Evaluating Core Racket Programs

Corky Cartwright

Fall 2022

1 Syntax

We refer to the dialect of Racket used in class and the book as "Core Racket". It is a subset of the HTDP Intermediate Student with `lambda` language because it excludes all library functions except those explicitly mentioned below. Recall that the syntax of Core Racket is constructed from *atoms* and deep lists of *atom*s where an *atom* is either:

- a *value* which is a special Racket symbol that designating a data value, including various forms of numbers (like `0`, `1.5`, `#i1.0`, `1+1i`, `#i1.0+1.0i`); single-quoted symbols (like `'A`, `'x`, `'foo`), boolean values (`#true` or `#false`); the empty list `'()`; the essential primitive functions on values including `equals?`, `not`, `number?`, `exact?`, `inexact?`, `complex?`, `symbol?`, `procedure?` (which is true only for values that are functions), all math functions in the Racket library (including relational operators on numbers such as `=`, `<`, and `>`), the primitive functions associated with the built-in `cons` and `empty` *struct*s and the additional primitive functions associated with the built-in type lists, namely the functions `list?` and `list`.
- a *keyword* which are is one of following Racket symbols (without a leading quote) `define`, `cond` `if`, `else`, `and`, `or`, `lambda`, `true`, `false`, `empty`; or
- an *identifier* which is any *symbol* (without a leading single quote mark) that is not a keyword.

A *Racket expression*, is either a *value*, an *identifier*, or a deep list of *atoms* that has one of the following forms:

- a *primitive application*

  $(s_1 \ \ldots \ s_n)$

  where $n > 0$ and s_1 is a primitive function and s_2, ..., s_n are Racket expressions;
- a *conditional expression*

  (`cond` $(p_1 \ e_1) \ \ldots \ (p_n \ e_n)$)

  where $n \geq 0$ and p_1, ..., p_n are Racket expressions or the keyword `else`, and e_1, ..., e_n are Racket expressions;
- an *if-expression*

  (`if` $e_1 \ e_2 \ e_3$)

  where e_1, e_2, and e_3 are Racket-expressions;
- an *and-expression*

(`and` e_1 e_2)

where e_1 and e_2 are Racket-expressions;

- an *or-expression*

  (`or` e_1 e_2)

  where e_1 and e_2 are Racket-expressions;

- a *lambda-abstraction*

  (`lambda` *vars body*)

  where *var* is a list (not a deep list) of unique *identifier*s and *body* is a Racket expression; and

- a *general application*

  (f e_1 ... e_n)

  where $n > 0$ and f, e_1, ..., e_n are Racket expressions.

A *Racket definition* is

(`define` *id e*)

where *id* is an identifier and e is a Racket expression. The expression e is called the *right-hand-side* of the definition.

A *Racket program* is a (possibly empty) sequence of Racket definitions followed by Racket-expression e where (*i*) the right-hand sides of the definitions do not contain any free variables other than the identifier defined in the program and (*ii*) the Racket-expression e contains no free variables other than the identifiers defined in the preceding block of Racket definitions.

1.1 Abbreviations

A few identifiers serve as *abbreviations* for values. In particular, the identifiers `true`, `false`, and `empty` are abbreviations for the values `#true`, `#false`, `'()`, and `'()`, respectively. Similarly, `#t` anand `#f` are abbreviations for `#true` and `#false`.

1.2 Notes on Racket Syntax:

1. The `equal?` function is unreliable on comparing two functions (Why?). Languages in the ML family (Standard ML, OCaml, Haskell) use type checking to prevent such comparisons.

2. For any exact number x, the inexact value #ix is not the same as x. So the `equal?` function distinguishes them. The binary function `=` only applies to numbers and its definition for inexact arguments is subtle. (There is no simple way to handle such comparisons.)

3. We will not use the `=` function, inexact numbers, or complex numbers in Core Racket. We will similarly avoid `#t` and `#f`.

1.3 Notational Conventions for Reduction Rules

Italicized meta-variables will stand for pieces of syntax within Racket programs as follows:

- E, E_1, E_2, ... will stand for expressions.
- B, V, W, potentially augmented by numerical subscripts, will stand for values.
- Lower-case letters like f, g, h, n, s, t, u, v, x, y, z, potentially augmented by numerical subscripts, stand for variable names. Some examples of such meta-variables are f_1, g_2, h, n_5.
- N is a non-negative integer.

A sequence of items like $E_1 \ldots E_N$ indexed by consecutive integers beginning with 1, is empty if N is 0.

