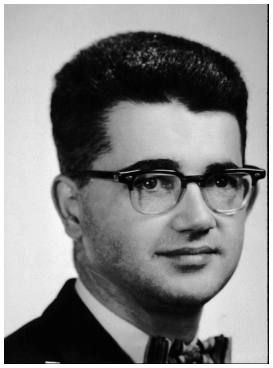
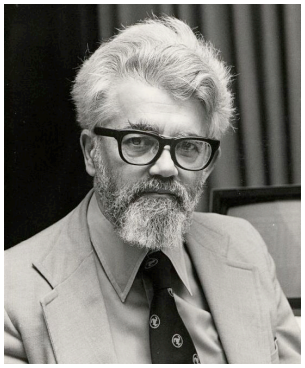
First Case Study: LISP and Racket

Designed by John McCarthy, 1958 published 1960





2

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

Racket

- Grandchild of LISP (via Scheme)
  - Some changes/improvements, quite similar
- Our goals:
  - Clean slate, unfamiliar
  - Careful study of PL foundations ("PL mindset")
  - Beauty of minimalism
  - Observe design constraints/historical context

Expressions, Values, and Bindings

- Entire language: these three things
- Expressions have *evaluation rules:*
  - How to reduce this expression to a value.
- 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`
- 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. `v1` = evaluate `e1` to a value
2. `v2` = evaluate `e2` to a value
3. Return the arithmetic sum of `v1 + v2`.

Not quite!

Addition expression: evaluation

Syntax: `(+ e1 e2)`

Evaluation rule:

More on this soon…

1. `v1` = evaluate `e1` to a value *in the current environment*
2. `v2` = evaluate `e2` to a value *in the current environment*
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

Other number expressions

Similar syntax and evaluation for:

`+ - * / quotient < > <= >= =`

Some small differences.

Build syntax and evaluation rules for: `/` and `>`

If expressions

Syntax: `(if e1 e2 e3)`

Evaluation rule:

1. Evaluate `e1` to a value `v1`.
2. If `v1` is not the value `#f` then
   evaluate `e2` and return the result
   otherwise
   evaluate `e3` and return the result

If expressions

If expressions are expressions.
They can go anywhere an expression is expected:

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

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

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)
```

Evaluation assertions, derivations

Environments

- ***Environment***:
  - A list of ***bindings*** mapping ***variable*** to ***value***.
- "Context" for evaluation.
- "Second argument" to evaluation rules.

Define bindings

Syntax: **`(define x e)`**

- ***`define`***: keyword
- ***`x`***: 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 mapping ***`x`*** to ***`v`***.

See define.rkt.

Variables

Syntax: ***`x`***

- ***`x`***: any *identifier*

Evaluation rule:

1. Look up and return the value to which ***`x`*** is bound in the current environment.
   - Look-up proceeds by searching from the most-recently added bindings to the least recently added bindings.

See define.rkt.

```
#lang racket

; Definitions introduce bindings into the dynamic environment, abbreviated env.
; Initially the dynamic environment is empty.
; env: = .

(define x 34)
; env: x –> 34, .

(define y 17)
; env: y –> 17, x –> 34, .

(define z (+ (+ x y) (+ y 2)))
; env: z –> 70, y –> 17, x –> 34, .

(define q (+ z 1))
; env: q –> 71, z –> 70, y –> 17, x –> 34, .

(define abs-of-z (if (< z 0) (- 0 z) z))
; env: abs-of-z –> 70, q –> 71, z –> 70, y –> 17, x –> 34, .
```