

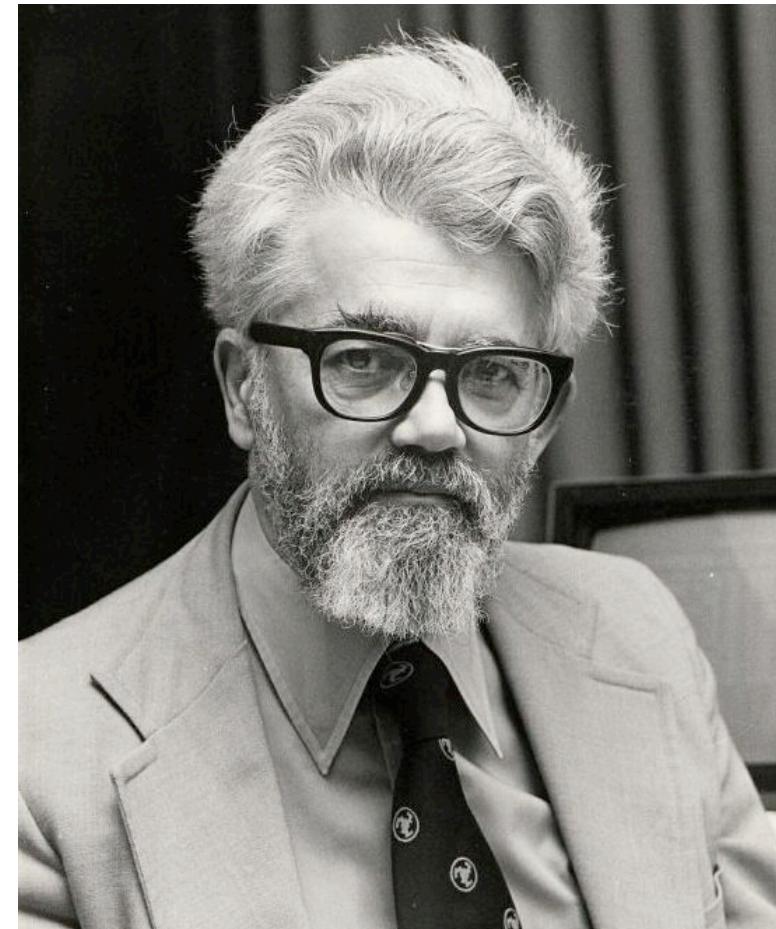
# Introduction to Racket, a dialect of LISP: Expressions and Declarations

---

Design of Programming Languages

---

LISP: designed by John McCarthy, 1958  
published 1960



# LISP: implemented by Steve Russell, early 1960s



# LISP: LISt Processing

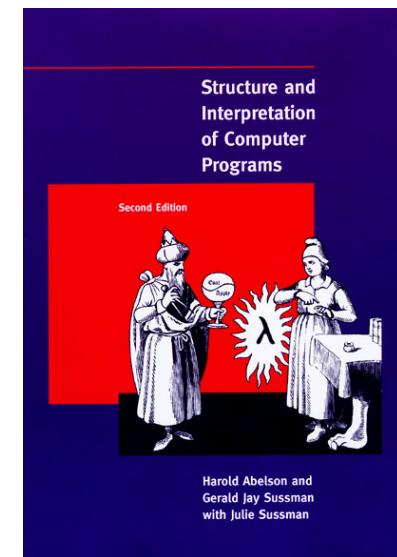
- McCarthy, MIT artificial intelligence, 1950s-60s
  - Advice Taker: represent logic as data, not just program
- Needed a language for:
  - Symbolic computation
  - Programming with logic
  - Artificial intelligence
  - Experimental programming
- So make one!

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?
  - 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:  $(+ E1 E2)$

Evaluation rule:

Note recursive  
structure!

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:  $(+ E1 E2)$

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 ...

