with **not**...)

```
> (map (negate negative?) '(3 -1 -8 5 -2))
'(#t #f #f #t #f)

> (list->string (filter (negate char-whitespace?)
                        (string->list "is it this or not?")))
"isitthisornot?"
```

Better example:

```
  (define mutable? (negate immutable?))
```

Further exploration:

```
> (define nz? (negate zero?))
> nz?
#<procedure:...racket/function.rkt:38:11>

> (nz? 5)
#t
```

Is **negate** a special form?

negate, continued

For reference:

```
> (define nz? (negate zero?))
> nz?
> (nz? 5)
#t
```

Problem: Write **negate**

Solution:

```
(define (my-negate p)
    (lambda (x) (not (p x))))
```

Testing:

```
> (define nz? (my-negate zero?))
> (map nz? '(-1 0 1))
'(#t #f #t)

> (define != (my-negate =))
> (!= 3 4)
my-negate.rkt:3:22: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 1
  given: 2
```

negate, continued

Our limited solution:

```
(define (my-negate p)
     (lambda (x) (not (p x))))
```

How can we negate a variadic procedure?

A general solution:

```
(define (my-negate proc)
   (lambda args
      (not (apply proc args))))
```

Speculate: What does **(lambda args ...)** do?

Experiment:

```
> ((lambda args (displayln args)) 1 2 3)
(1 2 3)
```

Testing:

```
> (define != (my-negate =))
> (!= 3 3 4)
#t
```

compose

Here's Racket's **compose**:

```
> (define next-to-last (compose car cdr reverse))
> (next-to-last (string->list "testing"))
#\n
```

Contrast with Haskell:

```
> nextToLast = head . tail . reverse
> nextToLast "testing"
'n'
```

Problem: Using **compose**, create **string-reverse**:

```
> (string-reverse "testing")
"gnitset"
```

Solution:

```
(define string-reverse
    (compose list->string reverse string->list))
```

curry

Here's **curry**:

```
> (define (add3 a b c) (+ a b c))
> (curry add3)
#<procedure:curried:add3>

> (^ 3)
#<procedure:curried:add3>

> (^ 4)
#<procedure:curried:add3>

> (^ 5)
12

> ((((curry add3) 10) 20) 30)
60
```

curry, continued

More experimentation:

```
> (procedure-arity add3)
3

> (procedure-arity (curry add3))
'(0 1 2 3)
```

Which inspires this...

```
> (map (curry add3 10 20) '(1 2 3))
'(31 32 33)

> (map (curry string-ref "mudge") (range 5))
'(#\m #\u #\d #\g #\e)

> ((((curry add3))))
#<procedure:curried:add3>
```

Tail recursion

Basics

Here's one way to recursively sum the numbers in a list in Python:

```
def sumnums(L):
   if L == []:
      return 0
   else:
      return L[0] + sumnums(L[1:])
```

If on some execution path, the very last computation performed by a function is to call itself, that call is said to be *tail-recursive*.

Do either of the execution paths for **sumnums** exhibit tail recursion?

Could we rewrite **sumnums** to be tail recursive?

Basics, continued

Here is a tail-recursive version of **sumnums**:

```
def sumnums_tr(L, total):
    if L == []:
        return total
    else:
        return sumnums_tr(L[1:], total + L[0])
```

Usage:

```
>>> sumnums_tr([1,2,3], 0)
6
```

How does it work?

Is this version easier to understand?

Does it present any advantages?

Basics, continued

If a function exhibits tail-recursion, those tail calls can be eliminated and replaced with a jump/goto.

Let's eliminate that tail call:

```
def sumnums_tr(L, total):
 top:
    if L == []:
        return total
    else:
         total = total + L[0]
         L = L[1:]
        goto top
```

Previous version:

```
def sumnums_tr(L, total):
    if L == []:
        return total
    else:
        return sumnums_tr(L[1:], total + L[0])
```

Does Python have a "goto" statement?

Basics, continued

Python doesn't have a goto* but we could just make an infinite loop:

```
def sumnums_loop(L, total):
    while True:
        if L == []:
            return total
        else:
            total = total + L[0]
            L = L[1:]
```

```
def sumnums_tr(L, total):
    if L == []:
        return total
    else:
        return sumnums_tr(L[1:], total + L[0])
```

What's the longest list we can process with each of the two versions?

Racket, and more generally, Scheme, guarantee that tail calls are turned into gotos/jumps.

R6RS:

> "Implementations of Scheme must be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar. "

What does that guarantee do for us?

