Scheme

Announcements

(Scheme)

Scheme is a Dialect of Lisp

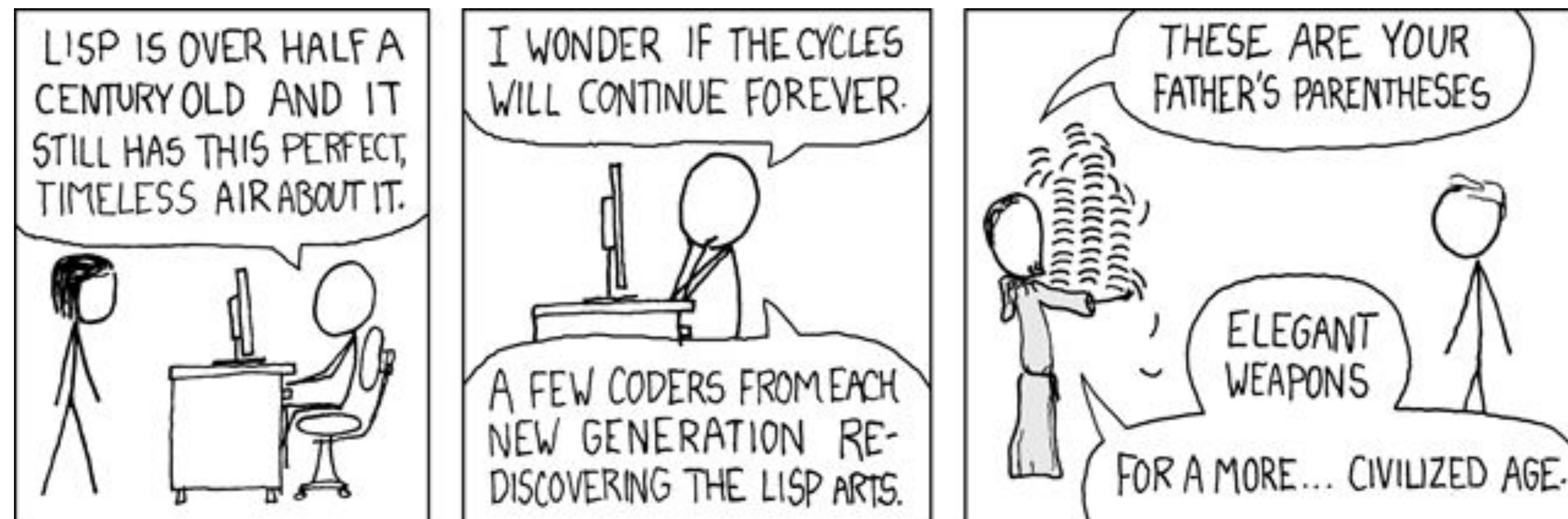
What are people saying about Lisp?

•"If you don't know Lisp, you don't know what it means for a programming language to be powerful and elegant."

- Richard Stallman, created Emacs & the first free variant of UNIX

•"The greatest single programming language ever designed."

-Alan Kay, co-inventor of Smalltalk and OOP



Why Scheme?

Scheme has heavily influenced the development of other languages like Python and still sees some use today.

The goal of this class is not to teach Python, but rather to teach a **variety of programming techniques** and core concepts

Contrasting Scheme and Python allows us to what **transfers between languages**.

- In both Python and Scheme, we can write functions and store data in lists
- Functional programming is present in pretty much all languages
- Primitive values and evaluation

Contrasting Scheme and Python allows us to see **what paradigms certain languages prioritize**:

- Imperative languages - languages that compute by changing the program state
- Functional languages - computation as a series of function applications
- Declarative languages - describes **what not how** a computation should perform

We will use Scheme to learn **how interpreters are implemented**! Scheme is an easier language to parse than Python given its minimal syntax.

Important Scheme Resources

In CS 61A we teach our own dialect of Scheme—this means that there may be slight differences between the behavior of our version of Scheme and other versions that you find online, so we recommend that you primarily stick to our resources for reference

Important Scheme resources for CS 61A:

- code.cs61a.org - Has a live Scheme interpreter, and a Scheme text editor (feel free to play around with these during the lecture!)
- cs61a.org/articles/scheme-spec/ - Full specification for the CS 61A dialect of Scheme (accessible on the Resources page of the course site)
- cs61a.org/articles/scheme-builtins/ - Built-in procedure reference (also accessible on the resources page)

Scheme Expressions

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2   3.3   true   +   quotient
- Combinations: (quotient 10 2)   (not true)

Numbers are self-evaluating; symbols are bound to values

Call expressions include an operator and 0 or more operands in parentheses

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Combinations can span multiple lines (spacing doesn't matter)

(Demo)

