It’s dangerous to go alone…

Lisp

Scheme

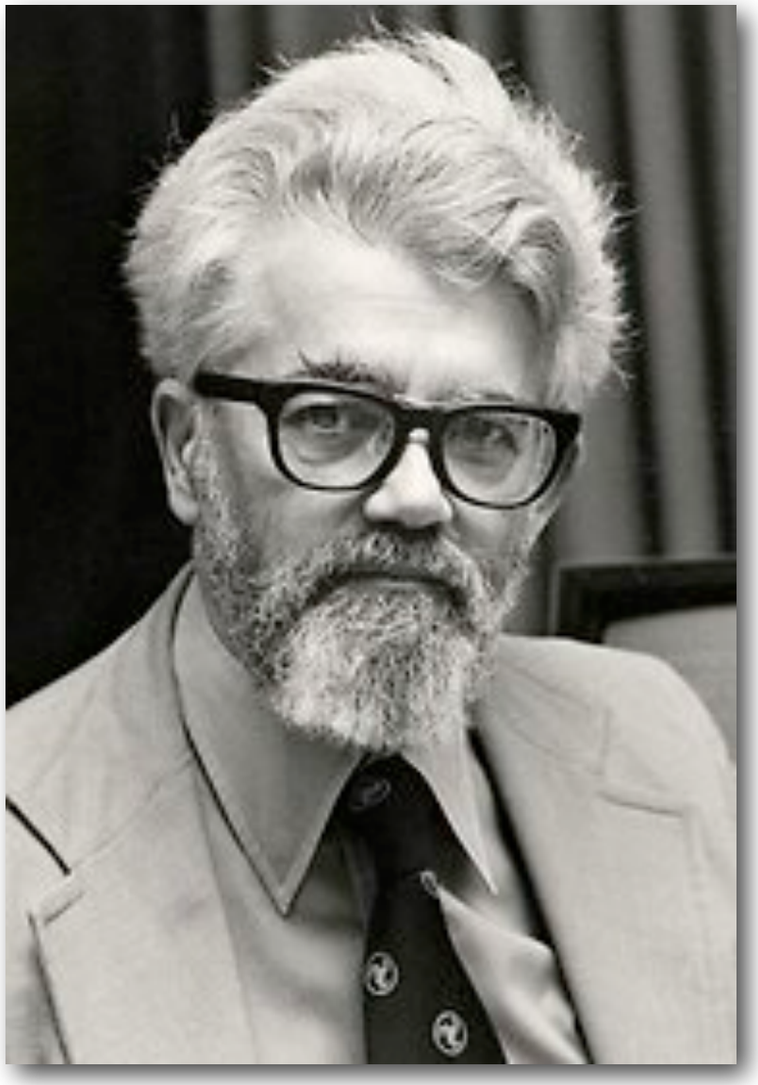
Racket

Administrative matters

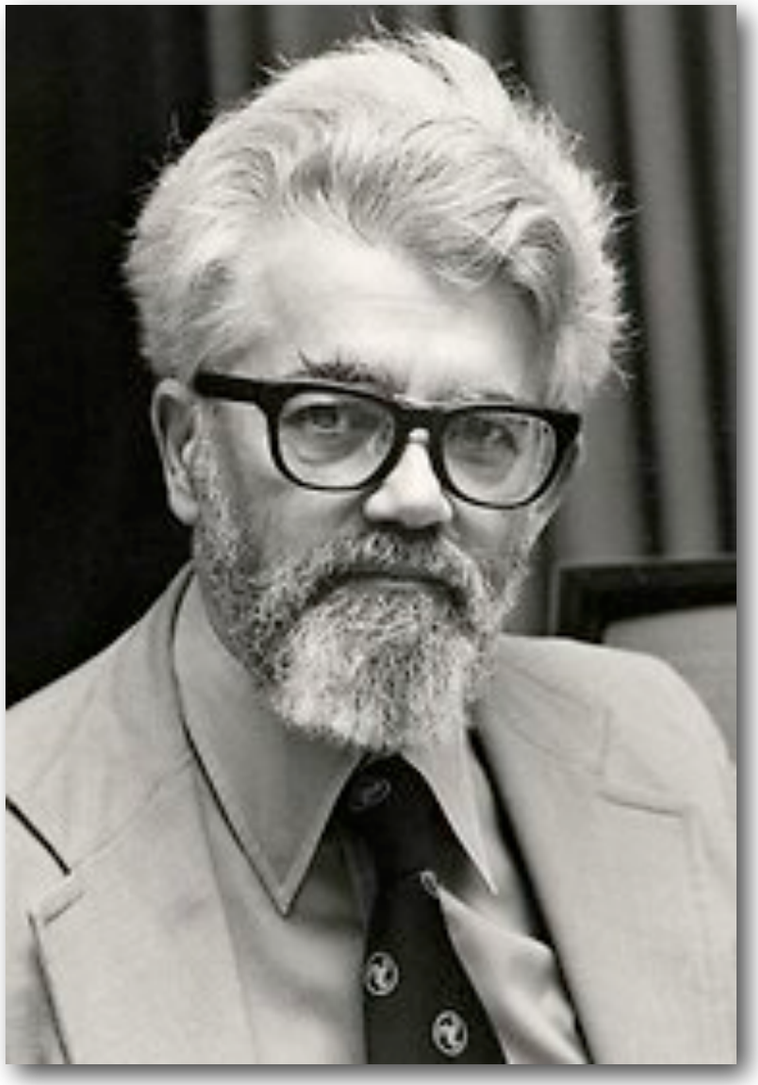
- Project out on Thursday
- Project to be due Feb 9th

