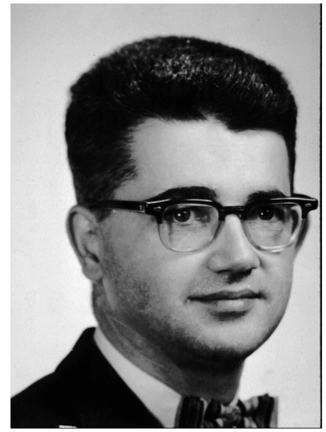
Introduction to Racket, a dialect of LISP: Expressions and Declarations

CS251 Programming Languages
Spring 2019, Lyn Turbak

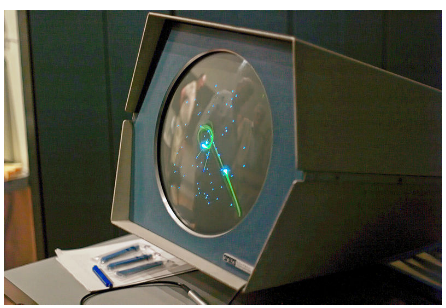
Department of Computer Science
Wellesley College

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

Emacs: M-x doctor

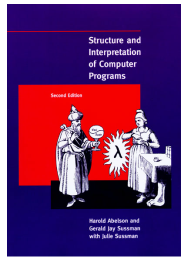
- Needed a language for:
  - Symbolic computation
  - Programming with logic
  - Artificial intelligence
  - Experimental programming

i.e., not just number crunching

- 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 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.

Note recursive structure!

Examples:

```
(+ 251 240)
(+ (+ 251 240) 301)
(+ #t 251)
```


Addition expression: evaluation

Syntax: `(+ E1 E2)`

Note recursive structure!

Evaluation rule:

1. Evaluate ***E1*** to a value ***V1***
2. Evaluate ***E2*** to a value ***V2***
3. Return the arithmetic sum of ***V1*** + ***V2***.

Not quite!

Addition: dynamic type checking

Syntax: `(+ E1 E2)`

Still not quite!
More later …

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

Evaluation Assertions Formalize Evaluation

The **evaluation assertion** notation ***E*** ↓ ***V*** means ``***E*** evaluates to ***V***''.

Our evaluation rules so far:

- *value rule*: ***V*** ↓ ***V*** (where ***V*** is a number or boolean)
- *addition rule*:

  if ***E1*** ↓ ***V1*** and ***E2*** ↓ ***V2***
  and ***V1*** and ***V2*** are both numbers
  and ***V*** is the sum of ***V1*** and ***V2***
  then (+ ***E1*** ***E2***) ↓ ***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))` ↓ `12` by the addition rule because:

- `3` ↓ `3` by the value rule
- `(+ 5 4)` ↓ `9` by the addition rule because:
  - `5` ↓ `5` by the value rule
  - `4` ↓ `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 ↓ ***V*** [value rule]

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

side conditions of rules

E1 ↓ ***V1***
E2 ↓ ***V2***
——————— [addition rule]
`(+` ***E1*** ***E2***`)` ↓ ***V***

Where ***V1*** and ***V2*** are numbers and ***V*** is the sum of ***V1*** and ***V2***.

```
|  3 ↓ 3 [value]
|  | 5 ↓ 5 [value]
|  | 4 ↓ 4 [value]
|  ------------------ [addition]
|  (+ 5 4) ↓ 9
--------------------- [addition]
(+ 3 (+ 5 4)) ↓ 12
```


Errors Are Modeled by "Stuck" Derivations

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

```
| #t ↓ #t [value]
|  | 5 ↓ 5 [value]
|  | 4 ↓ 4 [value]
|  ------------- [addition]
| (+ 5 4) ↓ 9
----------------
```

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

How to evaluate `(+ (+ 1 2) (+ 5 #f))`?

```
| 1 ↓ 1 [value]
| 2 ↓ 2 [value]
---------------
(+ 1 2) ↓ 3 [addition]
| 5 ↓ 5 [value]
| #f ↓ #f [value]
---------------
```

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

Syntactic Sugar for Addition

The addition operator + can take any number of operands.