Dynamic type-checking

# 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*:  
*if*  $E1 \downarrow V1$  and  $E2 \downarrow V2$   
and  $V1$  and  $V2$  are both numbers  
and  $V$  is the sum of  $V1$  and  $V2$   
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

# More Compact Derivation Notation

$V \downarrow V$  [value rule]

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

↑  
side conditions of rules →

$E1 \downarrow V1$

$E2 \downarrow V2$

$(+ E1 E2) \downarrow V$  [addition rule]

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

$$\begin{array}{c} 3 \downarrow 3 \text{ [value]} \\ | \\ 5 \downarrow 5 \text{ [value]} \\ | \\ 4 \downarrow 4 \text{ [value]} \\ \hline (+ 5 4) \downarrow 9 \text{ [addition]} \\ | \\ (+ 3 (+ 5 4)) \downarrow 12 \text{ [addition]} \end{array}$$

# Errors Are Modeled by “Stuck” Derivations

How to evaluate

(+ #t (+ 5 4)) ?

$$\begin{array}{l} \#t \downarrow \#t \text{ [value]} \\ | \\ 5 \downarrow 5 \text{ [value]} \\ | \\ 4 \downarrow 4 \text{ [value]} \\ \hline (+ 5 4) \downarrow 9 \end{array}$$

[addition]

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

How to evaluate

(+ (+ 1 2) (+ 5 #f)) ?

$$\begin{array}{l} 1 \downarrow 1 \text{ [value]} \\ | \\ 2 \downarrow 2 \text{ [value]} \\ | \\ (+ 1 2) \downarrow 3 \text{ [addition]} \\ | \\ 5 \downarrow 5 \text{ [value]} \\ | \\ \#f \downarrow \#f \text{ [value]} \end{array}$$

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  $(+ \ E1 \ E2 \ \dots \ En)$  as  $(+ \ (+ \ E1 \ E2) \ \dots \ En)$   
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  $(+)$  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:

$$\begin{array}{l} E1 \downarrow V1 \\ E2 \downarrow V2 \\ \hline (< E1 E2) \downarrow V \end{array} \quad [\text{less than}]$$

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

$Etest \downarrow Vtest$

$Ethen \downarrow Vthen$

[if nonfalse]

(if  $Etest\ Ethen\ Else$ )  $\downarrow Vthen$

Else is not evaluated!

Where  $Vtest$  is not #f

$Etest \downarrow \#f$

$Else \downarrow Velse$

[if false]

(if  $Etest\ Ethen\ Else$ )  $\downarrow Velse$

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)



# Solution #1

```

8 ↓ 8 [value]
2 ↓ 2 [value]
(< 8 2) ↓ #f [less-than]
3 ↓ 3 [value]
4 ↓ 4 [value]
(+ 3 4) ↓ 7 [addition]
(if#f (< 8 2) (+ #f 5) (+ 3 4)) ↓ 7 [if-false]

```

```

1 ↓ 1 [value]
2 ↓ 2 [value]
(+ 1 2) ↓ 3 [addition]
3 ↓ 3 [value]
7 ↓ 7 [value]
(- 3 7) ↓ -4 [subtraction]
(if#t (+ 1 2) (- 3 7) (/ 9 0)) ↓ -4 [if-nonfalse]

```



# Solution #2

```

1 ↓ 1 [value]
2 ↓ 2 [value]
(< 1 2) ↓ #t [less-than]

3 ↓ 3 [value]
4 ↓ 4 [value]
(* 3 4) ↓ 12 [multiplication]
(if (< 1 2) (* 3 4) (/ 5 6)) ↓ 12 [if-nonfalse]
7 ↓ 7 [value]
(+ (if (< 1 2) (* 3 4) (/ 5 6)) 7) ↓ 19 [addition]

```

```

1 ↓ 1 [value]
2 ↓ 2 [value]
(if 1 2 3) ↓ 2 [if-nonfalse]
#t ↓ #t [value]
(+ (if 1 2 3) #t)

```

Stuck here! Can't apply [addition] rule because #t is not a number in (+ 2 #t).  
 This models a dynamic type checking error.

