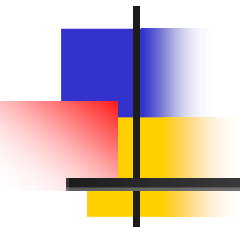
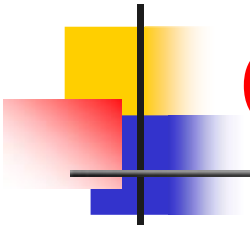


Racket Review

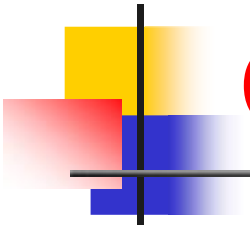
Robert “Corky” Cartwright
Department of Computer Science
Rice University



Core Racket Data

Built-in

- Booleans: **`true`**, **`false`** (also written **`#true`**, **`#false`**)
- Numbers including unbounded integers, rationals, inexact, complex (inexact and exact)
- Symbols: **`'A`**, … (almost any finite sequence of chars, other than blank, preceded by a tick (`'`) mark.)
- Strings: (excluded from subset used in class, homework, and exams)
- Lists: `empty` (also written **`'()`**), non-empty lists constructed using **`cons`** where first element is *any* data value and second is a list. Recall the library function **`list`** which has variable arity (polyvariadic?).
- Other built-in forms of data not included in our subset (vectors, …)



Core Racket Data cont.

Built-in: (cont.)

- Functions including primitive functions (like the functions for manipulating built-in data) and library functions (from *Intermediate Student with lambda* Racket language)
- Program defined functions denoted by lambda-abstractions.

Defined Functions:

- Via **`define`** operations (not the same as what can be defined directly as lambda-abstractions if execution behavior is considered)
- Generated by **`define-struct`** operation

Core Racket Syntax

Expressions:

- Constants for all built-in primitive data values; the term primitive values includes primitive functions but excludes lambda-abstractions.
- lambda-abstractions

  (lambda (*x*$_1$ … *x*$_n$) e) where n ≥ 0
- Applications of an expression ***M*** that evaluates to a primitive or defined function ***f*** (including lambda-abstractions) to 0 or more expressions:

  (*M* *e*$_1$ ... *e*$_n$) where n ≥ 0.

  Note that some applications (constructors to values) are values.
- Conditional expressions:

  (cond [*p*$_1$ *e*$_1$] ... [*p*$_n$ *e*$_n$]) where n > 0.

Core Racket Syntax cont

Expressions (cont.):

- Other Boolean expressions (which are *not* applications)

  (or e_1 **...** e_n**)** where n > 1

  (and e_1 **...** e_n**)** where n > 1

Operations that are not expressions:

(define ***s*** ***e*****)** where ***s*** is a symbol

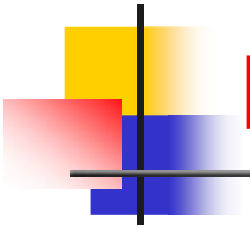
A program is a sequence of **define** operations followed by an expression with no *free* variables in the program (no references to variables that are not declared (in a **define** or lambda abstraction) and in scope). The Racket interactions windows supports the interleaving of **define** operations and expressions. Every expression is evaluated in the context of the **define** operations that precede it.

Some Abbreviations

- **`(define (`*`f`* x_1 … x_n`) e)`** abbreviates
  `(define `*`f`*` (lambda (`x_1 … x_n**`) e))`**

- List abbreviations
  `'( . . . )` where the preceding is a Racket list containing no symbols annotated with tick marks or identifiers designating constants like **`true`**, **`false`**, **`empty`**

  `Note: '( () )` is a list abbreviation; **`'(empty)`** is illegal in Core Racket; it evaluates to **`(list 'empty)`** in DrRacket teaching languages.

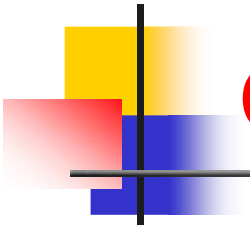


Reduction Semantics

Details are specified in the handout on Laws of Evaluation.

Big picture: The reduction process reduces expressions to *values*. Given a program, the value of the expression following the block of **`define`** operations is the result or answer of the program computation.

A *value* is a data constant or a lambda-expression.



Computation As Repeated Reduction

- Every Racket program execution is the evaluation of a given expression constructed from primitive or defined functions and variables (constants).
- Evaluation proceeds by repeatedly performing the leftmost possible reduction (simplification) until the resulting expression is a ***value*.**
- A ***value*** is the canonical textual representation of any constant (ignoring lazy lists). Numbers, booleans, symbols are all values.

Reduction for primitive functions

- A *reduction* is an atomic computational step that replaces some expression by a simpler expression as specified by a Racket evaluation rule (law). Every application of a basic operation to values yields a value (where run-time error is a special kind of value).
- Example reduction of expression built from primitive functions

```
    (* (+ 1 2) (+ 3 4))
=>  (* 3 (+ 3 4))
=>  (* 3 7)
=>  21
```

- Always perform leftmost reduction
- The following is **not** an atomic step, and so **not** a reduction

```
    (- (+ 1 3) (+ 1 3))  =  0
```

It is an equivalence in the transitive closure of reduction. (In pure reduction languages, every value reduces to itself!)

