CSE 341
Lecture 15

introduction to Scheme

slides created by Marty Stepp
http://www.cs.washington.edu/341/

Looking back: Language timeline

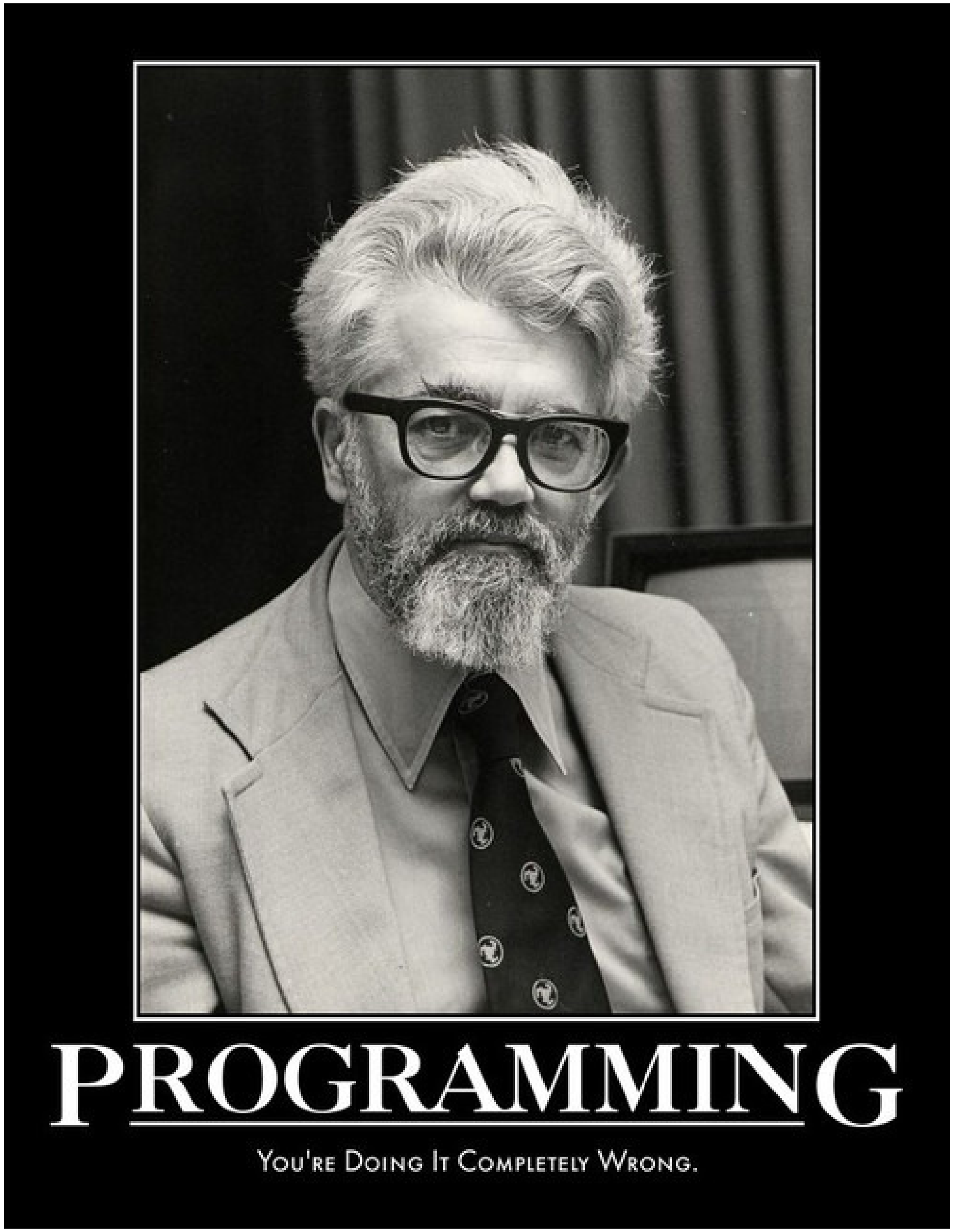
category	**1960s**	**1970s**	**1980s**	**1990s**	**2000s**
scientific	Fortran			Matlab	
business	Cobol	DBMSes	SQL	VB	
functional	**Lisp**	ML, **Scheme**	Erlang	Haskell	F#
imperative/ procedural	Algol	Pascal, C, Smalltalk	Ada, C++	Java	C#
scripting	BASIC		Perl	Python, Ruby, PHP, JavaScript	
logical		Prolog	CLP(R)		

History of LISP

- **LISP** ("List Processing"): The first functional language.
  - made: 1958 by John McCarthy, MIT (Turing Award winner)
    - godfather of AI (coined the term "AI")
  - developed as a math notation for proofs about programs
  - pioneered idea of a program as a collection of functions
  - became language of choice for **AI programming**

