Introduction to Racket, a dialect of LISP: Expressions and Bindings

LISP: designed by John McCarthy, 1958 published 1960

LISP: implemented by Steve Russell, early 1960s

LISP: LISt Processing

- McCarthy, MIT artificial intelligence, 1950s-60s
  - Advice Taker: represent logic as data, not just program
  - Needed a language for:
    - Symbolic computation
    - Programming with logic
    - Artificial intelligence
    - Experimental programming
  - So make one!
Scheme

- Gerald Jay Sussman and Guy Lewis Steele (mid 1970s)
- Lexically-scoped dialect of LISP that arose from trying to make an “actor” language.
- Described in amazing “Lambda the Ultimate” papers (http://library.readscheme.org/page1.html)
  - Lambda the Ultimate PL blog inspired by these: http://lambda-the-ultimate.org
- Led to Structure and Interpretation of Computer Programs (SICP) and MIT 6.001 (https://mitpress.mit.edu/sicp/)

Racket

- Grandchild of LISP (variant of Scheme)
  - Some changes/improvements, quite similar
- Developed by the PLT group (https://racket-lang.org/people.html), the same folks who created DrJava.
- Why study Racket in CS251?
  - Clean slate, unfamiliar
  - Careful study of PL foundations (“PL mindset”)
  - Functional programming paradigm
    - Emphasis on functions and their composition
    - Immutable data (lists)
  - Beauty of minimalism
  - Observe design constraints/historical context

Expressions, Values, and Bindings

- Entire language: these three things

- Expressions have evaluation rules:
  - How to determine the value denoted by an expression.

- For each structure we add to the language:
  - What is its syntax? How is it written?
  - What is its evaluation rule? How is it evaluated to a value (expression that cannot be evaluated further)?

Values

- Values are expressions that cannot be evaluated further.

- Syntax:
  - Numbers: 251, 240, 301
  - Booleans: #t, #f
  - There are more values we will meet soon (strings, symbols, lists, functions, ...)

- Evaluation rule:
  - Values evaluate to themselves.
Addition expression: syntax

Add two numbers together.

Syntax: \((+ \ e_1 \ e_2)\)
- Every parenthesis required; none may be omitted.
- \(e_1\) and \(e_2\) stand in for any expression.
- Note prefix notation.

Examples:
- \((+ \ 251 \ 240)\)
- \((+ \ (+ \ 251 \ 240) \ 301)\)
- \((+ \ #t \ 251)\)

Addition expression: evaluation

Syntax: \((+ \ e_1 \ e_2)\)

Evaluation rule:
1. evaluate \(e_1\) to a value \(v_1\)
2. evaluate \(e_2\) to a value \(v_2\)
3. Return the arithmetic sum of \(v_1 + v_2\).

Addition: dynamic type checking

Syntax: \((+ \ e_1 \ e_2)\)

Evaluation rule:
1. evaluate \(e_1\) to a value \(v_1\)
2. evaluate \(e_2\) to a value \(v_2\)
3. If \(v_1\) and \(v_2\) are both numbers then
   return the arithmetic sum of \(v_1 + v_2\).
4. Otherwise, a type error occurs.

Evaluation Assertions Formalize Evaluation

The evaluation assertion notation \(e \downarrow v\) means "\(e\) evaluates to \(v\)".

Our evaluation rules so far:
- value rule: \(v \downarrow v\) (where \(v\) is a number or boolean)
- addition rule:
  \[\text{if } e_1 \downarrow v_1 \text{ and } e_2 \downarrow v_2 \text{ and } v_1 \text{ and } v_2 \text{ are both numbers and } v \text{ is the sum of } v_1 \text{ and } v_2 \text{ then } (+ e_1 e_2) \downarrow v\]
Evaluation Derivation in English

An evaluation derivation is a “proof” that an expression evaluates to a value using the evaluation rules.

\[(+ 3 (+ 5 4)) \downarrow 12\] by the addition rule because:

- \(3 \downarrow 3\) by the value rule
- \((+ 5 4) \downarrow 9\) by the addition rule because:
  - \(5 \downarrow 5\) by the value rule
  - \(4 \downarrow 4\) by the value rule
  - \(5\) and \(4\) are both numbers
  - \(9\) is the sum of \(5\) and \(4\)
- \(3\) and \(9\) are both numbers
- \(12\) is the sum of \(3\) and \(9\)

More Compact Derivation Notation

\[v \downarrow v\ (value\ rule)\]
where \(v\) is a value
(number, boolean, etc.)

\[e1 \downarrow v1\ (addition\ rule)\]
\[e2 \downarrow v2\ (addition\ rule)\]

\[ (+ e1 e2) \downarrow v\]
side conditions of rules

Where \(v1\) and \(v2\) are numbers and \(v\) is the sum of \(v1\) and \(v2\).

