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

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
    - Emacs: M-x doctor
  - Needed a language for:
    - Symbolic computation
    - Programming with logic
    - Artificial intelligence
    - Experimental programming
  - So make one!

i.e., not just number crunching
**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](http://library.readscheme.org/page1.html))
  - Lambda the Ultimate PL blog inspired by these: [http://lambda-the-ultimate.org](http://lambda-the-ultimate.org)
- Led to Structure and Interpretation of Computer Programs (SICP) and MIT 6.001 ([https://mitpress.mit.edu/sicp/](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](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 Declarations**

- 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

Adds two numbers together.

Syntax: \((+ \ E1 \ E2)\)
Every parenthesis required; none may be omitted.
\(E1\) and \(E2\) stand in for any expression.
Note prefix notation.

Examples:
\((+ \ 251 \ 240)\)
\((+ \ (+ \ 251 \ 240) \ 301)\)
\((+ \ \texttt{#t} \ 251)\)

Addition: dynamic type checking

Syntax: \((+ \ E1 \ E2)\)

Evaluation rule:
1. Evaluate \(E1\) to a value \(V1\)
2. Evaluate \(E2\) 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.

Dynamic type-checking

Addition expression: evaluation

Syntax: \((+ \ E1 \ E2)\)

Evaluation rule:
1. Evaluate \(E1\) to a value \(V1\)
2. Evaluate \(E2\) to a value \(V2\)
3. Return the arithmetic sum of \(V1 + V2\).

Note recursive structure!

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:

\[
\text{if } E1 \downarrow V1 \text{ and } E2 \downarrow V2
\text{ and } V1 \text{ and } V2 \text{ are both numbers}
\text{ and } V \text{ is the sum of } V1 \text{ and } V2
\text{ then } (+ \ E1 \ E2) \downarrow V
\]
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

Errors Are Modeled by “Stuck” Derivations

How to evaluate \((+ \#t (+ 5 4))\)?

\[
\begin{align*}
\#t & \downarrow \#t \quad \text{[value]} \\
5 & \downarrow 5 \quad \text{[value]} \\
4 & \downarrow 4 \quad \text{[value]} \\
(+ 5 4) & \downarrow 9
\end{align*}
\]

Stuck here. Can’t apply (addition) rule because \#t is not a number in (+ #t 9)

How to evaluate \((+ 3 (+ 5 \#f))\)?

\[
\begin{align*}
1 & \downarrow 1 \quad \text{[value]} \\
2 & \downarrow 2 \quad \text{[value]} \\
(+ 1 2) & \downarrow 3 \quad \text{[addition]} \\
5 & \downarrow 5 \quad \text{[value]} \\
\#f & \downarrow \#f \quad \text{[value]}
\end{align*}
\]

Stuck here. Can’t apply (addition) rule because \#f is not a number in (+ 5 #f)

More Compact Derivation Notation

\[\begin{align*}
V & \downarrow V \quad \text{[value rule]} \\
E1 & \downarrow V1 \\
E2 & \downarrow V2 \\
(+ E1 E2) & \downarrow V
\end{align*}\]

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

Side conditions of rules:

Where \(V1\) and \(V2\) are numbers and \(V\) is the sum of \(V1\) and \(V2\).

Syntactic Sugar 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 \(E\) (or say if \(E \downarrow V\) then \((+ E) \downarrow V)\)
• Treat \((+ E)\) as 0 (or say \((+ E) \downarrow 0)\)
• This approach is known as syntactic sugar: introduce new syntactic forms that “desugar” into existing ones.
• In this case, an alternative approach would be to introduce more complex evaluation rules when + has a number of arguments different from 2.
Other Arithmetic Operators

Similar syntax and evaluation for
- * / quotient remainder min max
except:
- Second argument of /, quotient, remainder must be nonzero
- Result of / is a rational number (fraction) when both values are integers. (It is a floating point number if at least one value is a float.)
- quotient and remainder take exactly two arguments; anything else is an error.
- \((- E)\) is treated as \((- 0 E)\)
- \((/ E)\) is treated as \((/ 1 E)\)
- \((\text{min } E)\) and \((\text{max } E)\) treated as \(E\)
- (*) evaluates to 1.
- \((/), (-), (\text{min}), (\text{max})\) are errors (i.e., stuck)

