Racket

September-14-10 12:35 PM

RedExp:

Reduce Expression

Eg. (+ 1 1)

REPL:

Read, Evaluate, Print Loop

Racket is a dialect of Scheme
Contains a number of sub dialects
How to Design Programs uses teaching languages or dialects
"training wheels"

Function notation

+: number X number X .... --> number
The domain is some dimension of numbers and returns a number

Any computation that can be computed can be expressed as a single expression. A

(+ 1 1)
Adds 1 and 1

(+ 1 2 3 4 5)
Adds 1 through 5

(functions arg arg arg arg ...)

(* (+ 1 1) 2)

s-expressions or RedExp

Semicolons - ; - are the comment character.
2 is standard

Code is saved in the definitions (top) window. Bottom is command-line style

(define VAR_NAME REDEXP) - defines a variable, specifically a named constant

(number? 10) evaluates to true. Checks data type of argument passed

(number? true) evaluates to false

'dog - a symbol. Pronounced "quote dog". Different than strings
(symbol? 'dog)
(symbol=? 'dog 'dog) - true
(symbol=? 'dog 'cat) - false checks if two symbols are the same

(< 1 2) - boolean expression
(if CONDITION IF_TRUE IF_FALSE)

(define (double x) (+ x x))
Function definition
Consumes x and returned x doubled

(exact? 2/3) A predicate that tells you whether the number was exactly calculated.

Models of Computation

September-17-10 2:48 PM

Alan Turing - Turing Machine

The Turing machine is a model of computation.
A machine which can read, write, and exist in a certain state, as well as a tape of infinite length.

Can only read, write something and change state - depending on the current state, move the tape and repeat.

Can compute any computable result. However, some programs are impossible to compute on a Turing machine, such as the halting problem - whether a given program with halt on a given input.

Finite State Machine

A Turing Machine with finite memory. Aka a Finite Automaton

λ - Calculus

Invented by Church
A very simple method of expressing functions.

Syntax: A λ-expression:

- x (variable)
- e_1, e_2 (e_1, e_2 are λ - expressions)
- λx · e (function with variable x, e is a λ-expression) - x is substituted into function
    - Ex. (λx · X y z) w ⇒ w y z
- λx · λy · x "true"
- λx · λy · y "false"
- "If true then a else b"
    - (λx · λy · x) a b ⇒ (λy · a) b ⇒ a
    - (λx · λy · y) a b ⇒ (λy · y) b ⇒ b
- (λt · λa · λb · t a b) (Condition If_true If_false) "if"
- (λx x x) (λy y y) "infinite loop"

Combinator Theory

Using 3 functions, K, S, and I.
You can write any function with just K, S, and I
Combinator expression:

K
S
T
$e_1 \ e_2$

RAM Model (e.g. C/C++)

You have infinite RAM, which can be indexed. You can read and write to a given index in RAM.

Functions, Substitutions, Recursion

September-21-10 11:27 AM

Sign Function (Signum)

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

In scheme:

- (sgn -10) ;; -1
- (sgn 10) ;; 1
- (sgn 0) ;; 0

Define new signum function:

```
;; signum: number -> number
;;returns -1 if number is negative; +1 if positive; 0 if 0
;; example (signum 99) is 1

(define (signum n)
	(if (< n 0) -1
		(if (> n 0) 1 0)))

(signum -876);; -1
(signum 0);; 0
(signum 77);; 1
```

Nested ifs can be replaced by cond:

```
(define (signum n)
	(cond
		[(< n 0) -1]
		[(> n 0) 1]
		[(= n 0) 0])) or [else 0]))
```

[] brackets are treated the same as () but conventionally used in conditional statements.

Factorial

$$n! = \begin{cases} 1, & n = 0 \\ n \times (n-1)!, & n > 0 \end{cases}, n \in \mathbb{N}$$

```
(define (fact n)
	(cond
		[(= n 0) 1]
		[true (* n (cond (- n 1)))]))
```

```
;; returns m× k!
(define (fact-times m k)
	(cond
		[(= k 0) m]
		[else (fact-times (* m k) ( - k 1))]))
```

The advantage to the second function is that it keeps the expression constant. (fact n) produces an expression of size n.