# 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)
```

# 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, ...)

# Design choice in conditional semantics

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

$$\begin{array}{l} Etest \downarrow Vtest \\ Ethen \downarrow Vthen \end{array} \quad [\text{if nonfalse}]$$
$$(\text{if } Etest\ Ethen\ Else) \downarrow Vthen$$

Where  $Vtest$  is not #f

This is a design choice for the language designer.

What would happen if we replace the above rule by

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

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:  
 $\text{num} \mapsto 17, \text{absoluteZero} \mapsto -273, \text{true} \mapsto \#t$
  - Abstract Example (use ***Id*** to range over identifiers = names):  
 $\text{Id}_1 \mapsto V_1, \text{Id}_2 \mapsto V_2, \dots, \text{Id}_n \mapsto V_n$
  - 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:  $(+ E1 E2)$

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:  $\text{Id}$

$\text{Id}$ : 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  $\text{num} \mapsto 17$ ,  $\text{absZero} \mapsto -273$ ,  $\text{true} \mapsto \#t$ ,  $\text{num} \mapsto 5$
- 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: **(define  $Id$   $E$ )**

**define**: keyword

$Id$ : any *identifier*

$E$ : any expression

This is a **declaration**, not an **expression**!

We will say **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  $Id \rightarrow V$  at the front. Use this new environment as the current environment going forward.

# Environments: Example

**env0** =  $\emptyset$  (can write as . in text)

```
(define x (+ 1 2))
```

**env1** =  $x \mapsto 3, \emptyset$  (abbreviated  $x \mapsto 3$ ; can write as  $x \rightarrow 3$  in text)

```
(define y (* 4 x))
```

**env2** =  $y \mapsto 12, x \mapsto 3$  (most recent binding first)

```
(define diff (- y x))
```

**env3** =  $diff \mapsto 9, y \mapsto 12, x \mapsto 3$

```
(define test (< x diff))
```

**env4** =  $test \mapsto \#t, diff \mapsto 9, y \mapsto 12, x \mapsto 3$

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

environment here is still **env4**