- Fortran (procedural, 1957), LISP (functional, 1958)
  - languages created at roughly the same time
  - battled for dominance of coder mindshare
  - Fortran "won" because LISP was slow, less conventional

John McCarthy, creator of LISP



LISP key features

- a functional, **dynamically typed**, type-safe, language
  - anonymous functions, closures, no return statement, etc.
  - less compile-time checking (run-time checking instead)
  - accepts more programs that ML would reject
- fully parenthesized syntax ("**s-expressions**")
  - Example:

    ```
    (factorial (+ 2 3))
    ```
- **everything is a list** in LISP (even language syntax)
  - allows us to manipulate **code as data** (powerful)
  - first LISP compiler was written in LISP

LISP advanced features

- LISP was *extremely* advanced for its day (and remains so):
  - recursive, first-class functions ("procedures")
  - dynamic typing
  - powerful macro system
  - ability to extend the language syntax, create dialects
  - programs as data
  - garbage collection
  - continuations: capturing a program in mid-execution

- It took other languages 20-30 years to get these features.

LISP "today"

- current dialects of LISP in use:
  - Common LISP (1984) - unified many older dialects
  - **Scheme** (1975) - minimalist dialect w/ procedural features
  - Clojure (2007) - LISP dialect that runs on Java JVM
- well-known software written in LISP:
  - Netscape Navigator, v1-3
  - Emacs text editor
  - movies (*Final Fantasy*), games (*Jak and Dexter*)
  - web sites, e.g. reddit
  - Paul Graham (tech essayist, *Hackers and Painters*)

Scheme

- **Scheme**: Popular dialect of LISP.
  - made in 1975 by Guy Steele, Gerald Sussman of MIT
  - Abelson and Sussman's influential textbook:
    - *Structure and Interpretation of Computer Programs* (SICP) http://mitpress.mit.edu/sicp/

- innovative differences from other LISP dialects
  - **minimalist** design (50 page spec), derived from λ-calculus
  - the first LISP to use **lexical scoping** and block structure
  - lang. spec forces implementers to optimize **tail recursion**
  - **lazy evaluation**: values are computed only as needed
  - first-class **continuations** (captures of computation state)

TeachScheme!

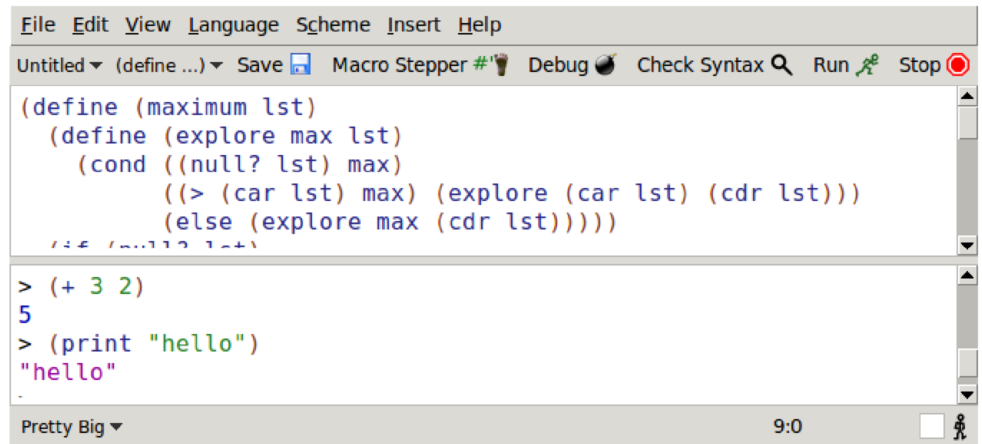
- 1995 movement by Matthias Felleisen of Rice's PLT group
  - goal: create pedagogic materials for students and teachers to educate them about programming and Scheme
  - push for use of Scheme and functional langs. in intro CS
  - radical yahoos who take themselves too seriously :-)
- major TeachScheme! developments
  - DrScheme editor, for use in education
  - *How to Design Programs*, influential Scheme intro textbook

http://www.teach-scheme.org/

http://www.htdp.org/

DrScheme

- **DrScheme**: an educational editor for Scheme programs
  - built-in interpreter window
    - Alt+P, Alt+N = history
  - syntax highlighting
  - graphical debugger
  - multiple "language levels"
    - (set ours to "Pretty Big")



- similar to DrJava editor for Java programs

(you can also use a text editor and command-line Scheme)