Full Reductions for Defined Functions

- Assume we defined the two functions

```
(define (area-of-box x) (* x x))
(define (half x) (/ x 2))
```

- Then Racket can perform these reductions

```
   (half (area-of-box 3))
=> ((lambda (x) (/ x 2)) (area-of-box 3))
=> ((lambda (x) (/ x 2)) ((lambda (x) (* x x)) 3))
=> ((lambda (x) (/ x 2)) (* 3 3))
=> ((lambda (x) (/ x 2)) 9)
=> (/ 9 2)
=> 4.5
```

- Reduction stops when we get to a value or an error

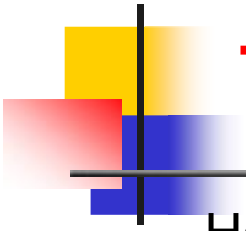
Simple Reductions for Defined Functions

- Restrict Core Racket to programs where all right-hand-sides are *values* and function definitions only appear using the syntax
  `(define (<fn-name> x1 … xn) <body>)`
- Then the (top-level) functions defined in definitions block program can be treated as ***values***, which greatly simplifies hand evaluation. Given

```
(define (area-of-box x) (* x x))
(define (half x) (/ x 2))
```

the expression

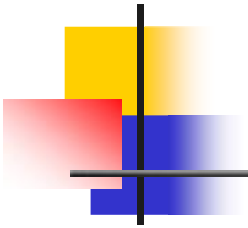
- **`(half (area-of-box 3))`**
- **`=> (half (* 3 3))`**
- **`=> (half 9)`**
- **`=> (/ 9 2)`**
- **`=> 4.5`**



The Design Recipe

How should we go about writing programs?

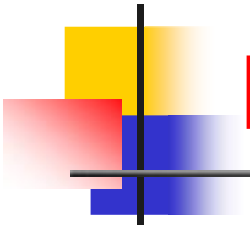
1. Analyze problem and define any requisite data types including examples and determine what functions should be in the "API" for the problem. Generate any **`struct`** definitions (code) and type definitions (comments) that you will need in writing the program.
2. State type and behavioral contracts and for each *function* in the API
3. Give examples of function uses and results
4. Select and instantiate a data-driven template for the function body; many are *degenerate.*
5. Write the function itself.
6. If the template uses generative (non-structural) show that it terminates.
7. Test it, and confirm that examples (tests) work.



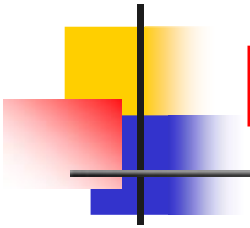
Informal Definitions of Types

Since type definitions are not embedded in Racket code, we provide them in program documentation written at the same level of rigor as informal mathematical proof (what is accepted as a proof in a proof-oriented math course). Most interesting type definitions *inductive.* The definition consists of a collection of clauses that either refer to known types, already defined types, and *data constructors* that take arguments of specified type, which may be the type being defined. Data constructors are declared using **`define-struct`**. The inductive type **`list`** is built-in to Racket but we often identify regular subsets of the form **`(list-of`** ***`alpha`*****`)`**. Sometimes we impose constraints of subset types that cannot be expressed using simple rules (called "context-free"). An example of such a type is ordered lists of numbers.

What Is Distinctive About Functional Programming

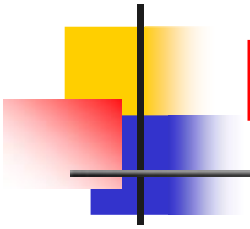


- A program is simply a collection of data definitions and pure function definitions which can be composed to describe a computation.
- Computation proceeds by performing leftmost reductions which embody obvious, nearly trivial rules.
- Well-written functional programs repeatedly decompose problems into simpler problems until we reach problems that can be solved by nearly trivial function definitions.
- Decomposition driven by structure of data being processed: *data-directed* design
- Most functional languages support functions as data values.



Programming Techniques

- Data Driven Program Design
- Help functions.
- Abstracting common patterns as help/library functions. Don't Repeat Yourself (DRY). Trivial repeats are OK.
- Simplest form of abstracting common patterns is **`let`** binding (codified as **`local`** in HTDP).
- Tail recursion with accumulators.
- Powerful functionals like **`map`**, **`filter`**, **`foldr`**, **`foldl`**
- Non-structural recursion (termination argument)
- Lazy evaluation (not part of Core Racket)



Different Forms of `let`

- Not emphasized in HTDP. Supported in Advanced teaching language.
- Important in practice but not difficult
- Three forms: **`let`**, **`let*`**, **`letrec`**
- HTDP **`local`** is alternate syntax for **`letrec`**
- Syntax is trivial and nearly identical for all three forms
- **`(let/ let*/letrec [(`*x*$_1$ *e*$_1$**`) … (`***x***$_n$ ***e***$_n$**`)] `***e***`)`**
- The three forms only differ in the scopes of the new local variables ***x***$_1$, . . ., ***x***$_n$

Different Forms of `let`

Scopes

- In **`let`**, the new variables are visible only in the body ***e***
- In **`let*`**, each new variable x_i is visible in all subsequent right-hand sides e_{i+1}, …, e_n as well as the body ***e***.
- In **`letrec`**, each new variable x_i is visible in e_1, …, e_n as well as the body ***e***.
- Most functional languages support **`let`** and **`letrec`**; Java supports **`let*`** in compound statements, and Algol-like languages support **`letrec`** in blocks (as does Java with regard to class members).