We will typically use the (possibly subscripted) meta-variables f, g, and h to refer to variables that are bound to functions and the meta-variables u, v, x, y, z to stand for variables that are bound to data values that are not functions. Sometimes we need meta-variables to refer to variables that can be bound to either functions or non-functions. We will try to mention when this situation can arise.

Recall that the program to be evaluated consists a (possibly empty) sequence of definitions followed by an expression constructed from program-defined and primitive functions and variables:

`(define` f_1 `...)` ... `(define` f_N `...)` E

Also recall that the variables f_i can be bound to either functions or non-functions. If the sequence of definitions is empty, the program expression to be evaluated is degenerate in the sense there are no program-defined variables.

1.4 Normalizing the Form of Definitions

In our reduction semantics for Core Racket, we interpret a function definition of the form

`(define (`f x_1`, ...` x_N`)` E`)`

to be an abbreviation for the variable definition

`(define` f `((lambda (`x_1 ... x_N`)` E`)`

The Racket compiler literally generates exactly the same code for each definition.

2 Expression Evaluation

Evaluating an expression means finding a value for that expression. We use a step-by-step process to repeatedly simplify an expression until it is so simple that it is a value. *Evaluating—/* a program means repeatedly evaluating the leftmost reducible or "stuck" sub-expression (any expression within the program). A "stuck" expression is one that cannot be reduced–even if all of its arguments are reduced to values, yet the irreducible expression is not a value. In a minimalist reduction semantics, error-generating sub-expressions are simply left as irreducible. Such a semantics could be completed by adding a collection of error values and adding rules for every expression form (including every primitive function) to propagate these errors to the top level where they are reported as aborting errors. But such a semantics would have far more rules than the minimalist alternative.

2.1 Stuck Expressions

Some syntactically well-formed Core Racket expressions, such as `(+ 'a 2)`, `(first empty)`, `(1 2)`, and `(/ 1 0)`, do not have a value according to our reduction rules because there is no reduction law that matches such an expression. For this reason, we say that evaluation of such expressions "sticks", which is a very simple, approach to formalizing the notion of a run-time error.

A stuck sub-expression is one where the arguments do not have the proper form to support a reduction step. A good example in most languages (including Racket) is division-by-zero. The expression

```
(/ 1 0)
```

cannot be reduced to a value and all of its arguments are values, so it is "stuck". If it occurs in leftmost reducible position, the program aborts. It is possible for an expression containing a control operation (`cond`, `if`, `and`, `or` in Core Racket) to have a stuck sub-expression without sticking on that stuck state. For example, the expression

```
(if true then 1 else (/ 1 0))
```

contains the stuck sub-expression `(/ 1 0)` but it reduces to `true` (abbreviating `#true`) because the stuck sub-expression never reaches leftmost reducible position.

In practice, run-time errors are easy to detect in manual Racket program evaluations because the leftmost sub-expression in position to be reduced is stuck.

2.2 The Reduction Laws

A law of the form $P \implies Q$ where P and Q are program fragments (expressions or sequences of expressions) means that P and Q have the same behavior; one can be substituted for the other without changing the meaning of the program. Hence, $\implies$ means exactly what it means in high school algebra. In addition, every law $P \implies Q$ has the property that Q is closer (measured in remaining reduction steps) to a value for P (assuming one exists) than P.

2.3 Values are values, are values, ...

Values are the answers produced by computations. Every value is also an expression, but it cannot be reduced because it is an answer.

Some examples are:

Value	Kind of Value
`0`	number (exact)
`1/3`	number (exact)
`0.3333333333333333`	number (inexact)
`6.023e23`	number (inexact)
`true`	boolean
`false`	boolean
`'piston`	symbol
`"Racket"`	string
`empty`	list
`(cons 'a empty)`	list
`(list 6 120)`	list
`+`	built-in function (primitive operation)
`(lambda (x) (+ x y))`	user-defined function (`lambda` expression)

bf Notes:

1. In nearly all functional languages, *functions* are values just like numbers are. The most common notation for functions as values are **lambda**-abstractions. More pedestrian languages do not view functions as values, forcing workarounds like representing functions explicitly as objects in Java.

2. In our hand-evaluations, no distinction is made between list abbreviations and the corresponding list construction using the `list` function. Hence, `(list)` is another way to write `empty`. Similarly no distinction is made and between `true` (`false`) and `#true` (`#false`). They all are values.

