Evaluating Core Racket Programs

Corky Cartwright

Fall 2020

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 *atoms* 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*

    $(\texttt{cond} \ (p_1 \ e_1) \ \ldots \ (p_n \ e_n))$

    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*

    $(\texttt{if} \ e_1 \ e_2 \ e_3)$

    where e_1, e_2, and e_3 are Racket-expressions;

- an *and-expression*

$$(\texttt{and}\ e_1\ e_2)$$

where e_1 and e_2 are Racket-expressions;

- an *or-expression*

$$(\texttt{or}\ e_1\ e_2)$$

where e_1 and e_2 are Racket-expressions;

- a *lambda-abstraction*

$$(\texttt{lambda}\ vars\ body)$$

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

- a *general application*

$$(f\ \ e_1\ \ldots\ e_n)$$

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

A *Racket definition* is

$$(\texttt{define}\ id\ e)$$

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 Syntaxd:

1. The `equal?` function is unreliable on comparing two functions (why?); languages in the ML family use type checking to prevent such comparisons.

2. For any exact number x, the inexact value $\#\text{i}x$ 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 inexact numbers in Core Racket. We will similarly avoid `#t` and `#f`.

1.3 Notational Conventions for Reduction Rules

Italicized metavariables 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 metavariables 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) metavariables f, g, and h to refer to variables that are bound to functions and the metavariables u, v, x, y, z to stand for variables that are bound to data values that are not functions. Sometimes we need metavariables 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 subexpressions 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 propogate 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-or-stuck 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 left-most reducible-or-stuck position.

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

2.2 The Reduction Laws

A law of the form

$$P => Q$$

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, $=¿$ means exactly what it means in high school algebra. In addition, every law

$$P => Q$$

has the property that Q is "closer" to a value (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)

Note: in hand-evaluations, no distinction is made between list abbreviations and the corresponding list construction using `list`. Similarly, no distinction is made between `true` and `#true` and between `false` and `#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) =¿ E_2
(if false E_2 E_3) =¿ 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] ...) =¿ (cond ...)
(cond [true E] ...) =¿ E
(cond [else E] ...) =¿ 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) =¿ true
(or false B) =¿ B
(or false W) is a stuck state (and false E) =¿ false
(and true B) =¿ B (and true W) is a stuck state
```

2.5 The Laws of Application

Given an application consisting of values

$$(V_1 \ V_2 \ \ldots \ V_N)$$

we apply different laws depending on whether the head value V_1 is a primitive function or a a `lambda` expression. 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 decribed (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.

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)) =¿ U
(first (list U_1 ... U_n)) =¿ U_1
(rest (cons U V)) =¿ V
(rest (list U_1 ... U_n))) =¿ (list] U_2 ... U_n)
(cons?  (cons U V)) => true
(cons?   W) =¿ 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)`.

2.5.2 `lambda` applications

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

$$(\texttt{lambda} \ (x_1 \ \ldots \ x_N) \ E)$$

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

$$((\texttt{lambda} \ (x_1 \ \ldots \ x_N) \ E) \ V_1 \ \ldots \ V_N) =_{\unicode{0xBF}} E_{[V_1 \text{ for } x_1]\ldots[V_N \text{ for } x_N]}$$

where the notation $E_{[V \text{ for } x]}$ means E with all free occurrences[1] of x *safely* replaced by *Value*. (Locally bound variables in 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.

Examples:

```
((lambda (x) (+ x x)) 7) =¿ (+ 7 7)
((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y)))
     ≠ (lambda (x) ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) x)))
((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y)))
     =¿(lambda (z) ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) z)))
```

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 n₁ E₁)
(define n₂ E₂)
...
(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 $(\texttt{define} \ n_k \ E_k)$.

If the definition bodies E_k are all values

```
(define n₁ V₁)
(define n₂ V₂)
...
(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.

3.1 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

---

[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 use occurrence of a variable is *free within a particular program fragment P* (expression or whole program) iff it not enclosed by a binding occurence of the same variable name in P.

```
(define-struct n (n₁ ... 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.

3.2 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 n₁ E₁) ... (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 n₁ V₁)
...
(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 n₁ E₁) ... (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 n₁ V₁)
...
(define n_{k-1} V_N)
C[(local [(define n₁ E₁) ... (define n_N E_N)] E)]

=>

(define n₁ V₁)
...
(define n_{k-1} V_N)
(define n'₁ E₁[n'₁ for n₁] ... [n'_N for n_N])
...
(define n'_N E_N[n'₁ for n₁] ... [n'_N for n_N])
C[E[n'₁ for n₁] ... [n'_N 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 occurences of the names $n_1, \ldots, n_N$ must be renamed in the expressions $E_1, \ldots, E_N$, as well as E.