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!

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/)
• 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

Adds 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: \((+ \; e1 \; e2)\)

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!

Not quite!
Addition: dynamic type checking

Syntax: \((+ \text{ } e1 \text{ } e2)\)

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.

Still not quite! More later ...
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**:

  \[
  \text{if } e_1 \downarrow v_1 \text{ and } e_2 \downarrow v_2
  \hspace{1cm}
  \text{and } v_1 \text{ and } v_2 \text{ are both numbers}
  \hspace{1cm}
  \text{and } v \text{ is the sum of } v_1 \text{ and } v_2
  \hspace{1cm}
  \text{then } ( + \ e_1 \ e_2 ) \downarrow v
  \]
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

\[ v \downarrow v \quad \text{(value rule)} \]

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

side conditions of rules

\[ e1 \downarrow v1 \]
\[ e2 \downarrow v2 \quad \text{(addition rule)} \]
\[ (+ e1 e2) \downarrow v \]

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

\[
\begin{align*}
3 & \downarrow 3 \quad \text{(value)} \\
5 & \downarrow 5 \quad \text{(value)} \\
4 & \downarrow 4 \quad \text{(value)} \\
(+ 5 4) & \downarrow 9 \quad \text{(addition)} \\
(+ 3 (+ 5 4)) & \downarrow 12 \quad \text{(addition)}
\end{align*}
\]
Errors 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

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

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

Stuck here. Can’t apply (addition) rule because #f is not a number
Special Cases 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 \(e\)

- Treat \((+)\) as 0 (or say \((+) \downarrow 0\) )
Other Arithmetic Operators

Similar syntax and evaluation for

\[- \ast / \textit{quotient remainder}\]

except:

• Second argument of `/`, \textit{quotient, remainder} must be nonzero

• Result of `/` is a rational number (fraction)

• \textit{quotient} and \textit{remainder} take exactly two arguments; anything else is an error.

• \(\text{- e}\) is treated as \(\text{- 0 e}\)

• \(\text{/ e}\) is treated as \(\text{/ 1 e}\)

• \((\ast)\) evaluates to 1.

• \((/ )\) and \((\text{-})\) are errors.
Relation Operators

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

For example:

\[
\begin{align*}
& e_1 \downarrow \mathbf{v}_1 \\
& e_2 \downarrow \mathbf{v}_2 \\
& (< e_1 e_2) \downarrow \mathbf{v}
\end{align*}
\]

(less than rule)

Where \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \) are numbers and
\( \mathbf{v} \) is \#t if \( \mathbf{v}_1 \) is less than \( \mathbf{v}_2 \)

or \#f if \( \mathbf{v}_1 \) is not less than \( \mathbf{v}_2 \)
Conditional (if) expressions

Syntax: \((\text{if } e_1 \ e_2 \ e_3)\)

Evaluation rule:
1. Evaluate \(e_1\) to a value \(v_1\).
2. If \(v\) is not the value \(#f\) then
   return the result of evaluating \(e_2\)
   otherwise
   return the result of evaluating \(e_3\)
Conditional (if) expressions

\[
\begin{align*}
  e_1 &\downarrow v_1 \\
nonfalse
  e_2 &\downarrow v_2 \\
\end{align*}
\] (if nonfalse)

\[
(\text{if} \ e_1 \ e_2 \ e_3) \downarrow v_2
\]

where \(v_1\) is not \#f

\[
\begin{align*}
  e_1 &\downarrow \#f \\
nonfalse
  e_3 &\downarrow v_3 \\
\end{align*}
\] (if false)

\[
(\text{if} \ e_1 \ e_2 \ e_3) \downarrow v_3
\]

\(e_3\) not evaluated!

\(e_2\) not evaluated!
Your turn

Use evaluation derivations to evaluate the following expressions

\[
\begin{align*}
&\text{(if (< 8 2) (+ #f 5) (+ 3 4))} \\
&\text{(if (+ 1 2) (- 3 7) (/ 9 0))} \\
&\text{(+ (if (< 1 2) (* 3 4) (/ 5 6)) 7)}
\end{align*}
\]
Expressions vs. statements

If expressions can go anywhere an expression is expected:

\[
\text{(if (< 9 (\(-\ 251\ 240\)))}
\begin{align*}
\text{(* 3 (+ 4 5))} \\
\text{(+ 6 (* 7 8))}
\end{align*}
\]\n
\[
(+ 4 (* (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, ...)

3-22
If expressions: careful!

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

(if (> 251 240) 251 (/ 251 0))

(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 (- 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: ∅

- 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: \textit{id}

\textit{id}: any identifier

Evaluation rule:

Look up and return the value to which \textit{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)

Examples:

- Suppose \textit{env} is num → 17, absoluteZero → −273, true → #t
- In \textit{env}, num evaluates to 17, absoluteZero evaluates to −273, and true evaluates to #t
define bindings

Syntax: \( (\text{define } id \ e) \)

- **define**: keyword
- **id**: any identifier
- **e**: any expression

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 \( id \rightarrow v \) at the front.
Environments: Example

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

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

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

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

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

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

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

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

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

\( \text{(if test (+ (* x 5) diff) 17)} \)

Environment here is still \emph{env4}
Evaluation Assertions & Rules with Environments

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

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

- \( v \# \text{env} \downarrow v \) (value)
  - where \( v \) is a value
    - (number, boolean, etc.)

- \( e1 \# \text{env} \downarrow \#\text{\^f} \)
  - \( e3 \# \text{env} \downarrow v3 \) (if false)
  - (if \( e1 \ e2 \ e3 \) \# \text{env} \downarrow v3)

- \( e1 \# \text{env} \downarrow v1 \)
  - \( e2 \# \text{env} \downarrow v2 \)
  - \( (\ + \ e1 \ e2) \# \text{env} \downarrow v \) (addition)

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

- \( e1 \# \text{env} \downarrow v1 \)
  - \( e2 \# \text{env} \downarrow v2 \)
  - \( (\text{if} \ e1 \ e2 \ e3) \# \text{env} \downarrow v2 \) (if nonfalse)

  Where \( v1 \) is not \( \#\text{\^f} \)
Example Derivation with Environments

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

\[
\begin{align*}
\text{test} & \quad \# env4 \downarrow \#t \quad \text{(varref)} \\
\text{x} & \quad \# env4 \downarrow 3 \quad \text{(varref)} \\
5 & \quad \# env4 \downarrow 5 \quad \text{(value)} \\
(* \quad x \quad 5) & \quad \# env4 \downarrow 15 \quad \text{(multiplication)} \\
\text{diff} & \quad \# env4 \downarrow 9 \quad \text{(varref)} \\
(+ \quad (* \quad x \quad 5) \quad \text{diff}) & \quad \# env4 \downarrow 24 \quad \text{(addition)} \\
\text{if test} \quad (+ \quad (* \quad x \quad 5) \quad \text{diff}) \quad 17) & \quad \# env4 \downarrow 24 \quad \text{(if nonfalse)}
\end{align*}
\]
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 ! @ $ % ^ & * . - + : <= > ? / 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?
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