Racket Review

Robert “Corky” Cartwright
Department of Computer Science
Rice University

Racket Data

Built-in

- Booleans: **`true`**, **`false`** (also written **`#true`**, **`#false`**)
- Numbers including unbounded integers, rationals, inexact, complex (inexact and exact)
- Symbols: **`bA`**, … (almost any finite sequence of chars 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, …)

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)
- Functions denoted by lambda-abstractions.

Defined:

- 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

Program Syntax

Expressions:

- Constants for all built-in primitive data values; primitive excludes lambda-abstractions.
- lambda-abstractions
- **`(lambda (`**$\boldsymbol{x_1}$ **…** $\boldsymbol{x_n}$**`) e)`** where n ≥ 0
- Applications of a primitive or defined functions *f* (including lambda-abstractions) to 0 or more expressions:

  `(`$\boldsymbol{f\ e_1}$ **`...`** $\boldsymbol{e_n}$**`)`** where n ≥ 0.
- Conditional expressions:

  `(cond [`$\boldsymbol{p_1\ e_1}$**`] ... [`**$\boldsymbol{p_n\ e_n}$**`])`** where n > 0.

Program 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 neither defined or enclosing function parameter names). 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**

Reduction Semantics

Details are specified in the handout on Laws of Evaluation.

Big picture: Reduction 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 Is 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!)

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))   ←
   =>     (half (* 3 3))
   =>     (half 9)                 ←
   =>     (/ 9 2)
   =>     4.5
```

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

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 contract and purpose for each *function* in the API
3. Give examples of function use and result
4. Select and instantiate a 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. The inductive type list is built-in to Racket but we often identify regular subsets of the form **`(listOf`** ***`alpha`*****`)`**, which we sometimes write ***`alpha`*****`-list`**. The **`listOf`** form is preferred. 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.
- We will learn how to repeatedly decompose problems into simpler problems until we reach problems that can be solved by simple 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).
- Simplest form of abstracting common patterns is **`let`** binding (codified as **`local`** in HTDP).
- Tail recursion with accumulators.
- Powerful functional like **`map`**, **`filter`**, **`foldr`**, **`foldl`**
- Non-structural recursion (termination argument)
- Lazy evaluation

Different Forms of `let`

- Not emphasized in HTDP.
- 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 $x_1, \ldots, x_n$

Different Forms of `let`

Scopes

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