Scheme data types

- numbers
  - integers: `42` `-15`
  - rational numbers: `1/3` `-3/5`
  - real numbers: `3.14` `.75` `2.1e6`
  - complex/imaginary: `3+2i` `0+4i`
- text
  - strings: `"\"Hello\", I said!"`
  - characters: `#\X` `#\q`
- boolean logic: `#t` `#f`
- lists and pairs: `(a b c)` `'(1 2 3)` `(a . b)`
- symbols: `x` `hello` `R2D2` `u+me`

Basic arithmetic procedures

(***procedure arg1 arg2 ... argN***)

- in Scheme, almost every non-atomic value is a procedure
  - even basic arithmetic must be performed in `()` prefix form

- Examples:
  - `(+ 2 3)` → `5` `; 2 + 3`
  - `(- 9 (+ 3 4))` → `2` `; 9 - (3 + 4)`
  - `(* 6 -7)` → `-42` `; 6 * -7`
  - `(/ 32 6)` → `16/3` `; 32/6 (rational)`
  - `(/ 32.0 6)` → `5.333...` `; real number`
  - `(- (/ 32 6) (/ 1 3))` → `5` `; 32/6 - 1/3 (int)`

More arithmetic procedures

`+`	`-`	`*`
`quotient`	`remainder`	`modulo`
`max`	`min`	`abs`
`numerator`	`denominator`	`gcd`
`lcm`	`floor`	`ceiling`
`truncate`	`round`	`rationalize`
`expt`		

- Java's `int` / and % are quotient and modulo
  - `remainder` is like `modulo` but does negatives differently
- `expt` is exponentiation (pow)

Defining variables

`(define` ***`name expression`***`)`

- Examples:
  - `(define x 3)` `; int x = 5;`
  - `(define y (+ 2 x))` `; int y = 2 + x;`
  - `(define z (max y 7 3))` `; int z = Math.max..`

- Unlike ML, in Scheme all top-level bindings are mutable!

`(set!` ***`name expression`***`)`

  - `(set! x 5)`
    – (Legal, but changing bound values is discouraged. Bad style.)

Procedures (functions)

```
(define (name param1 param2 ... paramN)
        (expression))
```

- defines a procedure that accepts the given parameters and uses them to evaluate/return the given expression

```
> (define (square x) (* x x))
> (square 7)
49
```

  - in Scheme, all procedures are in curried form

Basic logic

- `#t, #f`   `; atoms for true/false`

- <, <=, >, >=, = operators (as procedures);   `equal?`
  - `(< 3 7)`   `; 3 < 7`
  - `(>= 10 (* 2 x))`   `; 10 >= 2 * x`

- and, or, not (also procedure-like; accept >=2 args) *

```
> (or (not (< 3 7)) (>= 10 5) (= 9 6))
#t
```

(technically and/or are not procedures because they don't always evaluate all of their arguments)

The if expression

(if ***test trueExpr falseExpr***)

- Examples:

```
> (define x 10)
> (if (< x 3) 10 25)
25
> (if (> x 6) (* 2 4) (+ 1 2))
8
> (if (> 0 x) 42 (if (< x 100) 999 777))  ; nested if
999
```

The cond expression

(cond (***test1 expr1***) (***test2 expr2***)
   ... (***testN exprN***))

- set of tests to try in order until one passes (nested if/else)

```
> (cond ((< x 0) "negative")
        ((= x 0) "zero")
        ((> x 0) "positive"))
"positive"
```

- parentheses can be [ ]; optional `else` clause at end:

```
> (cond [(< x 0) "negative"]
        [(= x 0) "zero"]
        [else "positive"])
"positive"
```

Testing for equality

- `(eq? `***`expr1 expr2`***`)` `; reference/ptr comparison`
- `(eqv? `***`expr1 expr2`***`)` `; compares values/numbers`
- `(= `***`expr1 expr2`***`)` `; like eqv; numbers only`
- `(equal? `***`expr1 expr2`***`)` `; deep equality test`

  - `(eq? 2.0 2.0)` is #f, but `(= 2.0 2.0)` and `(eqv? 2.0 2.0)` are #t
  - `(eqv? '(1 2 3) '(1 2 3))` is #f, but `(equal? '(1 2 3) '(1 2 3))` is #t

  - Scheme separates these because of different speed/cost

Scheme exercise

- Define a procedure `factorial` that accepts an integer parameter *n* and computes *n*!, or 1*2*3*...*(*n*-1)*n .
  - `(factorial 5)` should evaluate to 5*4*3*2*1, or `120`

- solution:

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

List of Scheme keywords

`=>`	`do`	`or`
`and`	`else`	`quasiquote`
`begin`	`if`	`quote`
`case`	`lambda`	`set!`
`cond`	`let`	`unquote`
`define`	`let*`	`unquote-splicing`
`delay`	`letrec`	

- Scheme is a small language; it has few reserved words