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

CS251 Programming Languages  
Spring 2018, Lyn Turbak  
Department of Computer Science  
Wellesley College

These slides build on Ben Wood’s Fall ‘15 slides

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!

Emacs: M-x doctor

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)
  - 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 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:
\[(+ 251 240)\]
\[(+ (+ 251 240) 301)\]
\[(+ \text{#t} 251)\]

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 \(V1 + V2\).

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.

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

\[(+ \#t (+ 5 4))\]?

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

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

How to evaluate

\[(+ (+ 1 2) (+ 5 \#f))\]?

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

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

More Compact Derivation Notation

\[V \downarrow V \ [\text{value rule}]
\]

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

\[
\begin{align*}
E_1 & \downarrow V_1 \\
E_2 & \downarrow V_2 \\
(+ E_1 E_2) & \downarrow V \\
\end{align*}
\]

(addition rule)

Errors Are Modeled by “Stuck” Derivations

How to evaluate

\[(+ \#t (+ 5 4))\]?

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

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

Syntactic Sugar for Addition

The addition operator + can take any number of operands.

- For now, treat \((+ E_1 E_2 \ldots E_n)\) as \((+ (+(E_1 E_2) \ldots E_n))\)
- 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 \((+)\) as 0 (or say \((+) \downarrow 0\))

- 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)
* (min E) and (max E) treated as E
* (*) evaluates to 1.
* (/), (-), (min), (max) are errors (i.e., stuck)

Relation Operators

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

For example:

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

[less than]

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: (if Etest Ethen Eelse)

Evaluation rule:
1. Evaluate Etest to a value Vtest.
2. If Vtest is not the value #f then return the result of evaluating Ethen otherwise return the result of evaluating Eelse

Derivation-style rules for Conditionals

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

Where Vtest is not #f

\[
\begin{array}{c}
\text{Etest } \downarrow #f \\
\text{Eelse } \downarrow \text{Velse} \text{ [if false]} \\
\text{(if Etest Ethen Eelse)} \downarrow \text{Velse}
\end{array}
\]

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:

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

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

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, ...)
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:
    ```scheme
    num ⟷ 17, absoluteZero ⟷ -273, true ⟷ #t
    ```
  - Abstract Example (use Id to range over identifiers = names):
    ```scheme
    Id1 ⟷ V1, Id2 ⟷ V2, ..., Idn ⟷ Vn
    ```
  - Empty environment: ∅
- 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: `( + E1 E2 )`

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.

Variable references

Syntax: **Id**

- **Id**: any identifier

Evaluation rule:
- Look up and return the value to which **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 **Id** is not bound in the current environment, evaluating it is “stuck” at an unbound variable error.

Examples:
- Suppose `env` is `num ⟷ 17, absZero ⟷ -273, true ⟷ #t, num ⟷ 5`
- In `env`, `num` evaluates to `17` (more recent than `5`), `absZero` evaluates to `-273`, and `true` evaluates to `#t`. Any other name is stuck.
**define Declarations**

Syntax: `(define 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 `Id → V` at the front. Use this new environment as the current environment going forward.

---

**Environments: Example**

```scheme
(env0 = ∅)  
(define x (+ 1 2))  
(define y (* 4 x))  
(define diff (- y x))  
(define test (< x diff))  
(define x (* x y))
```

- `env0 = ∅` (can write as . in text)
- `(define x (+ 1 2))`
- `env1 = x → 3, ∅` (abbreviated x -> 3 in text)
- `(define y (* 4 x))`
- `env2 = y → 12, x → 3` (most recent binding first)
- `(define diff (- y x))`
- `env3 = diff → 9, y → 12, x → 3`
- `(define test (< x diff))`
- `env4 = test → #t, diff → 9, y → 12, x → 3`
- `(define x (* x y))`

Note that binding `x → 36" shadows" x → 3, making it inaccessible.

---

**Example Derivation with Environments**

Suppose `env4 = test → #t, diff → 9, y → 12, x → 3`

```scheme
(env4 ↓ test → #t, diff → 9, y → 12, x → 3)
```

---

**Evaluation Assertions & Rules with Environments**

The **evaluation assertion** notation `E # env ↓ V` means "Evaluating expression `E` in environment `env` yields value `V". 

- `Id # env ↓ V` [varref]
  - Where `Id` is an identifier and `Id → V` is the first binding in `env` for `Id` Only this rule actually uses `env`; others just pass it along.

- `V # env ↓ V` [value]
  - where `V` is a value (number, boolean, etc.)

- `E1 # env ↓ #f` [if false]
  - `E3 # env ↓ V3` [if false]
  - `(if E1 E2 E3) # env ↓ V3` Where `V1` is not `#f`

- `E1 # env ↓ #t` [varref]
  - `E2 # env ↓ V2` [addition]
  - `(E1 E2) # env ↓ 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.
Conclusion-below-subderivaNons, in text

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

<p>| test # env4 ; #t [varref] | (* x 5) # env4 ; 15 [value] |
| _____________________________ | ____________________________ |</p>
<table>
<thead>
<tr>
<th>diff # env4 ; 9 [varref]</th>
<th>(+ (* x 5) diff) # env4 ; 24 [if nonfalse]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ (+ (* x 5) diff) 17) # env4 ; 24</td>
<td></td>
</tr>
</tbody>
</table>

Formalizing definitions

The declaration assertion notation \(\text{define } Id E \# env \downarrow env'\) means “Processing the definition \(\text{define } Id E\) 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.