Built-in arithmetic procedures

+	(+ 1 2 3)
-	(- 12) (- 3 2 1)
*	(*) (* 2) (* 2 3)
/	(/ 2) (/ 4 2) (/ 16 2 2)
quotient	(quotient 7 3)
abs	(abs -12)
expt	(expt 2 10)
remainder	(remainder 7 3) (remainder -7 3)

cs61a.org/articles/scheme-builtins/#arithmetic-operations

Built-in boolean procedures (for numbers)

These procedures only work on numbers

`=`	`(= 4 4) (= 4 (+ 2 2))`
`<`	`(< 4 5)`
`>`	`(> 5 4)`
`<=`	`(<= 4 5) (<= 4 4)`
`>=`	`(>= 5 4) (>= 4 4)`
`even?`	`(even? 2)`
`odd?`	`(odd? 3)`
`zero?`	`(zero? 0) (zero? 0.0)`

cs61a.org/articles/scheme-builtins/#on-numbers

Built-in boolean procedures

`eq`	`(eq? #t #t)` `(eq? 0 (- 1 1))`	`(eq? #t #f)` `(eq? 0 0.0)`
`not`	`(not #f)`	`(not 0)` `(not #t)`
`list?`	`(list? '(1 2))`	`(list? 0)`
`null?`	`(null? nil)`	`(null? 'a)`

Fun/important fact: the only falsey data value in Scheme is #f—all other data values are truthy

cs61a.org/articles/scheme-builtins/#general

Special Forms

Special Forms

A combination that is not a call expression is a special form:

- **if** expression: (if <predicate> <consequent> <alternative>)
- **and** and **or**: (and <e1> ... <en>), (or <e1> ... <en>)
- Binding symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)

Evaluation:

(1) Evaluate the predicate expression

(2) Evaluate either the consequent or alternative

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The symbol “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

A procedure is created and bound to the symbol “abs”

(Demo)

Scheme Interpreters

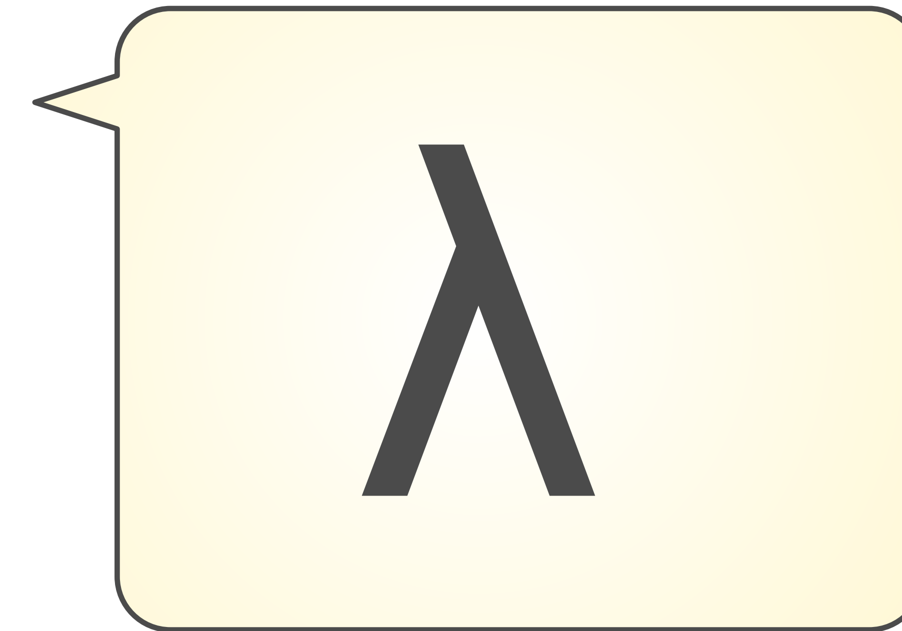
(Demo)

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```



Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a call expression too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

▶ 12

Evaluates to the $x+y+z^2$ procedure

More Special Forms

Cond & Begin

The cond special form that behaves like if-elif-else statements in Python

```
if x > 10:
    print('big')
elif x > 5:
    print('medium')
else:
    print('small')
```

```
(cond ((> x 10) (print 'big))
      ((> x 5)  (print 'medium))
      (else     (print 'small)))
```

```
(print
  (cond ((> x 10) 'big)
        ((> x 5)  'medium)
        (else     'small)))
```

The begin special form combines multiple expressions into one expression

```
if x > 10:
    print('big')
    print('guy')
else:
    print('small')
    print('fry')
```

```
(cond ((> x 10) (begin (print 'big)   (print 'guy)))
      (else     (begin (print 'small) (print 'fry))))
```

```
(if (> x 10) (begin
               (print 'big)
               (print 'guy))
             (begin
               (print 'small)
               (print 'fry)))
```