Relation Operators

The following relational operators on numbers return booleans: \(< \leq = \geq \rangle\)

For example:

\[
\begin{array}{c}
E1 \downarrow V1 \\
E2 \downarrow V2 \\
(\langle E1 \ E2 \rangle) \downarrow V
\end{array}
\]

Where \(V1\) and \(V2\) are numbers and
\(V\) is \#t if \(V1\) is less than \(V2\)
or \#f if \(V1\) is not less than \(V2\)

Conditional (if) expressions

Syntax: \((\text{if } E\text{test } E\text{then } E\text{else})\)

Evaluation rule:
1. Evaluate \(E\text{test}\) to a value \(V\text{test}\).
2. If \(V\text{test}\) is not the value \#f then return the result of evaluating \(E\text{then}\)
   otherwise return the result of evaluating \(E\text{else}\)

Derivation-style rules for Conditionals

\[
\begin{array}{c}
E\text{test} \downarrow V\text{test} \\
E\text{then} \downarrow V\text{then} \quad \text{[if nonfalse]} \\
(\text{if } E\text{test } E\text{then } E\text{else}) \downarrow V\text{then}
\end{array}
\]

Where \(V\text{test}\) is not \#f

\[
\begin{array}{c}
E\text{test} \downarrow \#f \\
E\text{else} \downarrow V\text{else} \quad \text{[if false]} \\
(\text{if } E\text{test } E\text{then } E\text{else}) \downarrow V\text{else}
\end{array}
\]

Eelse is not evaluated!

Ethen is not evaluated!
Your turn

Use evaluation derivations to evaluate the following expressions

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

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

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

\[
\text{(+ (if 1 2 3) #t)}
\]

Expressions vs. statements

Conditional expressions can go anywhere an expression is expected:

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

\[
\text{(if (if (< 1 2) (> 4 3) (> 5 6)) (+ 7 8) (* 9 10))}
\]

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

Conditional expressions: careful!

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

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

\[
\text{(if #f (+ #t 240) 251)}
\]

Design choice in conditional semantics

In the \([\text{if nonfalse}]\) rule, \textit{Vtest} is not required to be a boolean!

\[
\begin{array}{c}
\text{Etest} \downarrow \text{Vtest} \\
\text{Ethen} \downarrow \text{Vthen} \quad \text{[if nonfalse]} \\
\text{(if Etest Ethen Eelse)} \downarrow \text{Vthen}
\end{array}
\]

Where \textit{Vtest} is not #f

This is a design choice for the language designer. What would happen if we replace the above rule by

\[
\begin{array}{c}
\text{Etest} \downarrow #t \\
\text{Ethen} \downarrow \text{Vthen} \quad \text{[if true]} \\
\text{(if Etest Ethen Eelse)} \downarrow \text{Vthen}
\end{array}
\]

This design choice is related to notions of “truthiness” and “falsiness” that you will explore in PS2.
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 (- 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:
    $\text{num} \leftrightarrow 17, \text{absoluteZero} \leftrightarrow -273, \text{true} \leftrightarrow \#t$
  – Abstract Example (use $\text{Id}$ to range over identifiers = names):
    $\text{Id}_1 \leftrightarrow V_1, \text{Id}_2 \leftrightarrow V_2, ..., \text{Id}_n \leftrightarrow V_n$
  – Empty environment: $\emptyset$
• An environment serves as a context for evaluating expressions that contain identifiers.
• Second argument to evaluation, which takes both an expression and an environment.

Addition: evaluation with environment

Syntax: $(+ \ E_1 \ E_2)$

Evaluation rule:
1. evaluate $E_1$ in the current environment to a value $V_1$
2. Evaluate $E_2$ in the current environment 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.