<u>Complexity</u>

Number of steps for factorial:

a + bn

Linear time algorithm

Digits

```
;; digits: number -> number
;; returns the number of digits in a given number

(digits 2146); 4
(digits 0); 1
(digits 42) ;2
```

$$(digits\ x) = \begin{cases} 1, & -10 < x < 10 \\ 1 + \left(digits\ \left(\frac{x}{10}\right)\right), & |x| \geq 10 \end{cases}$$

```
(define (digits n)
 (if (< -10 n 10)
   1
   (+ (digits(quotient n 10)) 1)))
```

Tail Recursion

Ex.

```
(define (fibo n)
	(case
		[(= n 0) 0]
		[(= n 1) 1]
		[else (+ (fibo (- n 1)) (fibo (- n 2)))]))
```

Data Structures

September-21-10 12:41 PM

```
(define-struct couple (him her))
;; This defines several statements:

(make-couple exp1 exp2)

(define franks (make-couple (+ 1 2) (+ 3 4)))

(couple-him franks)  ; 3
(couple-her franks) ; 7
(couple? franks) ; true
(couple? 42) ; false

(define fronks (make-couple franks 42))
```

Franks

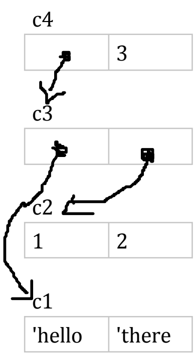
3	7

Fronks

Franks 3 | 7	42

Instead of nested boxes we use box-and-pointer diagrams.

```
(define-struct couple (a b))
(define c1 (make-couple 1 2))
(define c2 (make-couple 'hello 'there))
(define c3 (make-couple c1 c2))
(define c4 (make-couple c3 3))
```



Example use of Structures

Quadratic Field

$a + b\sqrt{r}$

```
(define-struct quadratic-field (a b r))

(define (add-qf a b)
        (if (= (quadratic-field-r a) (quadratic-field-r b))
                (make-quadratic-field (+ (quadratic-field-a a) (quadratic-field-a b))
                                        (+ (quadratic-field-b a)(quadratic-field-b b))
                                        r)
                0))

(define (mult-qf a b)
        (if (= (quadratic-field-r a) (quadratic-field-r b))
                (make-quadratic-field
                        (+ (* (quadratic-field-a a) (quadratic-field-a b))
                           (* (quadratic-field-b a) (quadratic-field-b b) (quadratic-field-r a)))
                        (+ (* (quadratic-field-a a) (quadratic-field-b b))
                           (* (quadratic-field-b a) (quadratic-field-a b)))
                        r)
                0))
```

Structures are new data types and will be incompatible with most other premade functions.

```
(define-struct mark (m))

(define mark->number mark-m)
(define (mark-number x) (mark-m x))
```

Those two statements are equivalent

Abstraction: "bunch of numbers"

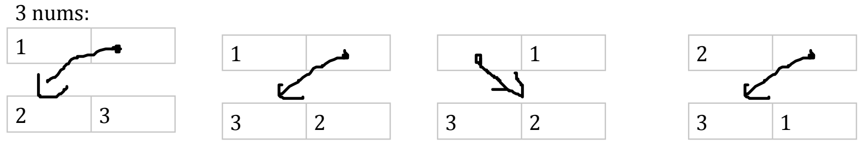
{1, 2, 3}
{1, 5, 1, 6}
{1, 1, 5, 6}

Multiset with 2 or more numbers

2 nums:

1	3

3 nums:



... etc. Many different representations.

This are all members of an equivalence class. Can choose a specific member of the equivalence class for representation. This is known as a canonical representation.