2.4 Conditionals

2.4.1 The Laws of `if`

When the test of an `if` expression is a value, the next step depends on whether the value is `true` or `false`. (If any other value appears in the test position for an `if` expression, that expression is a "stuck state".)

(`if true` E_2 E_3) $\implies$ E_2
(`if false` E_2 E_3) $\implies$ E_3

2.4.2 The Laws of `cond`

When the test of the first clause in a `cond` is a value, the next step depends on whether the value is `true` or `false`. Let W be any value that is not a boolean.

(`cond` [`false` E] ...) $\implies$ (`cond` ...)
(`cond` [`true` E] ...) $\implies$ E
(`cond` [`else` E] ...) $\implies$ E
(`cond` [W E] ...) is a stuck state
(`cond`) is a stuck state

In the absence of errors or non-termination, evaluation of a `cond` expression should result in selection of one of the clauses (and evaluation of its consequent expression.)

Here are some examples:

```
(cond [(> 10 12) (+ 7 8)] [else (* 6 4)]) ⟹ (cond [false (+ 7 8)] [else
   (* 6 4)])
                                           ⟹ (cond [else (* 6 4)])
                                           ⟹ (* 6 4)
(cond [true (+ 7 8)] [else (* 6 4)])  ⟹ (+ 7 8)
(cond ['foo (+ 7 8)] [else (* 6 4)])  ⟹ (+ 7 8)
```

2.4.3 The Laws of `or` and `and`

Let E be an arbitrary expression, B be an arbirary boolean value, and W is an value that is not a boolean. Then

(`or true` E) $\implies$ `true`
(`or false` B) $\implies$ B
(`or false` W) is a stuck state (`and false` E) $\implies$ `false`
(`and true` B) $\implies$ B (`and true` W) is a stuck state

2.5 The Laws of Application

Given an application consisting of values

(`(`V_1 V_2 ... V_N`)`)

(where V_1 must be primitive function or a `lambda`-abstraction) we apply different laws depending on whether the head value V_1 is a primitive function or a a `lambda`-abstraction. If the head value is not a function, the application is "stuck". Some "stuck" expressions are `(1 2)`, `(1)`, and `((cons 'a empty) empty)`.

2.5.1 Primitive applications

There is a large table of laws for directly reducing to a value the application of a primitive to a set of values. You know many of these rules from grammar school; the remainder are described (implicitly) in the course lecture notes and Racket Help Desk embedded in DrRacket. If you are uncertain about a result in the table, use the DrRacket **Intermediate Student with lambda** dialect to evaluate it. If you suspect an operation is primitive but do not know its name in Racket, ask a question on Piazza or try to find using the Racket Help Desk.

For instance, if U, U_1 ..., U_n are values, V is a list value, and W is a non-list value, then:

`(first (cons` U V`))` $\Longrightarrow$ U
`(first (list` U_1 ... U_n`))` $\Longrightarrow$ U_1
`(rest (cons` U V`))` $\Longrightarrow$ V
`(rest (list` U_1 ... U_n`)))` $\Longrightarrow$ `(list]` U_2 ... U_n`)`
`(cons?` `(cons` U V`))` $\Longrightarrow$ `true`
`(list?` `(cons` U V`))` $\Longrightarrow$ `true`
`(cons?` W`)` $\Longrightarrow$ `false`
`(list?)` W`)` $\Longrightarrow$ `false`

Examples:

```
(first (cons 1 empty)) ⟹ 1
(rest (cons 1 empty)) ⟹ empty
(cons? 1) ⟹ false
(cons? (cons 1 empty)) ⟹ true
(+ 1 2) ⟹ 3
```

If a primitive operation is applied to illegal inputs, then the primitive application is *stuck*. Some sticking expressions are `(first empty)`, `(rest 1)`, and `(+ empty 2)`. A stuck expression is a special form of reducible expression that aborts the computation with an error message describing why the application is illegal.

2.5.2 `lambda` Applications

If the head value in an application is a `lambda` expression

(`lambda` (x_1 ... x_N) E)

where x_1, ..., x_N are variable names and E is an expression, then the following rule specifies a potential next step in evaluating the application:.

((`lambda` (x_1 ... x_N) E) V_1 ... V_N) $\Longrightarrow$ $E_{[V_1 \text{ for } x_1]...[V_N \text{ for } x_N]}$

where the notation $E_{[\overline{V} \text{ for } \overline{x}]}$ means E with all free occurrences[1] of variables in $\overline{x}$ *safely* replaced by the corresponding *values* in $\overline{V}$. Obviously $\overline{x}$ and $\overline{V}$ must have the same arity N.