A long time ago in a galaxy far,
far away....

Alonzo Church



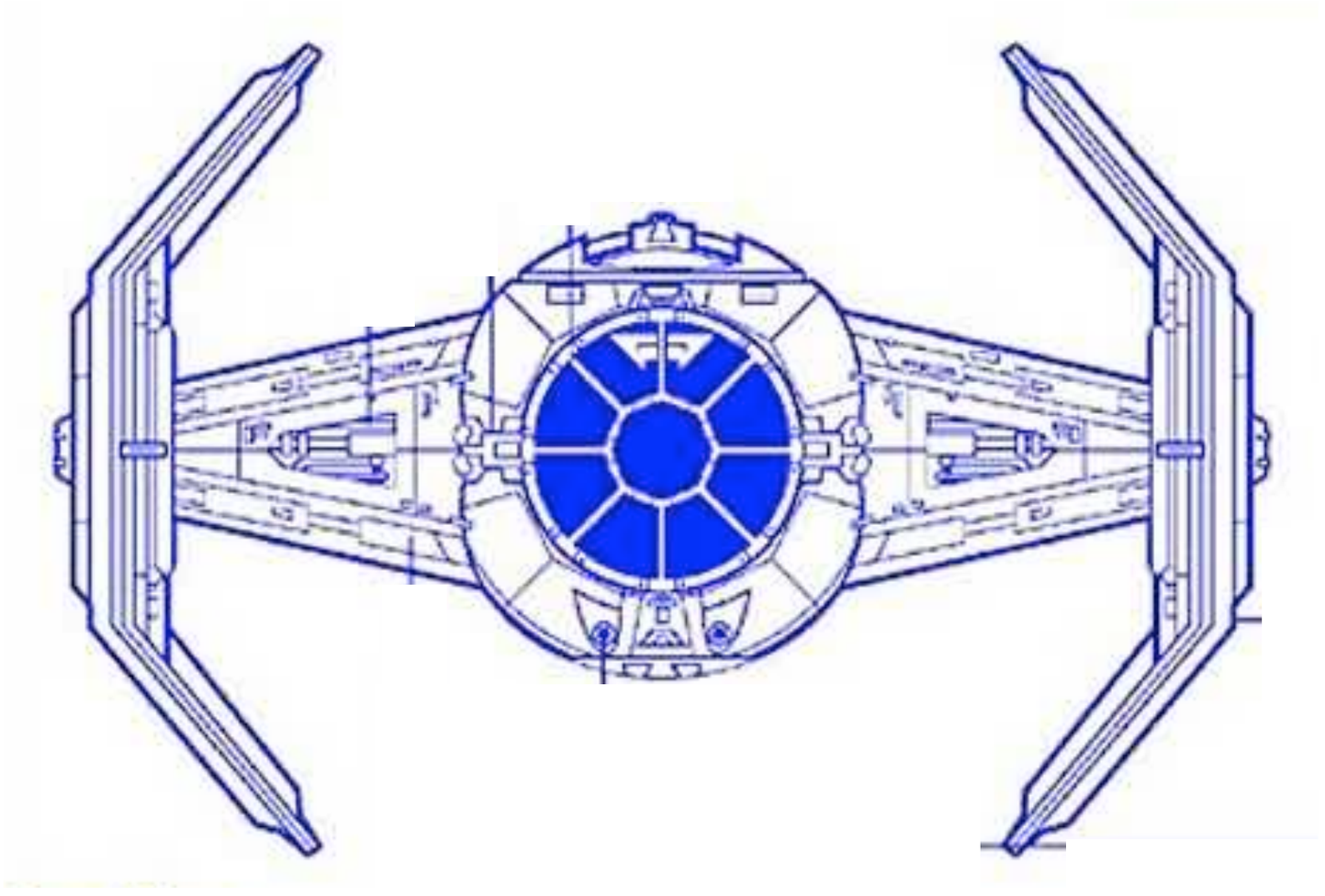
John McCarthy



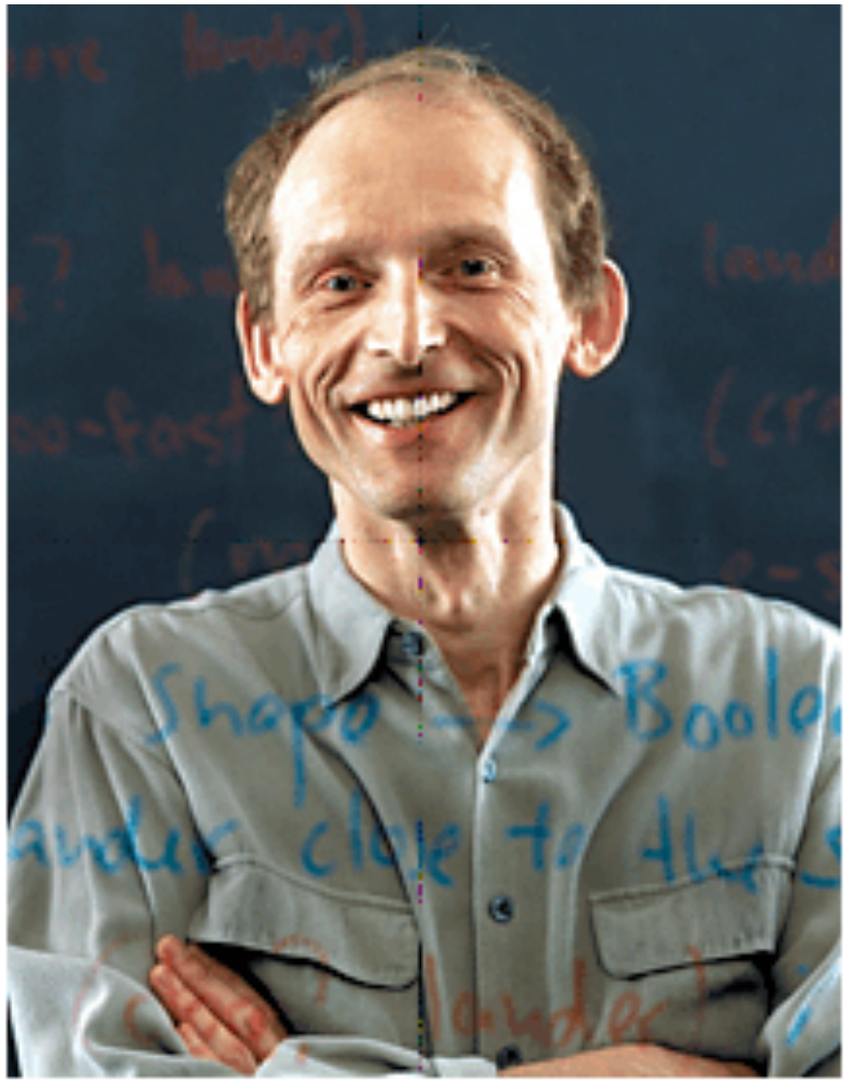
John McCarthy