Errors Modeled by “Stuck” Derivations

How to evaluate
\[(+ \#t (+ 5 4))?\]
\[\#t \downarrow \#t\ (value)\]
\[5 \downarrow 5\ (value)\]
\[4 \downarrow 4\ (value)\]
\[(+ 5 4) \downarrow 9\]
Stuck here. Can’t apply (addition) rule because \#t is not a number

How to evaluate
\[(+ 3 (+ 5 \#f))?\]
\[3 \downarrow 3\ (value)\]
\[5 \downarrow 5\ (value)\]
\[\#f \downarrow \#f\ (value)\]
Stuck here. Can’t apply (addition) rule because \#f is not a number

Special Cases for Addition

The addition operator + can take any number of operands.

- For now, treat \((+ e1 e2 \ldots en)\) as \((+ (+ e1 e2) \ldots en)\)
  E.g., treat \((+ 7 2 -5 8)\) as \((+ (+ 7 2) -5) 8)\)
- Treat \((+ e)\) as \(e\)
- Treat \((+ )\) as \(0\) (or say \((+ ) \downarrow 0)\)
Other Arithmetic Operators

Similar syntax and evaluation for

- * quotient remainder

except:

- Second argument of /, quotient, remainder must be nonzero
- Result of / is a rational number (fraction)
- quotient and remainder take exactly two arguments; anything else is an error.
- (- e) is treated as (- 0 e)
- (/ e) is treated as (/ 1 e)
- (*) evaluates to 1.
- (/) and (-) are errors.

Relation Operators

The following relational operators on numbers return booleans: < <= = >= >

For example:

\[
\begin{align*}
& e_1 \downarrow v_1 \\
& e_2 \downarrow v_2 \\
& < e_1 e_2 \downarrow v
\end{align*}
\]

Where \(v_1\) and \(v_2\) are numbers and \(v\) is #t if \(v_1\) is less than \(v_2\)
or #f if \(v_1\) is not less than \(v_2\)

Conditional (if) expressions

Syntax: (if e1 e2 e3)

Evaluation rule:

1. Evaluate e1 to a value v1.
2. If v is not the value #f then
   return the result of evaluating e2
   otherwise
   return the result of evaluating e3
Your turn

Use evaluation derivations to evaluate the following expressions

(\text{if} (< 8 2) (+ #f 5) (+ 3 4))

(\text{if} (+ 1 2) (- 3 7) (/ 9 0))

(+ (\text{if} (< 1 2) (* 3 4) (/ 5 6)) 7)

Expressions vs. statements

If expressions can go anywhere an expression is expected:

(\text{if} (< 9 (- 251 240))
 (* 3 (+ 4 5))
 (+ 6 (* 7 8)))

(+ 4 (* (\text{if} (< 9 (- 251 240)) 2 3) 5))

Note: this is an expression, not a statement. Do other languages you know have conditional expressions in addition to conditional statements? (Many do! Java, JavaScript, Python, ...)

If expressions: careful!

Unlike earlier expressions, not all subexpressions of if expressions are evaluated!

(\text{if} (> 251 240) 251 (/ 251 0))

(\text{if} #f (+ #t 251) 251)

Environments: Motivation

Want to be able to name values so can refer to them later by name. E.g.;

(define x (+ 1 2))

(define y (* 4 x))

(define diff (\text{-} y x))

(define test (< x diff))

(if test (+ (* x y) diff) 17)
Environments: Definition

- An environment is a sequence of bindings that associate identifiers (variable names) with values.
  - Concrete example:
    ```
    num → 17, absoluteZero → -273, true → #t
    ```
  - Abstract Example (use id to range over identifiers):
    ```
    id1 → v1, id2 → v2, ..., idn → vn
    ```
  - Empty environment: Ø

- An environment serves as a context for evaluating expressions that contain identifiers.

  ```
  (define id e)
  ```