This rule is a special case of the most famous reduction-rule in the lambda-calculus called beta-reduction, which is identical except that it does not restrict the argument expressions $V_1 \ldots V_N$ to values. Our restriction is often called *beta-value-reduction*. In the reduction semantics for Racket, all of the substitutions performed in beta-value-reduction steps, except for one rare corner case discussed in Section 3.1, are *safe*.

3 Evaluating definitions

The preceding section gives laws for evaluating Scheme expressions in the absence of program definitions. But Scheme programs have the form

```
(define n_1 E_1)
(define n_2 E_2)
...
(define n_N E_N)
E
```

where n_1, n_2, ..., n_N are names and E_1, E_2, ..., E_N, E are expressions using Scheme primitives and the defined names n_1, n_2, ..., n_N. The expression E is called the body of the program and each expression E_k is called the body of the definition (`define` n_k E_k).

If the definition bodies E_k are all values

```
(define n_1 V_1)
(define n_2 V_2)
...
(define n_N V_N)
E
```

then we evaluate the expression E as described above with the added provision that the names n_1, n_2, ..., n_N have values V_1, V_2, ..., V_N, respectively. This situation prevails if all top-level `define` constructs bind variables to functions denoted by `lambda`-expressions. Hence, the only Racket programs that require evaluating the right-hand sides of `define` constructs are those with definitions (in the form of `define` constructs) that bind variables to expressions that are not functions! These right-hand-sides are reduced in sequential (leftmost) order.

These laws force us to evaluate the bodies of all definitions in sequential order before evaluating the body of the program. Note that the body of a program with a prefix that is a sequence of `define` declarations may contain variables that are free in the body. Their bindings must be provided by preceding `define` declarations; otherwise the program would be ill-formed because some variables would be free with respect to the entire program. DrRacket checks for such free variables before attempting to evaluate a program. Of course, such checking could be deferred and performed during program evaluation, but this convention is generally considered bad form on the part of the language implementors.

[1] An occurrence of a variable is a *binding occurrence* if it appears as the variable defined in a Racket `define` construct or a parameter in a `lambda`-expression. A variable occurrence that is not a binding occurrence is called a *use occurrence*. In practice, most variable occurrences are use occurrences. A use occurrence of a variable is *free within a particular program fragment* P (expression or whole program) iff it is not enclosed by a binding occurrence of the same variable name in P.

3.1 Safe Substitution in Racket Programs

The only case where unsafe substitutions can occur in the reduction semantics for Racket appears in the context of reducing programs with definitions. This case is rare but understanding the possible pathologies in general beta-reduction is important, because beta-reduction is a very common optimizing transformation (often called "inlining") performed by programmers during program development and by optimizing compilers in generating efficient machine code. The reduction semantics for Racket deftly avoids these pathologies except in one special case.

The potential pathologies in substitution can be systematically eliminated by using a process called *safe substitution*, which is an elaboration of ordinary (sometimes called "raw") substitution. In reducing Racket programs to answers, we can simply use raw substitution, except for one pathological case that rarely occurs in real programs, which will discuss now.

In the safe substitution $E_{[V \text{ for } x]}$ variables bound variables within E *must be renamed* if they clash with free variables in $V_1, \ldots, V_N$. This anomaly is called *capturing free variables* and it is the bane of existence for logicians and programming language theorists. Many formal systems proposed for reasoning about mathematical domains and programs have failed to prevent this pathology, breaking the soundness (correctness) of those systems.

The concept of capturing a free occurrence of a variable sounds obscure but it is actually very intuitive. It can only happen in transforming a program text in Racket (or other programming languages) if there are *multiple* variables with the same name. Every binding occurrence of a variable x has a *scope*, which is simply the portion of the program text where that binding is active (visible). If the expression N that is being substituted for a variable y contains an occurrence of a free variable z then the meaning of the variable z will change after the substitution if the binding occurrence corresponding to the occurrence of z changes as a result of the substitution. If an occurrence of the "target" variable x_i being replaced occurs in within the scope of a binding occurrence of z, capture happens! Fortunately, this event never happens in the reduction process in Racket except for one obscure corner case was overlooked by the authors of HTDP. If the binding occurrence of a variable x is a top-level `define` statement in the program being reduced, free occurrences of a different variable named x can be captured by the top-level binding. The last example in the block of examples below shows this rare pathology.