In this case, we chose the representation in which the numbers are stored in order and pointers originate from the 2nd element.

```
(define-struct bunch-pair (a b))

;; bunch twin: num X num -> bunch
;; returns a bunch containing exact n and m
(define (bunch-twin n m)
        (make-bunch-pair m n))

(define (bunch-triple n m q)
        (make-bunch-pair n (make-bunch-pair m q)))

(define (bunch-many n)
   (cond
        [( = n 2) (make-bunch-pair 1 2)]
        [else (make-bunch-pair (bunch-many (- n 1)) n)]))

(define (bunch-union b1 b2) (make-bunch-pair b1 b2))
```

Binary Tree

A tree node is either empty or

Tree	Tree

Information is stored by 'decorating' the tree.

Builtin type:
DrRacket - empty
Anything else - null

```
;; is e a member of b?
(define (bunch-member? e b)
    (cond
    [(and (number? (bunch-pair-a b)) (= e (bunch-pair-a b))) true]
    [(and (number? (bunch-pair-b b)) (= e (bunch-pair-b b))) true]
    [(and (bunch-pair? (bunch-pair-a b)) (bunch-member? e (bunch-pair-a b))) true]
    [(and (bunch-pair? (bunch-pair-b b)) (bunch-member? e (bunch-pair-b b))) true]
    [else false]))
```

Efficiency

September-24-10 2:30 PM

"More Efficient"

b is more efficient than a if for all C > 0, ∃ n_0 such that ∀ n ≥ n_0
$T_a(n) > C \times T_b(n)$

Notation for Asymptotic Efficiency

o (little-oh)

T_b is $o(T_a)$
T_b is $o(T_a(n))$
$\cancel{T_b = o(T_a)}$
$\cancel{T_b = o(T_a(n))}$

$2n + 2$ is $o(n^2 + 1)$

$o(T)$ is the set of all function that are more efficient than T

$2n + 2 \in o(n^2 + 1)$

What would be correct, but nobody uses:
λn.2n+2 ∈ o(λn.$n^2 + 1$)

f(x) is o(g(x))

If f(x) is $o(x^2)$
2f(x) is $o(x^2)$
$a \times f(x) + b$ is $o(x^2)$

Running Time or Time Efficiency

T(n) - number of steps required to evaluate a problem of size n

Examples:
$T(n) = a + bn$ - linear
$T(n) = a * 2^n + b$ - exponential
$T(n) = a \times \log_2 n$ - logarithmic
$T(n) = x_k n^k + x_{k-1} n^{k-1} + \cdots$ - polynomial time
$T(n) = b\sqrt{n} + c$

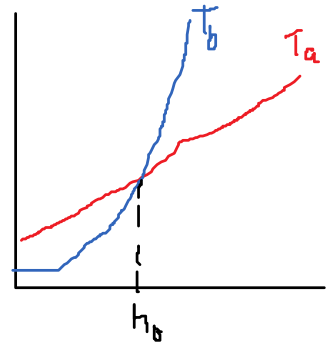
Space is another quantity that is monitored.

"More Efficient"

$T_a(n) = n^2 + 1$
$T_b(n) = 2n + 2$

Which is more efficient?
$T_a(1) = 2, T_b(1) = 4$
$T_a(2) = 5, T_b(2) = 6$
$T_a(3) = 10, T_b(3) = 8$

$n_0 = \max(4, 4C)$
$n \geq n_0 \geq 4$
$n \geq n_0 \geq 4C$
$0 > -\frac{1}{n}$ [because n≥ 4 ⇒ n > 0]
$2C > \frac{2C}{n}$
$4C \geq 2C + \frac{2C}{n} - \frac{1}{n}$
$n > 2C + \frac{2C}{n} - \frac{1}{n} [n \geq n_0 \geq 4C]$
$n^2 > 2cn + 2c - 1$
$n^2 + 1 > C(2n + 2)$
$T_a(n) > C \times T_b(n)$



Trees - Recursive Data Structures

September-28-10 11:46 AM

Info about Trees

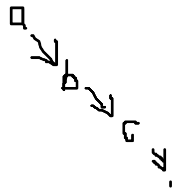
Size - number of nodes
Cormack's definition:
Height - ε is 0
· is 1
Other definition:
Height- · is 0

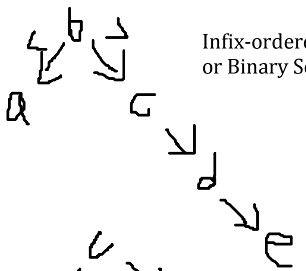
Note:

No overloading in Racket
Overloading is defining more than 1 function with the same name, determine correct one by # and type of parameters.