Lisp

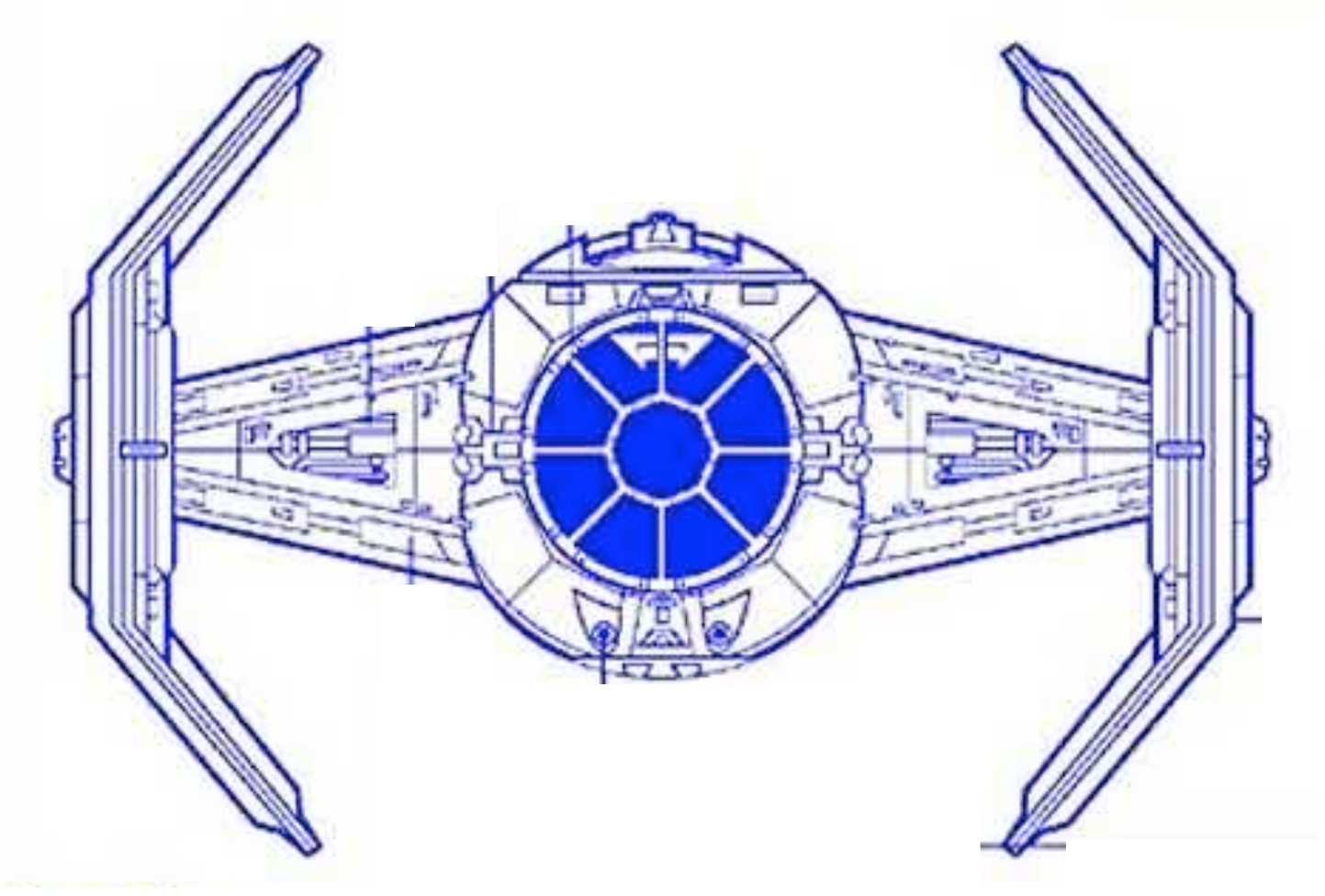
Guy Steele



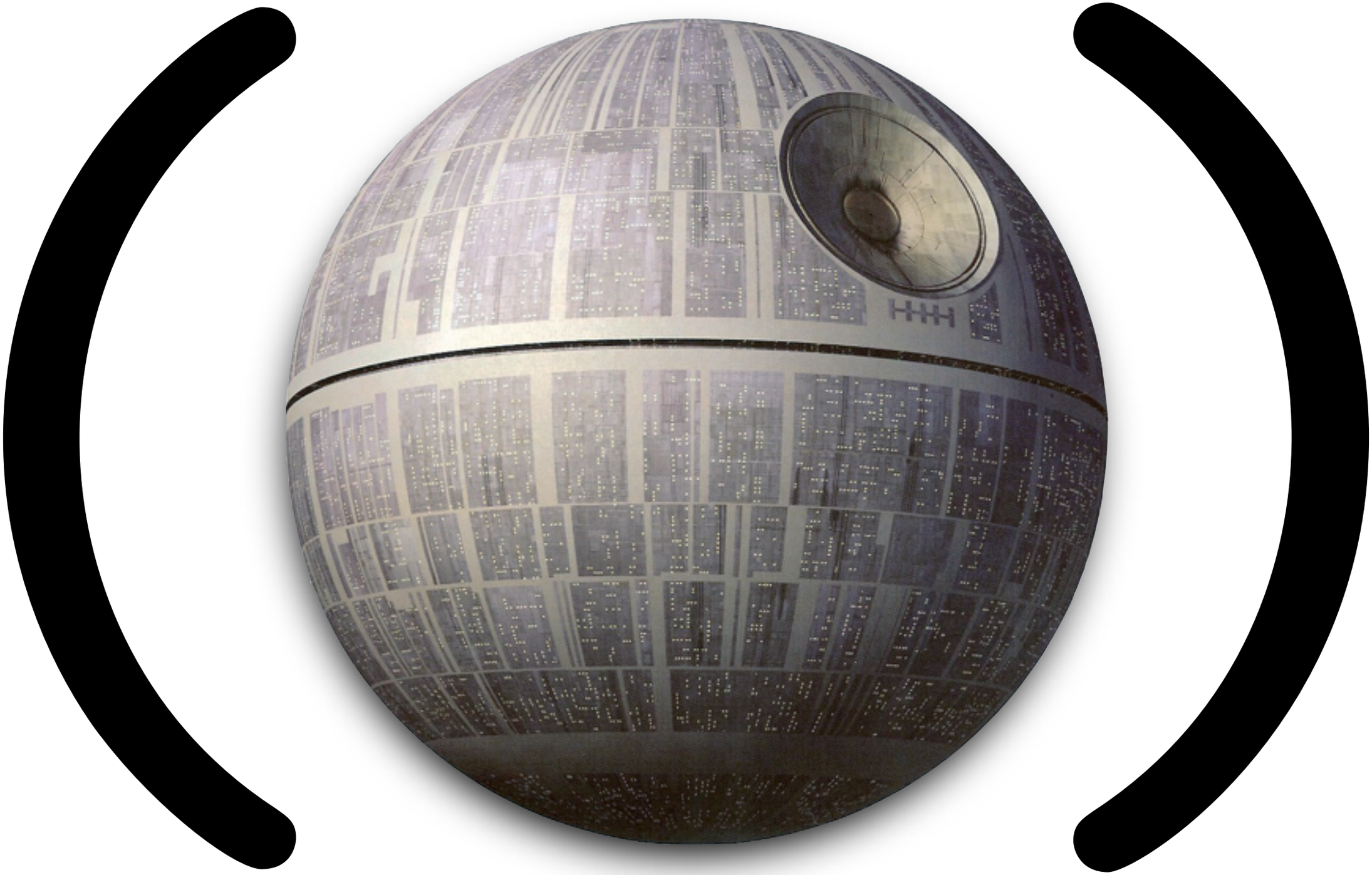
Scheme



Matthias Felleisen and Matthew Flatt



Scheme



Racket

Why Racket?

- Easy to parse
- Unbounded arithmetic
- Pattern matching (on ‘roids)
- Dynamic typing
- Mixed-hygiene macros
- Optional rich type system

Use DrRacket!

(and the REPL)


PARENTHESES
PARENTHESES EVERYWHERE

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

S-Expressions

Symbolic-Expressions

Textual encoding of trees

(node-type children)