Let Expressions

The let special form binds symbols to values temporarily; just for one expression

```
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)
```

*a and b are **still** bound down here*

```
(define c (let ((a 3)
                (b (+ 2 2)))
               (sqrt (+ (* a a) (* b b)))))
```

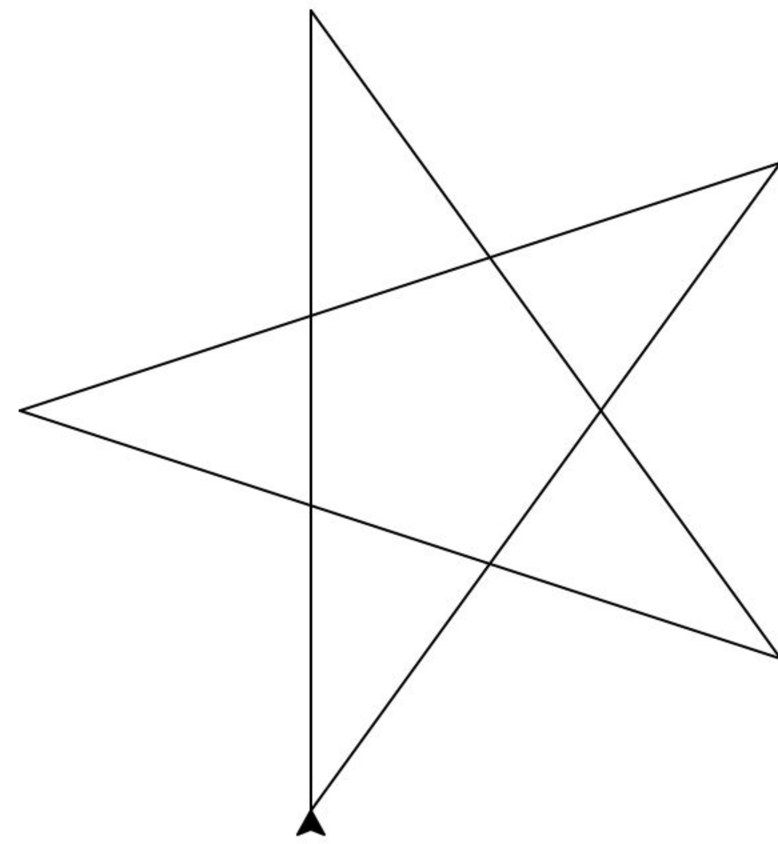
*a and b are **not** bound down here*

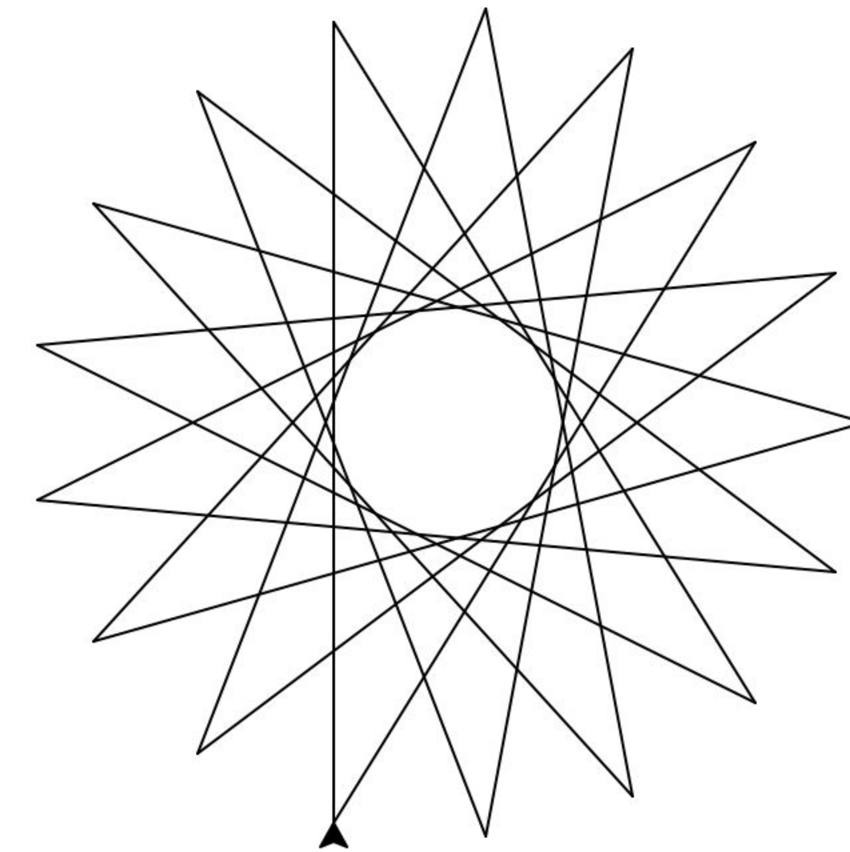
Turtle Graphics

Drawing Stars

(forward 100) or (fd 100) draws a line

(right 90) or (rt 90) turns 90 degrees





(Demo)