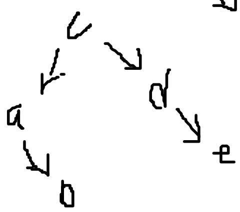
Order in Trees

a < b < c < d < e

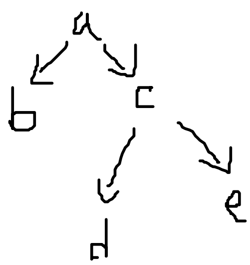




Infix-ordered tree
or Binary Search Tree



Breadth First Ordering

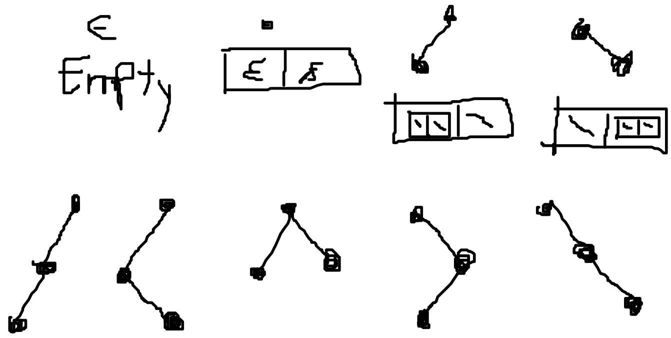


Binary Tree

Empty or

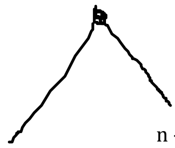
t1	t2

Where t1 and t2 are trees



Notation:
ε - empty
· - one node

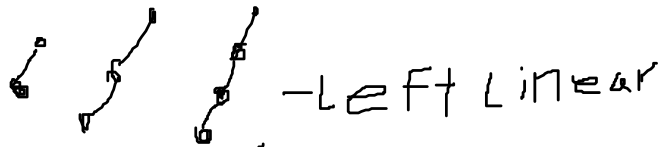
Building a tree with n ≥ 1 nodes



n - k - 1 nodes

0 ≤ k < n nodes

Linear Trees



Right Linear

Decorate (annotate/label) trees

```
(define-struct dot (L R label)

(define (smart-dot l)
    (make-dot empty empty l))
```

Sets: (Multisets have repeated elements- sets do not)

Set	Tree
{}, ∅	ε
{a}	· a, a
{a, b}	a -> b, b<- a, b -> a, a <- b
{a, b, c}	a-> b -> c,... etc. Many trees

Can reduce the number of tree combinations by using a linear tree (right linear tree)
Down to n! ways of representing a set.
Suppose we have a total ordering ⇒ Then we can represent as an ordered linear tree - 1 possibility

Lists in Beginning Student

September-28-10 12:39 PM

List Construction

Empty or
(cons x list)

(list 10 20 30)
produces 10 -> 20 -> 30

In racket
(first x) is (car x)
(rest x) is (cdr x)

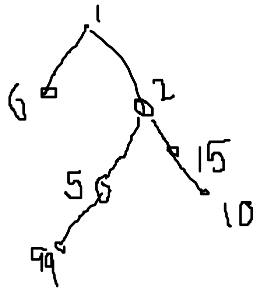
(car (cdr (cdr q)))
⇒ (caddr q)

Built in type called pair
-construct called cons (not make-pair)
(cons 42 empty)

Ex. (define q (cons 10 (cons 20 (cons 30))))
Gives 10 -> 20 -> 30
(first 1) is 10
(rest q) is · 20 -> · 30
(rest (rest q)) is · 30
(first (rest (rest q))) is 30

Trees - Searching

September-30-10 11:31 AM



Binary Tree

Does this tree contain a node decorated with n?