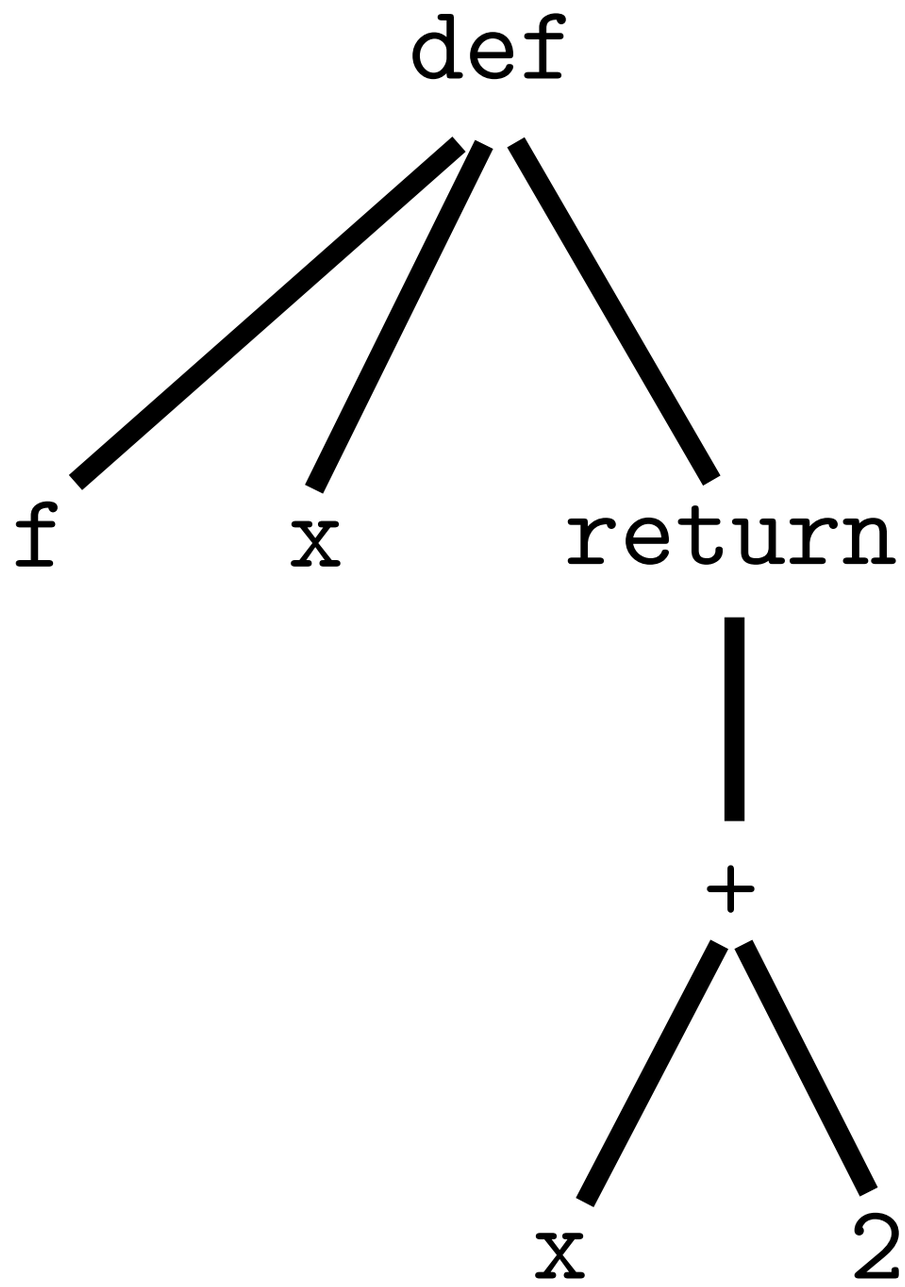
<node-type>

children

</node-type>

(node-type children)

```
def  f ( x ) :
return  x + 2
```


def
f
x
return
+
x
2

```
(def f (x)
  (return (+ x 2)))
```

Now, Racket

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

```
(define (fact n)
  (if (n . = . 0)
      1
      (n . * . (fact (n . - . 1)))))
```

```
(define (fact n)
  (if (n . = . 0)
      1
      (n . * . (fact (n . - . 1)))))
```

Programs = Definitions + Expressions

Expressions

Numbers

3.14

0+3.14i

Arithmetic

3 + 4

```
(+ 3 4)
```

```
(3 . + . 4)
```

```
(3 + 4) * 6
```

```
(* (+ 3 4) 6)
```

```
(gcd 10 4)
```

```
(sqrt -1)
```

```
(expt 10 200)
```

100000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000

Strings

"foo bar"

```
"foo
bar"
```

Symbols

```
'foo
```

'λ

$$\chi_{\alpha\beta\delta\alpha}$$

```
'pretty-much-anything-can-be-a-symbol
```

```
'you\ can\ even\ put\ spaces\ in\ symbols
```

```
'|you can even put spaces in symbols|
```

```
"you can even put spaces in symbols"
```

Great! So why not just use strings?

Symbols are “canonicalized.”

On symbols, value equality implies pointer equality.