Variable references

Syntax: $\text{Id}$

$\text{Id}$: any identifier

Evaluation rule:
Look up and return the value to which $\text{Id}$ is bound in the current environment.
• Look-up proceeds by searching from the most-recently added bindings to the least-recently added bindings (front to back in our representation)
• If $\text{Id}$ is not bound in the current environment, evaluating it is “stuck” at an unbound variable error.

Examples:
• Suppose $\text{env}$ is $\text{num} \leftrightarrow 17, \text{absZero} \leftrightarrow -273, \text{true} \leftrightarrow \#t, \text{num} \leftrightarrow 5$
• In $\text{env}$, $\text{num}$ evaluates to 17 (more recent than 5), $\text{absZero}$ evaluates to $-273$, and $\text{true}$ evaluates to $\#t$. Any other name is stuck.
**define Declarations**

Syntax: \((\text{define } \text{Id } E)\)

- **define**: keyword
- **Id**: any identifier
- **E**: any expression

This is a declaration, not an expression! We will say a declarations are processed, not evaluated.

Processing 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 \(\text{Id} \rightarrow V\) at the front. Use this new environment as the current environment going forward.

---

**Environments: Example**

\(\text{env0} = \emptyset\) (can write as . in text)

\((\text{define x (+ 1 2)})\)

\(\text{env1} = x \rightarrow 3, \emptyset\) (abbreviated \(x \rightarrow 3\) in text)

\((\text{define y (* 4 x)})\)

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

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

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

\((\text{define test (< x diff)})\)

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

\((\text{define test (< x diff)})\)

\(\text{env5} = x \rightarrow 36, \text{test} \rightarrow \#t, \text{diff} \rightarrow 9, y \rightarrow 12, x \rightarrow 3\)

Note that binding \(x \rightarrow 36\) "shadows" \(x \rightarrow 3\), making it inaccessible.

---

**Evaluation Assertions & Rules with Environments**

The **evaluation assertion** notation \(E \# env \downarrow V\) means “Evaluating expression \(E\) in environment \(env\) yields value \(V\)”.

\(\text{Id} \# env \downarrow V\) [varref]

Where \(\text{Id}\) is an identifier and \(\text{Id} \rightarrow V\) is the first binding in \(\text{env}\) for \(\text{Id}\). Only this rule actually uses \(\text{env}\); others just pass it along.

\(V \# env \downarrow V\) [value]

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

\(E1 \# env \downarrow V1\)

\(E2 \# env \downarrow V2\) [addition]

\((+ E1 E2) \# env \downarrow V\)

Where \(V1\) and \(V2\) are numbers and \(V\) is the sum of \(V1\) and \(V2\). Rules for other arithmetic and relational ops are similar.

\(E1 \# env \downarrow \#f\)

\(E3 \# env \downarrow V3\) [if false]

\((\text{if } E1 E2 E3) \# env \downarrow V3\)

Where \(V1\) is not \(\#f\).

---

**Example Derivation with Environments**

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

\(\text{test} \# \text{env4} \downarrow \#t\) [varref]

\(x \# \text{env4} \downarrow 3\) [varref]

\(5 \# \text{env4} \downarrow 5\) [value]

\((\ast x 5) \# \text{env4} \downarrow 15\) [multiplication]

\(\text{diff} \# \text{env4} \downarrow 9\) [varref]

\((\ast (\ast x 5) \text{diff}) \# \text{env4} \downarrow 24\) [addition]

\((\text{if test } (+ (\ast x 5) \text{diff}) 17) \# \text{env4} \downarrow 24\) [if nonfalse]
Conclusion-below-subderivations, in text

Suppose env4 = test -> #t, diff -> 9, y -> 12, x -> 3

| test # env4 ; #t [varref] |
| x # env4 ; 3 [varref] |
| 5 # env4 ; 5 [value] |
| -------------------------- [multiplication] |
| (* x 5) # env4 ; 15 |
| diff # env4 ; 9 [varref] |
| -------------------------- [addition] |
| (+ (* x 5) diff) # env4 ; 24 |
| if nonfalse # env4 ; 24 |