On the other hand, capture can easily happen when a function application is expanded "in line" to eliminate the overhead of the function call and perhaps enable more optimization. Such transformations are commonly performed as part of the program development process. They are also increasingly used in highly optimizing compilers. A good way to prevent "capture" from possibly happening is to avoid introducing a new binding occurrence of a variable x within the scope of another variable with the same name x. Compilers typically use a form of notation for variable occurrences (called *static distance representation*) that makes every variable name unique within any given context eliminating the possibility of capturing free variables.

Study the following instances of beta-value-reduction (as performed in the reduction semantics for Racket) carefully.

Examples:

- `((lambda (x) (+ x x)) 7)` $\Longrightarrow$ `(+ 7 7)`
- `((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y)))`
  $\not\Longrightarrow$ `(lambda (x) ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) x)))`
- `((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y)))`
  $\Longrightarrow$ `(lambda (z) ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) z)))`
- `(define free 17)`
  `((((lambda (x) (lambda (free) x)) (lambda (y) free)) 0) 0)`

$\not\Longrightarrow$
`(define free 17)`
`(((lambda (free) (lambda (y) free)) 0) 0)` $\Longrightarrow$ `0`

- `(define free 17)`
  `((((lambda (x) (lambda (free) x)) (lambda (y) free)) 0) 0)`
  $\Longrightarrow$
  `(define free 17)`
  `(((lambda (free1) (lambda (y) free)) 0) 0)` $\Longrightarrow$ `17`

3.2 Rules for define-struct

For the sake of simplicity (and sanity), we will assume that all of the `define-struct` definitions for a program appear in a prelude that is preprocessed to augment the set of program values and primitive functions. Every `define-struct` definition

`(define-struct` n `(`n_1 ... n_k`))`

simply augments the primitive operations of Racket by the k-ary constructor `make-`n, the unary boolean function n`?`, and the accessors n`-`n_1, ..., n`-`n_k. Hence, the Racket program following such a prelude is evaluated like any other Racket program except of the expanded set of primitive operations.

3.3 Rules for `local`

To evaluate programs containing `local`, we need to introduce the concept of *promotion* (also called *flattening*). Given an expression of the form

(`local` [(`define` n_1 E_1) ... (`define` n_N E_N)] E)

we first convert the local definitions of the names $n_1, \ldots, n_N$ to global definitions of new names $n'_1, \ldots, n'_N$, renaming all bound occurrences of $n_1, \ldots, n_N$. Then we evaluate the transformed expression E in the context of the new definitions. This conversion process is called the *promotion* or *flattening* of a `local` expression. The new names $n'_1, \ldots, n'_N$ must be chosen so that they are distinct from all other names in the program.

Let

(`define` n_1 V_1)
...
(`define` n_{k-1} V_N)
E

be a program where the program body E has the form

$\mathcal{C}[L]$

where L is an expression

`(local [(define` n_1 E_1`)` ... `(define` n_N E_N`)] E)`

enclosed in the surrounding program text $\mathcal{C}[\,]$ to form the expression E. Assume that no subexpressions in E to the left of the subexpression L can be reduced. Hence, L is the leftmost expression in the entire program that can be reduced. In this case, the surrounding text $\mathcal{C}[\,]$ is called the *evaluation context* of L.

Using the notation introduced above, we can describe the *promotion step* reducing the program by the following rule:

(define n_1 V_1)
...
(define n_{k-1} V_N)
$\mathcal{C}$[(local [(define n_1 E_1) ... (define n_N E_N)] E)]

$\Longrightarrow$

(define n_1 V_1)
...
(define n_{k-1} V_N)
(define n'_1 $E_{1[n'_1 \text{ for } n_1] \ldots [n'_N \text{ for } n_N]}$)
...
(define n'_N $E_{N[n'_1 \text{ for } n_1] \ldots [n'_N \text{ for } n_N]}$)
$\mathcal{C}[E_{[n'_1 \text{ for } n_1] \ldots [n'_N \text{ for } n_N]}]$

In other words, we replaced L by the body of L with $n_1, \ldots, n_N$ renamed and we added appropriate definitions for the new names in the sequence of `define` statements preceding the program body. Note that free occurrences of the names $n_1, \ldots, n_N$ must be renamed in the expressions $E_1, \ldots, E_N$, as well as E.

In practice, the rules for reducing local are rather messy so none of our examples will involve using these rules. The key take-away here is that constructions built using local have a straightforward meaning that is simple in principle if messy in practice.