Sierpinski's Triangle

(Demo)

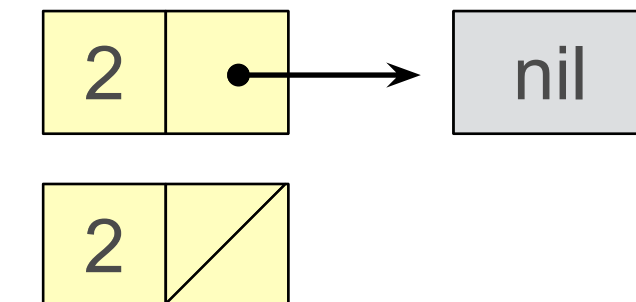
Break

Lists

Scheme Lists

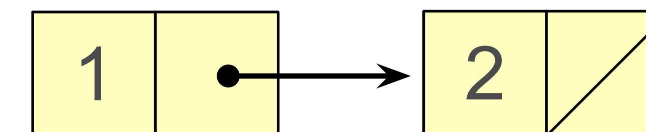
In the late 1950s, computer scientists used confusing names

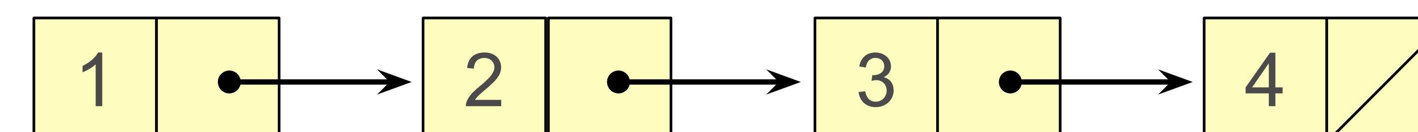
- **cons**: Two-argument procedure that creates a linked list
- **car**: Procedure that returns the first element of a list
- **cdr**: Procedure that returns the rest of a list
- **nil**: The empty list



Important! Scheme lists are written in parentheses with elements separated by spaces

```
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```





(Demo)

Built-in list procedures

cs61a.org/articles/scheme-builtins/#general

list?	(list? '(1 2))	(list? 0)
null?	(null? nil)	(null? 'a)

length	(length '(1 3 5))	3
list	(list 1 2 3)	(1 2 3)
list	(append '(1) '(2) '(3 4))	(1 2 3 4)
map	(map odd? '(2 2 2))	(#f #f #f)
filter	(filter even? '(1 2 3))	(2)
reduce	(reduce quotient '(13 3 2))	2

Symbolic Programming

Symbolic Programming

Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Short for (quote a), (quote b):
Special form to indicate that the expression itself is the value.

Quotation can also be applied to combinations to form lists.

```
> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

(Demo)

List Processing

Built-in List Processing Procedures

(**append s t)**: list the elements of s and t; append can be called on more than 2 lists

(**map f s)**: call a procedure f on each element of a list s and list the results

(**filter f s)**: call a procedure f on each element of a list s and list the elements for which a true value is the result

(**apply f s)**: call a procedure f with the elements of a list as its arguments

(Demo)

Example: Even Subsets

Even Subsets

Definition: a *non-empty subset* of a list **s** is a list containing some of the elements of **s**.

(A *non-empty subset* could contain all the elements of s, but not none of them.)

```
;;; Non-empty subsets of integer list s that have an even sum
;;;
;;; scm> (even-subsets '(3 4 5 7))
;;; ((5 7) (4 5 7) (4) (3 7) (3 5) (3 4 7) (3 4 5))
(define (even-subsets s) ... )
```

A recursive approach: The even subsets of s include...

- all the even subsets of the rest of s
- the first element of s followed by an (even/odd) subset of the rest
- just the first element of s if it is even

(Demo)

Discussion Question: Even Subsets Using Filter

Discussion Question: Complete this implementation of even-subsets

Definition: a *non-empty subset* of a list **s** is a list containing some of the elements of **s**.

(A *non-empty subset* could contain all the elements of s, but not none of them.)

```
;;; non-empty subsets of s
(define (nonempty-subsets s)
 (if (null? s) nil

   (let ((rest  (nonempty-subsets (cdr s)) ))
     (append rest
         (map (lambda (t) (cons (car s) t)) rest)
          (list (list (car s) ) ) ) ) ) )

;;; non-empty subsets of integer list s that have an even sum
(define (even-subsets s)

 (filter (lambda (s) (even? (apply + s))) (nonempty-subsets s)))
```