Conclusion-above-subderivations, with bullets

Suppose env4 = test -> #t, diff -> 9, y -> 12, x -> 3

(if test (+ (* x 5) diff) 17) # env4 ; 24 [if nonfalse]
| test # env4 ; #t [varref] |
| (+ (* x 5) diff) # env4 ; 24 [addition] |
| x # env4 ; 3 [varref] |
| 5 # env4 ; 5 [value] |

if nonfalse # env4 ; 24

Formalizing definitions

The declaration assertion notation (define Id E) # env ↓ env’ means “Processing the definition (define Id E) in environment env yields a new environment env’”. We use a different arrow, ↓, to emphasize that definitions are not evaluated to values, but processed to environments.

\[
E \# env \downarrow V \\
(\text{define } Id E) \# env \downarrow Id \mapsto V, env
\]

Threading environments through definitions

\[
\begin{align*}
2 \# 2 & \downarrow 2 [\text{value}] \\
3 \# 3 & \downarrow 3 [\text{value}] \\
(+ 2 3) & \downarrow 5 [\text{addition}] \\
\text{(define a (+ 2 3))} & \downarrow a \mapsto 5 [\text{define}] \\
\end{align*}
\]

\[
\begin{align*}
a \# a & \mapsto 5 \downarrow 5 [\text{varref}] \\
(+ a a) & \# a \mapsto 5 \downarrow 25 [\text{multiplication}] \\
\text{(define b (* a a))} & \# a \mapsto 5 \downarrow b \mapsto 25, a \mapsto 5 [\text{define}] \\
\end{align*}
\]

\[
\begin{align*}
a \# b & \mapsto 25, a \mapsto 5 \downarrow 25 [\text{varref}] \\
\end{align*}
\]

\[
\begin{align*}
(- b a) & \# b \mapsto 25, a \mapsto 5 \downarrow 20 [\text{subtraction}] \\
\end{align*}
\]
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 ((), [], {}”, ’; #|\n  - Can't have same syntax as a number

- This means variable names can use (and even begin with) digits and characters like !@$%^&*.-+:<=>?
  - myLongName, my_long__name, my-long-name
  - is_a+b, c*d-e?
  - 76Trombones

- Why are other languages less liberal with legal identifiers?
Small-step semantics: conditional example

\[
(+ \ (\text{if } (< \ 1 \ 2) \ (* \ 3 \ 4) \ (/ \ 5 \ 6)) \ 7)
\Rightarrow (+ \ (\text{if } \#t \ (* \ 3 \ 4) \ (/ \ 5 \ 6)) \ 7)
\Rightarrow (+ \ (* \ 3 \ 4) \ 7)
\Rightarrow (+ \ 12 \ 7)
\Rightarrow 19
\]

Small-step semantics: errors as stuck expressions

Similar to big-step semantics, we model errors (dynamic type errors, divide by zero, etc.) in small-step semantics as expressions in which the evaluation process is stuck because no reduction rule is matched. For example

\[
(- \ (* \ (+ \ 2 \ 3) \ #t) \ (/ \ 18 \ 6))
\Rightarrow (- \ (* \ 5 \ #t) \ (/ \ 18 \ 6))
\Rightarrow (if \ (= \ 2 \ (/ \ (+ \ 3 \ 4) \ (- \ 5 \ 5))) \ 8 \ 9)
\Rightarrow (if \ (= \ 2 \ (/ \ 7 \ (- \ 5 \ 5))) \ 8 \ 9)
\Rightarrow (if \ (= \ 2 \ (/ \ 7 \ 0)) \ 8 \ 9)
\]

Small-step semantics: your turn

Use small-step semantics 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)
(+ \ (\text{if } 1 \ 2 \ 3) \ #t)

Racket Documentation

Racket Guide:
[https://docs.racket-lang.org/guide/](https://docs.racket-lang.org/guide/)

Racket Reference:
[https://docs.racket-lang.org/reference](https://docs.racket-lang.org/reference)