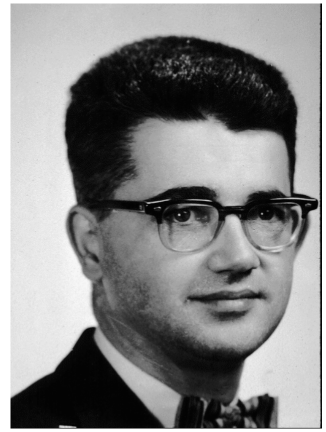
Introduction to Racket, a dialect of LISP: Expressions and Bindings

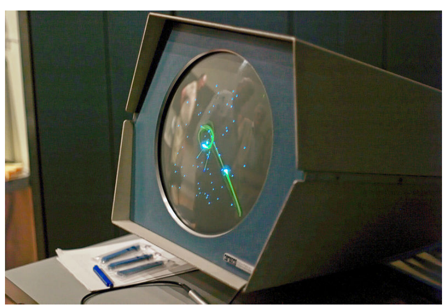
CS251 Programming Languages
Spring 2016, Lyn Turbak

Department of Computer Science
Wellesley College

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

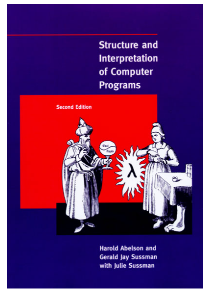
- Needed a language for:
  - Symbolic computation
  - Programming with logic
  - Artificial intelligence
  - Experimental programming

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 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: **`(+ 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.)

e1 ↓ ***v1***
e2 ↓ ***v2***
`(+` ***e1 e2*** `)` ↓ ***v*** (addition rule)

Where ***v1*** and ***v2*** are numbers and ***v*** is the sum of ***v1*** and ***v2***.

side conditions of rules

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


Errors 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

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

```
3 ↓ 3 (value)
    5 ↓ 5 (value)
    #f ↓ #f (value)
    ---------------
```

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 `(+` ***e1 e2*** … ***en***`)` as `(+ (+` ***e1 e2***`)` … ***en***`)`
  E.g., treat `(+ 7 2 -5 8)` as `(+ (+ (+ 7 2) -5) 8)`
- Treat `(+` ***e***`)` as ***e***
- Treat `(+)` as `0` (or say `(+)` ↓ `0`)

Other Arithmetic Operators

Similar syntax and evaluation for

`- * / quotient remainder`

except:

- Second argument of **`/`**, **`quotient`**, **`remainder`** must be nonzero
- Result of **`/`** is a rational number (fraction)
- **`quotient`** and **`remainder`** take exactly two arguments; anything else is an error.
- **`(- e)`** is treated as **`(- 0 e)`**
- **`(/ e)`** is treated as **`(/ 1 e)`**
- **`(*)`** evaluates to 1.
- **`(/)`** and **`(-)`** are errors.

Relation Operators

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

For example:

$$\frac{e1 \downarrow v1 \qquad e2 \downarrow v2}{(< \ e1 \ e2) \downarrow v} \text{ (less than rule)}$$

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 e1 e2 e3)`**

Evaluation rule:

1. Evaluate ***e1*** to a value ***v1***.
2. If ***v*** is not the value **`#f`** then
   return the result of evaluating ***e2***
   otherwise
   return the result of evaluating ***e3***

Conditional (if) expressions

$$\frac{e1 \downarrow v1 \qquad e2 \downarrow v2}{(\text{if } \ e1 \ e2 \ e3) \downarrow v2} \text{ (if nonfalse)}$$

e3 not evaluated!

where ***v1*** is not #f

$$\frac{e1 \downarrow \#f \qquad e3 \downarrow v3}{(\text{if } \ e1 \ e2 \ e3) \downarrow v3} \text{ (if false)}$$

e2 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)
```


Expressions vs. statements

If expressions can go anywhere an expression is expected:

```
(if (< 9 (- 251 240))
    (* 3 (+ 4 5))
    (+ 6 (* 7 8)))

(+ 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, …)

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: ***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)

Examples:

- Suppose ***env*** is `num` → `17`, `absoluteZero` → `-273`, `true` → `#t`
- In ***env***, `num` evaluates to `17`, `absoluteZero` evaluates to `-273`, and `true` evaluates to `#t`

define bindings

Syntax: **(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*** → ***v*** at the front.

Environments: Example

env0 = ∅

```
(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***

Evaluation Assertions & Rules with Environments

The **evaluation assertion** notation ***e*** # ***env*** ↓ ***v*** means ``Evaluating ***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 pas 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***.

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
```


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?

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