Equality checks on symbols are constant time.

```
(string->symbol "foo")
```

```
'foo
```

```
(string->symbol "foo bar")
```

```
'foo\ bar
```

```
(symbol->string 'foo)
```

"foo"

Characters

```
#\f
```

#\newline

#\u*hex*

Booleans

#t

#f

```
(and #f #t)
```

```
(or #f #t)
```

(Linked) Lists

```
(list 1 2 3)
```

```
'(1 2 3)
```

(list)

' ( )

```
(first (list 1 2 3))
```

```
(first (list 1 2 3))  =  1
```

```
(rest (list 1 2 3))
```

```
(rest (list 1 2 3))   =   '(2 3)
```

Historical note

```
first  =  car
rest   =  cdr
```

```
(car (list 1 2 3))  =  1
```

```
(cdr (list 1 2 3))  =  '(2 3)
```

```
(cadr (list 1 2 3))       =
(car (cdr (list 1 2 3))) =  2
```

```
(list-ref (list 1 2 3) 1)
```

```
(list-ref (list 1 2 3) 1) = 2
```

```
(list-ref (list 1 2 3) 0) = 1
```

```
(list-ref (list 1 2 3) 2) = 3
```

```
(list-ref (list 1 2 3) 2) = 3
```

```
(append (list 1 2) (list 3))
```

```
(append (list 1 2) (list 3)) = '(1 2 3)
```

```
(append (list 1) (list 2 3)) = '(1 2 3)
```

```
(cons          1   (list 2 3)) = '(1 2 3)
```

```
(null? (list))
```

```
(null? (list)) = #t
```

```
(null? (list 1 2 3)) = #f
```

```
(pair? (list)) = #f
```

```
(pair? (list 1 2 3)) = #t
```

99 ways to say '(I love you)

In spite of their simplicity, lists often confound new Racket programmers.

After lists, the many forms for expressing a computation in Racket take time to discover and then master.

To address both of these points, I've created 99 roughly tweet-sized expressions in Racket that all evaluate to the list `'(I love you)`.

Conditionals

```
(if #t 'alpha 'beta)
```

'alpha

```
(if #f 'alpha 'beta)
```

'beta

```
(if (< 3 4) 'alpha 'beta)
```

```
(cond
 [(< 3 4) 'alpha]
 [else    'beta ])
```

```
(cond
 [(< 4 3) 'alpha]
 [(< 5 6) 'gamma]
 [else    'beta ])
```

Let-binding variables

```
(let ((x 10))
(+ x 20))
```

```
(let ([x 10])
  (+ x 20))
```

```
(let ([foo 10]
      [bar 20])
  (+ foo bar))
```

Identifiers

Just like symbols! But, no “quote.”

foo

pretty-much-anything-can-be-an-identifier

EVEN\ SPACES\ AND\ !

|EVEN SPACES AND !|

also-λ

Even `if` and `let`!

Even `if` and `let`!

```
(let ([if  10])
  if)
```

```
(let ([if  10]
      [let 20])
  (+ if let))
```


IS
THERE
NOTHING
SACRED??????

```
(let ([a 10]
      [b (+ a 3)])
  b)
```

```
(let ([a 10]
      [b (+ a 3)])
  b)
```

```
(let* ([a 10]
       [b (+ a 3)])
  b)
```

```
(let* ([a 10]
       [b (+ a 3)])   =   13
  b)
```

Globals

```
(define foo 10)
```

```
(define π 3.14)
```

```
(let ([if  10]
      [let 20])
  (+ if let))
```

Lambdas

```
(lambda (x) (+ x 1))
```

```
(λ (x) (+ x 1))
```

```
((λ (x) (+ x 1)) 10)
```

```
((λ (x) (+ x 1)) 10)  =  11
```

```
((λ (x y) (+ x y)) 10 20)  =  30
```

Functions

```
(define fact (λ (n)
                (if (= n 0)
                    1
                    (* n (fact (- n 1))))))
```

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

Quoting

```
'foo
```

'3

'#t

#t

```
'(1 2 3)
```

```
'(1 (2 3))
```

```
(cdr '(1 (2 3)))
```

```
(cdr '(1 (2 3))) = '((2 3))
```

```
(car (cdr '(1 (2 3))))
```

```
(car (cdr '(1 (2 3))))  =  '(2 3)
```

```
(car (car (cdr '(1 (2 3)))))
```

```
(car (car (cdr '(1 (2 3))))) = 2
```

```
'(def f (x)
   (return x))
```

I/O

```
$ cat sample.sx
(def (f x)
  (return x))
```

```
(read (open-input-file "test.sx"))
```