A tail-recursive function will never exceed the stack space available!

sum-nums-tr

Problem: Make this **sum-nums** procedure tail-recursive:

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

Usage:

```
> (sum-nums-tr '(3 1 5 7) 0)
20
```

Solution:

```
(define (sum-nums-tr list total)
   (if (null? list)
      sum
      (sum-nums-tr (cdr list) (+ total (car list)))))
```

sum-nums-tr, continued

Let's compare traces for the two functions:

```
>(sum-nums '(3 1 5 7))
> (sum-nums '(1 5 7))
> >(sum-nums '(5 7))
> > (sum-nums '(7))
> > >(sum-nums '())
< < <0
< < 7
< <12
< 13
<16
16
```

```
>(sum-nums-tr '(3 1 5 7) 0)
>(sum-nums-tr '(1 5 7) 3)
>(sum-nums-tr '(5 7) 4)
>(sum-nums-tr '(7) 9)
>(sum-nums-tr '() 16)
<16
16
```

In the **sum-nums-tr** trace, note:

- Procedure call depth does not grow!
- We see evidence of only one return: `<16`

sum-nums-tr, continued

Here's the trace for **(sum-nums-tr '(3 1 5 7) 0)** again:

```
>(sum-nums-tr '(3 1 5 7) 0)
>(sum-nums-tr '(1 5 7) 3)
>(sum-nums-tr '(5 7) 4)
>(sum-nums-tr '(7) 9)
>(sum-nums-tr '() 16)
<16
16
```

What's the general pattern of computation we see?

- I see an element-by-element shifting of values from the list to the sum.
- A bit like a fold...

Let's time them:

```
> (define L (range 20000000))
> (time (sum-nums L))
cpu time: 1824 real time: 1830 gc time: 1443
199999990000000

> (time (sum-nums L))
cpu time: 1418 real time: 1482 gc time: 1091
...

> (time (sum-nums L))
cpu time: 1647 real time: 1652 gc time: 1284
...

> (time (sum-nums L))
cpu time: 1326 real time: 1345 gc time: 967
...
```

```
> (define L (range 20000000))
> (time (sum-nums-tr L 0))
cpu time: 118 real time: 168 gc time: 0
199999990000000

> (time (sum-nums-tr L 0))
cpu time: 95 real time: 98 gc time: 0
...

> (time (sum-nums-tr L 0))
cpu time: 101 real time: 105 gc time: 0
...

> (time (sum-nums-tr L 0))
cpu time: 102 real time: 106 gc time: 0
...
```

length

Here's an implementation of **length**. What makes it not tail-recursive?

```
(define (length lst)
   (if (empty? lst)
      0
      (add1 (length (cdr lst)))))
```

Problem: Write a tail-recursive version of **length**:

```
> (length-tr '(3 1 5 7) 0)
4
```

Solution:

```
(define (length-tr lst acc)
   (if (null? list) acc
      (length-tr (cdr lst) (add1 acc))))
```

A trace:

```
> (length-tr '(3 1 5 7) 0)
>(length-tr '(3 1 5 7) 0)
>(length-tr '(1 5 7) 1)
>(length-tr '(5 7) 2)
>(length-tr '(7) 3)
>(length-tr '() 4)
<4
4
```

length, continued

At hand:

```
(define (length-tr lst acc)
   (if (null? list) acc
      (length-tr (cdr lst) (add1 acc))))
```

It's a little icky to need to specify an accumulator: **(length-tr '(3 1 5 7) 0)**

Let's use a named **let** instead:

```
(define (length-nl lst)
   (let loop ([lst lst][acc 0])
      (if (null? list) acc
         (loop (cdr lst) (add1 acc)))))
```

Does the named **let** exhibit tail recursion?

Two "Wow!"s for me...

An interesting procedure

This procedure from page 50 in Dybvig was perhaps my first Wow! when learning Scheme:

```
(define count
   (let ([next 0])
      (lambda ()
         (let ([v next])
            (set! next (add1 next))
            v))))
```

Let's run it:

```
> (count)
0
> (count)
1
> (count)
2
> (count)
3
```

count, generalized...

Dybvig then generalizes it:

```
(define (make-counter)
   (let ([next 0])
      (lambda ()
         (let ([v next])
            (set! next (add1 next))
            v))))
```

Usage:

```
> (define c1 (make-counter))
> (define c2 (make-counter))
> (c1)
0
> (c1)
1
> (c1)
2
> (c2)
0
> (c2)
1
```

A WOW! in SICP

Here's an example from 2.1.3 What Is Meant by Data? in SICP:

```
(define (cons x y)
  (define (dispatch m)
    (cond
        ((= m 0) x)
        ((= m 1) y)))
  dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
```

What have we got here? (Don't peek!)

At hand:

```
(define (cons x y)
 (define (dispatch m)
  (cond
     ((= m 0) x)
     ((= m 1) y)))
 dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
```

Usage:

```
> (define L (cons 10 (cons 20 (cons 30 'empty))))

> (car L)
10

> (car (cdr L))
20

> (cdr (cdr (cdr L)))
'empty

> (cdr L)
#<procedure:dispatch>
```

"The single compound-data primitive pair, implemented by the procedures **cons**, **car**, and **cdr**, is the only glue we need."—SICP

If we had more time...

Study *continuations*, a way to capture the future of an expression.

Racket: A Programming-Language Programming Language
 —Robby Findler at Lambda Jam 2015
 See also:
 Other Languages in the Racket Environment on docs.racket-lang.org
 Beautiful Racket by Matthew Butterick

Look at some embedded Lisps:
 GNU Emacs Lisp
 Scheme in GIMP
 GNU's programming and extension language — GNU Guile
 www-isl.ece.arizona.edu/ACIS-docs
 (Type solid primitives in box, click Index, click on Solid Primitives link)

Look at Common Lisp, for contrast. (common-lisp.net)

Read *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*