- For now, treat `(+` ***E1*** ***E2*** … ***En***`)` as `(+ (+` ***E1*** ***E2***`)` … ***En***`)`
  E.g., treat `(+ 7 2 -5 8)` as `(+ (+ (+ 7 2) -5) 8)`
- Treat `(+` ***E***`)` as ***E*** (or say if ***E*** ↓ ***V*** then `(+` ***E***`)` ↓ ***V***)
- Treat `(+)` as `0` (or say `(+)` ↓ `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:

$$\frac{E1 \downarrow V1 \qquad E2 \downarrow V2}{(\texttt{<}\ E1\ E2) \downarrow V}\ \text{[less than]}$$

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

$$\frac{Etest \downarrow Vtest \qquad Ethen \downarrow Vthen}{(\texttt{if}\ Etest\ Ethen\ Eelse) \downarrow Vthen}\ \text{[if nonfalse]}$$

Where ***Vtest*** is not #f

Eelse is not evaluated!

$$\frac{Etest \downarrow \texttt{\#f} \qquad Eelse \downarrow Velse}{(\texttt{if}\ Etest\ Ethen\ Eelse) \downarrow Velse}\ \text{[if false]}$$

Ethen is not evaluated!

Your turn

Use evaluation derivations to evaluate the following expressions

```
(if (< 8 2) (+ #f 5) (+ 3 4))

(if (+ 1 2) (- 3 7) (/ 9 0))

(+ (if (< 1 2) (* 3 4) (/ 5 6)) 7)

(+ (if 1 2 3) #t)
```


Expressions vs. statements

Conditional expressions can go anywhere an expression is expected:

```
(+ 4 (* (if (< 9 (- 251 240)) 2 3) 5))

(if (if (< 1 2) (> 4 3) (> 5 6))
    (+ 7 8)
    (* 9 10))
```

Note: `if` is an *expression*, not a *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!

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

(if #f (+ #t 240) 251)
```


Design choice in conditional semantics

In the [if nonfalse] rule, ***`Vtest`*** is **not** required to be a boolean!

$$\frac{\textit{Etest} \downarrow \textit{Vtest} \quad \textit{Ethen} \downarrow \textit{Vthen}}{(\texttt{if}\ \textit{Etest}\ \textit{Ethen}\ \textit{Eelse}) \downarrow \textit{Vthen}}\ \text{[if nonfalse]}$$

Where ***Vtest*** is not #f

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

$$\frac{\textit{Etest} \downarrow \texttt{\#t} \quad \textit{Ethen} \downarrow \textit{Vthen}}{(\texttt{if}\ \textit{Etest}\ \textit{Ethen}\ \textit{Eelse}) \downarrow \textit{Vthen}}\ \text{[if true]}$$

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:
    `num` ⟼ `17`, `absoluteZero` ⟼ `-273`, `true` ⟼ `#t`
  - Abstract Example (use ***Id*** to range over identifiers = names):
    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

env0 = ∅ (can write as `.` in text)

`(define x (+ 1 2))`

env1 = x ⟼ 3, ∅ (abbreviated x ⟼ 3; can write as `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

`(if test (+ (* x 5) diff) 17)`

environment here is still ***env4***

`(define x (* x y))`

env5 = x ⟼ 36, test ⟼ #t, diff ⟼ 9, y ⟼ 12, x ⟼ 3
Note that binding x ⟼ 36 "shadows" x ⟼ 3 , making it inaccessible

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
E3 # ***env*** ↓ ***V3***
――――――――――――― [if false]
(if ***E1 E2 E3***) # ***env*** ↓ ***V3***

E1 # ***env*** ↓ ***V1***
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.

E1 # ***env*** ↓ ***V1***
E2 # ***env*** ↓ ***V2***
――――――――――――― [if nonfalse]
(if ***E1 E2 E3***) # ***env*** ↓ ***V2***

Where ***V1*** is not #f

Example Derivation with Environments

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]
(if test (+ (* x 5) diff) 17) # *env4* ↓ 24

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]
(if test (+ (* x 5) diff) 17)# 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 5) # env4 ↓ 15 [multiplication]
    - x # env4 ↓ 3 [varref]
    - 5 # env4 ↓ 5 [value]
  - diff # env4 ↓ 9 [multiplication]

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 ↓ V
--------------------------------- [define]
(define Id E)# env ⇓ Id ↦ V, env
```