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

Conclusion-above-subderivaNons, with bullets

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

1. (if test (+ (* x 5) diff) 17) # env4 ↓ 24 [if nonfalse]
2. test # env4 ↓ #t [varref]
3. (+ (* x 5) diff) # env4 ↓ 24 [addition]
   - (* x 5) # env4 ↓ 15 [multiplication]
   - 5 # env4 ↓ 5 [value]
4. diff # env4 ↓ 9 [multiplication]

Formalizing definitions

Threading environments through definitions

\[
\begin{array}{c}
1 \# \varnothing \downarrow 2 \text{ [value]} \\
3 \# \varnothing \downarrow 3 \text{ [value]} \\
(+ 2 3)\# \varnothing \downarrow 5 \text{ [define]}
\end{array}
\]

\[
\begin{array}{c}
a \# a \mapsto 5 \downarrow 5 \text{ [varref]} \\
\ast a a \# a \mapsto 5 \downarrow 25 \text{ [multiplication]} \\
\end{array}
\]

\[
\begin{array}{c}
b \# b \mapsto 25, a \mapsto 5 \downarrow 25 \text{ [varref]} \\
\ast b a \# b \mapsto 25, a \mapsto 5 \downarrow 20 \text{ [subtraction]} \\
\end{array}
\]
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 !@$%^&*.-+:<=>/ E.g.:
  - 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 vs. big-step semantics

The evaluation derivations we’ve seen so far are called a big-step semantics because the derivation \( e \# \text{env} \downarrow v \) explains the evaluation of \( e \) to \( v \) as one “big step” justified by the evaluation of its subexpressions.

An alternative way to express evaluation is a small-step semantics in which an expression is simplified to a value in a sequence of steps that simplifies subexpressions. You do this all the time when simplifying math expressions, and we can do it in Racket, too. E.g;

\[
\begin{align*}
(- (* (+ 2 3) 9) (/ 18 6)) &\Rightarrow (- (* \underline{5}\underline{9}) (/ 18 6)) \text{[addition]} \\
&\Rightarrow (- \underline{45} (/ 18 6)) \text{[multiplication]} \\
&\Rightarrow (- \underline{45} 3) \text{[division]} \\
&\Rightarrow 42 \text{[subtraction]}
\end{align*}
\]

Small-step semantics: intuition

Scan left to right to find the first redex (nonvalue subexpression that can be reduced to a value) and reduce it:

\[
\begin{align*}
(- (* (+ 2 3) 9) (/ 18 6)) &\Rightarrow (- (* \underline{5}\underline{9}) (/ 18 6)) \\
&\Rightarrow (- \underline{45} (/ 18 6)) \\
&\Rightarrow (- \underline{45} 3) \\
&\Rightarrow 42
\end{align*}
\]

Small-step semantics: reduction rules

There are a small number of reduction rules for Racket. These specify the redexes of the language and how to reduce them.

The rules often require certain subparts of a redex to be (particular kinds of) values in order to be applicable.

\[
\begin{align*}
\text{Id} &\Rightarrow V, \text{ where Id} \Rightarrow V \text{ is the first binding for Id in the current environment}\* \text{[varref]} \\
(+ V1 V2) &\Rightarrow V, \text{ where } V \text{ is the sum of numbers } V1 \text{ and } V2 \text{[addition]} \\
\text{(if Vtest Ethen Eelse)} &\Rightarrow \text{Ethen}, \text{ if Vtest is not } \#f \text{[if nonfalse]} \\
\text{(if } \#f \text{ Ethen Eelse)} &\Rightarrow \text{Eelse} \text{[if false]}
\end{align*}
\]

* In a more formal approach, the notation would make the environment explicit. E.g., \( E \# \text{env} \Rightarrow V \)
Small-step semantics: conditional example

\[
(+ \ (if \ \{(\textless \ 1 \ 2)\} \ (* \ 3 \ 4) \ (/ \ 5 \ 6)) \ 7)
\Rightarrow \ (+ \ \{(\text{if \ #t} \ (* \ 3 \ 4) \ (/ \ 5 \ 6))\} \ 7) \ [\text{less than}]
\Rightarrow \ (+ \ \{(\text{* \ 3 \ 4}) \ 7\} \ [\text{if \ nonfalse}]
\Rightarrow \ \{(+ \ 12 \ 7)\} \ [\text{multiplication}]
\Rightarrow \ 19 \ [\text{addition}]
\]

Notes for writing derivations in text:
- You can use => for ⇒
- Use curly braces {...} to mark the redex
- Use square brackets to name the rule used to reduce the redex from the previous line to the current line.

Small-step semantics: your turn

Use small-step semantics to evaluate the following expressions:

\[
(\text{if} \ (< \ 8 \ 2) \ (+ \ #f \ 5) \ (+ \ 3 \ 4))
\]
\[
(\text{if} \ (+ \ 1 \ 2) \ (- \ 3 \ 7) \ (/ \ 9 \ 0))
\]
\[
(+ \ (\text{if} \ (< \ 1 \ 2) \ (* \ 3 \ 4) \ (/ \ 5 \ 6)) \ 7)
\]
\[
(+ \ (\text{if} \ 1 \ 2 \ 3) \ #t)
\]

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:

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

Racket Documentation

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

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