```
(define x (* x y))
```

**env5** =  $x \mapsto 36, test \mapsto \#t, diff \mapsto 9, y \mapsto 12, x \mapsto 3$

Note that binding  $x \mapsto 36$  “shadows”  $x \mapsto 3$ , making it inaccessible

Expr/decl 30

# Evaluation Assertions & Rules with Environments

The **evaluation assertion** notation  $E \# env \downarrow V$  means

``Evaluating expression  $E$  in environment  $env$  yields value  $V$ ''.

$Id \# env \downarrow V$  [varref]

Where  $Id$  is an identifier and  
 $Id \mapsto V$  is the first binding in  
 $env$  for  $Id$  Only this rule actually  
uses env; others just  
pass it along

$V \# env \downarrow V$  [value]

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

$E1 \# env \downarrow \# f$   
 $E3 \# env \downarrow V3$  [if false]

(if  $E1 E2 E3$ )  $\# env \downarrow V3$

$E1 \# env \downarrow V1$

$E2 \# env \downarrow V2$

(+  $E1 E2$ )  $\# env \downarrow V$  [addition]

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

$E1 \# env \downarrow V1$

$E2 \# env \downarrow V2$

[if nonfalse]

(if  $E1 E2 E3$ )  $\# env \downarrow V2$

Where  $V1$  is not  $\# f$

# Example Derivation with Environments

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

$$\begin{array}{c} \text{test } \# \text{env4} \downarrow \#t \text{ [varref]} \\ | \\ \text{x } \# \text{env4} \downarrow 3 \text{ [varref]} \\ | \\ 5 \# \text{env4} \downarrow 5 \text{ [value]} \\ \hline (\ast \text{ x } 5) \# \text{env4} \downarrow 15 \text{ [multiplication]} \\ | \\ \text{diff } \# \text{env4} \downarrow 9 \text{ [varref]} \\ \hline (+ (\ast \text{ x } 5) \text{ diff}) \# \text{env4} \downarrow 24 \text{ [addition]} \\ | \\ (\text{if test } (+ (\ast \text{ x } 5) \text{ diff}) \text{ 17}) \# \text{env4} \downarrow 24 \text{ [if nonfalse]} \end{array}$$

# Conclusion-below-subderivations, in text

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

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

$(\text{if test } (+ (* \text{x} 5) \text{diff}) 17) \# \text{env4} \downarrow 24$  [if nonfalse]

- $\text{test} \# \text{env4} \downarrow \#t$  [varref]
- $(+ (* \text{x} 5) \text{diff}) \# \text{env4} \downarrow 24$  [addition]
  - $(* \text{x} 5) \# \text{env4} \downarrow 15$  [multiplication]
    - $\text{x} \# \text{env4} \downarrow 3$  [varref]
    - $5 \# \text{env4} \downarrow 5$  [value]
  - $\text{diff} \# \text{env4} \downarrow 9$  [multiplication]

# 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,  $\Downarrow$ , to emphasize that definitions are not evaluated to values, but **processed to environments**.

$$\frac{E \ # \ env \Downarrow v}{(\text{define } Id \ E) \ # \ env \Downarrow Id \mapsto v, \ env} \text{ [define]}$$

# Threading environments through definitions

2 #  $\emptyset \downarrow 2$  [value]

3 #  $\emptyset \downarrow 3$  [value]

(+ 2 3) #  $\emptyset \downarrow 5$

[addition]

(define a (+ 2 3)) #  $\emptyset \Downarrow a \mapsto 5$

[define]

a #  $a \mapsto 5 \downarrow 5$  [varref]

a #  $a \mapsto 5 \downarrow 5$  [varref]

[multiplication]

(\* a a) #  $a \mapsto 5 \downarrow 25$

[define]

(define b (\* a a)) #  $a \mapsto 5 \Downarrow b \mapsto 25, a \mapsto 5$

b #  $b \mapsto 25, a \mapsto 5 \downarrow 25$  [varref]

a #  $b \mapsto 25, a \mapsto 5 \downarrow 5$  [varref]

[subtraction]

(- b a) #  $b \mapsto 25, a \mapsto 5 \downarrow 20$

# 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 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  $e \# env2 \downarrow v$  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;

$$\begin{aligned} & (- (* (+ 2 3) 9) (/ 18 6)) \\ \Rightarrow & (- (* 5 9) (/ 18 6)) \\ \Rightarrow & (- 45 (/ 18 6)) \\ \Rightarrow & (- 45 3) \\ \Rightarrow & 42 \end{aligned}$$

# 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:

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

The diagram shows a sequence of five reduction steps. In each step, a green box highlights a redex (a non-value subexpression), and a green arrow points from the previous step's redex to the current one. The redexes are:  $(+ 2 3)$ ,  $(* 5 9)$ ,  $(/ 18 6)$ ,  $(- 45 3)$ , and  $42$ . The annotations in blue indicate the operation being performed at each step: [addition], [multiplication], [division], and [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.

$Id \Rightarrow V$ , where  $Id \mapsto V$  is the first binding for  $Id$  in the current environment\* [varref]

$(+ V1 V2) \Rightarrow V$ , where  $V$  is the sum of numbers  $V1$  and  $V2$  [addition]

There are similar rules for other arithmetic relational operators

$(\text{if } Vtest Ethen Else) \Rightarrow Ethen$ , if  $Vtest$  is not  $\#f$  [if nonfalse]

$(\text{if } \#f Ethen Else) \Rightarrow Else$  [if false]

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

# 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  $\Rightarrow$  for  $\Rightarrow$
- 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:

$$\begin{aligned} & (- \ (\ast \ (+ \ 2 \ 3) \ \#t) \ (/ \ 18 \ 6) ) \\ \Rightarrow & (- \ (\ast \ 5 \ \#t) \ (/ \ 18 \ 6) ) \text{ Stuck!} \end{aligned}$$

$$\begin{aligned} & (\text{if } (= 2 (/ (+ 3 4) (- 5 5))) 8 9) \\ \Rightarrow & (\text{if } (= 2 (/ 7 (- 5 5))) 8 9) \\ \Rightarrow & (\text{if } (= 2 (/ 7 0)) 8 9) \text{ Stuck!} \end{aligned}$$



# 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)



# Solution #1

```
(if {(< 8 2)} (+ #f 5) (+ 3 4))  
=> { (if #f (+ #f 5) (+ 3 4))} [less than]  
⇒ {(* 3 4)} [if false]  
⇒ 12 [multiplication]
```

```
(if {(+ 1 2)} (- 3 7) (/ 9 0))  
=> (if 3 (- 3 7) (/ 9 0)) [addition]  
⇒ {(- 3 7)} [if nonfalse]  
⇒ -4 [subtraction]
```



# Solution #2

```
(+ (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]
```

```
(+ {((if 1 2 3)} #t)  
=> (+ 2 #t) [if nonfalse]
```

Stuck here! Can't apply  
[addition] rule because #t  
is not a number in (+ 2 #t).  
This models a dynamic type  
checking error.

# Racket Documentation

Racket Guide:

<https://docs.racket-lang.org/guide/>

Racket Reference:

<https://docs.racket-lang.org/reference>