Threading environments through definitions

```
| |2 # ∅ ↓ 2 [value]
| |3 # ∅ ↓ 3 [value]
| ----------------------- [addition]
| (+ 2 3)# ∅ ↓ 5
----------------------------------- [define]
(define a (+ 2 3))# ∅ ⇓ a ↦ 5
```

```
| |a # a ↦ 5 ↓ 5 [varref]
| |a # a ↦ 5 ↓ 5 [varref]
| ----------------------------- [multiplication]
| (* a a)# a ↦ 5 ↓ 25
---------------------------------------- [define]
(define b (* a a))# a ↦ 5 ⇓ b ↦ 25, a ↦ 5
```

```
| b # b ↦ 25, a ↦ 5 ↓ 25 [varref]
| a # b ↦ 25, a ↦ 5 ↓ 5 [varref]
---------------------------------- [subtraction]
(- b a)# b ↦ 25, a ↦ 5 ↓ 20
```

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 identifers. 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 # env2* ↓ *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;

```
(- (* (+ 2 3) 9) (/ 18 6))
⇒ (- (* 5 9) (/ 18 6))
⇒ (- 45 (/ 18 6))
⇒ (- 45 3)
⇒ 42
```


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:

```
(- (* [(+ 2 3)] 9) (/ 18 6))
⇒ (- [(* 5 9)] (/ 18 6)) [addition]
⇒ (- 45 [(/ 18 6)]) [multiplication]
⇒ [(- 45 3)] [division]
⇒ 42 [subtraction]
```


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.

Id ⇒ *V* , where *Id* ↦ *V* is the first binding for *Id* in the current environment* [varref]

`(+` *V1 V2* `)` ⇒ *V*, where *V* is the sum of numbers *V1* and *V2* [addition]

There are similar rules for other arithmetic/relational operators

`(if` *Vtest Ethen Eelse* `)` ⇒ *Ethen*, if *Vtest* is not *#f* [if nonfalse]

`(if #f` *Ethen Eelse* `)` ⇒ *Eelse* [if false]

* In a more formal approach, the notation would make the environment explicit. E.g., *E # env* ⇒ *V*

Small-step semantics: conditional example

```
(+ (if {(< 1 2)} (* 3 4) (/ 5 6)) 7)
=> (+ {(if #t (* 3 4) (/ 5 6))} 7) [less than]
⇒ (+ {(* 3 4)} 7) [if nonfalse]
⇒ {(+ 12 7)} [multiplication]
⇒ 19 [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: 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))
⇒ (- (* 5 #t) (/ 18 6)) Stuck!

(if (= 2 (/ (+ 3 4) (- 5 5))) 8 9)
⇒ (if (= 2 (/ 7 (- 5 5))) 8 9)
⇒ (if (= 2 (/ 7 0)) 8 9) Stuck!
```


Small-step semantics: your turn

Use small-step semantics to evaluate the following expressions:

```
(if (< 8 2) (+ #f 5) (+ 3 4))

(if (+ 1 2) (- 3 7) (/ 9 0))

(+ (if (< 1 2) (* 3 4) (/ 5 6)) 7)

(+ (if 1 2 3) #t)
```


Racket Documentation

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

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