  Syntax: (+ e1 e2)

  **Addition: evaluation with environment**

  Evaluation rule:
  1. Evaluate e1 in the current environment to a value v1
  2. Evaluate e2 in the current environment to a value v2
  3. If v1 and v2 are both numbers then return the arithmetic sum of v1 + v2.
  4. Otherwise, a type error occurs.

  ```
  (define id e)
  ```

  Syntax: (define id e)

  Evaluation rule:
  1. Evaluate e to a value v in the current environment.
  2. Produce a new environment that is identical to the current environment, with the additional binding id → v at the front.
Environments: Example

\( \text{env0} = \emptyset \)

(define x (+ 1 2))

\( \text{env1} = x \rightarrow 3, \emptyset \) (abbreviated \( x \rightarrow 3 \), can write as \( x \rightarrow 3 \), . in text)

(define y (* 4 x))

\( \text{env2} = y \rightarrow 12, x \rightarrow 3 \) (most recent binding first)

(define diff (- y x))

\( \text{env3} = \text{diff} \rightarrow 9, y \rightarrow 12, x \rightarrow 3 \)

(define test (< x diff))

\( \text{env4} = \text{test} \rightarrow \text{#t}, \text{diff} \rightarrow 9, y \rightarrow 12, x \rightarrow 3 \)

Environment here is still \( \text{env4} \)

Example Derivation with Environments

Suppose \( \text{env4} = \text{test} \rightarrow \text{#t}, \text{diff} \rightarrow 9, y \rightarrow 12, x \rightarrow 3 \)

\( \text{test} \underbrace{\uparrow \# \text{env}}_{\text{varref}} \rightarrow \text{#t} \)

\( \underbrace{x \uparrow \text{env}}_{\text{varref}} \rightarrow 3 \)

\( \underbrace{5 \uparrow \text{env}}_{\text{value}} \rightarrow 5 \)

\( \underbrace{(* \times 5) \uparrow \text{env}}_{\text{value}} \rightarrow 15 \)

\( \underbrace{\text{diff} \uparrow \text{env}}_{\text{varref}} \rightarrow 9 \)

\( \underbrace{(+ (* x 5) \text{diff}) \uparrow \text{env}}_{\text{varref}} \rightarrow 24 \)

Evaluation Assertions & Rules with Environments

The evaluation assertion notation \( e \# env \downarrow v \) means "Evaluating \( e \) in environment \( env \) yields value \( v \)."

\( \text{id} \ # \ env \downarrow v \) (varref)

where \( id \) is an identifier and \( id \rightarrow v \) is the first binding in \( env \) for \( id \).

Only this rule actually uses \( env \); others just pass it along.

\( e1 \ # \ env \downarrow v1 \)

\( e2 \ # \ env \downarrow v2 \) (addition)

\( (+ e1 e2) \ # \ env \downarrow v \)

Where \( v1 \) and \( v2 \) are numbers and \( v \) is the sum of \( v1 \) and \( v2 \).

\( v \ # \ env \downarrow v \) (value)

where \( v \) is a value (number, boolean, etc.)

\( e1 \ # \ env \downarrow \text{#f} \)

\( e3 \ # \ env \downarrow v3 \) (if false)

\( (if e1 e2 e3) \ # \ env \downarrow v3 \)

\( (if \text{false}) \)

\( \text{only this rule actually uses env; others just pass it along} \)

Racket Identifiers

- Racket identifiers are case sensitive. The following are four different identifiers: ABC, Abc, aBc, abc
- Unlike most languages, Racket is very liberal with its definition of legal identifiers. Pretty much any character sequence is allowed as identifier with the following exceptions:
  - Can’t contain whitespace
  - Can’t contain special characters ((),[]",',`;#\"
  - Can’t have same syntax as a number
- This means variable names can use (and even begin with) digits and characters like 1@5%^&*,-.<>?/. E.g.:
  - myLongName, my_long__name, my-long-name
  - is_a+b<e*d-e?
  - 76Trombones
- Why are other languages less liberal with legal identifiers?
Formalizing Definitions and Environments

Can’t Redefine a Variable in Racket

Other Racket Operators

Racket Documentation

Racket Guide:
https://docs.racket-lang.org/guide/

Racket Reference:
https://docs.racket-lang.org/reference