N = 99 yes
N = 77 no

If so, give me the node (aka, give me the subtree rooted at the node.)

Abstract

The tree represents the set S= {1, 2, 5, 6, 10, 15, 99}
Is n ∈ S

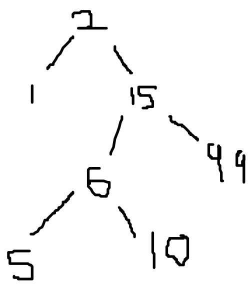
If S is a decorated binary tree
(define-struct node (left right key))

```
(define (tree-member t n)
    (cond
        [(empty? t) false]
        [(= n (node-key t)) true]
        [(tree-member (node-left t) n) true]
        [(tree-member (node-right t) n) true]
        [else false]))
```

Or

```
(define (tree-member t n)
    (and (not (empty? t))
        (or (= n (node-key t))
            (tree-member (node-left t) n)
            (tree-member (node-right t) n))))
```

This search takes linear time in the size of the tree



Binary Search Tree

Previous (tree-member t n) works but can be made more efficient

```
(define (tree-member t n)
    (cond
        [(empty? t) false]
        [(= n (node-key t)) true]
        [(< n (node-key t)) (tree-member (node-left t) n)]
        [else (tree-member (node-right t) n)]
```

This search takes linear time in the height of the tree.
From linear to logarithmic in n

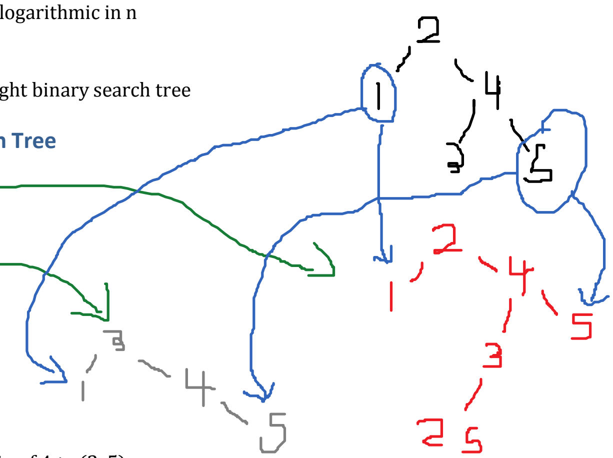
AVL Tree

A near-min height binary search tree

Binary Search Tree

Insert: 2.5

Delete 2



Delete min of 4-> (3, 5)

Building Trees

empty
(make-node .... (make-node (.....)) ...)

Pseudo-Function
(random n) ⇒ some int in [0, n)

Generator Function

```
(define (gen-min-tree n m)
 (if (= n 0) empty
    (make-node (gen-min-tree (quotient n 2) m)
          (gen-min-tree (- n 1 (quotient n 2)) m)
          (random m))))
```

Reuser Functions
When combining tree, link to the tree as a whole without re-building the entire tree.
Only have to build "spine" of tree, all of the branches off of it can be reused.

Midterm Topics

October-01-10 2:30 PM

Test

Scheme expressions
Substitution
Variables
Functions
Counting Evaluation Steps
Structures and Types
Trees (Including linear trees)
Counting Trees
Building, Searching, Combining, and Destroying Trees

Not Test

Models of computation
Lambda calculus
Asymptotic Analysis
Lists

Types of Trees

October-01-10 2:39 PM

Balanced Tree

Size-Balanced

A tree is balanced if there are the same number of nodes on each side of a node (± 1) and all sub-trees are balanced.

Path-Balanced

The shortest path from root to node is $\geq$ the longest path from root to leaf node -1

A path/size balanced tree always produces a min height tree.

Height Balanced

The height of 2 sub-trees differs by at most 1.

Perfect Tree

$n = 2^k - 1$ nodes and min height

Complete Tree

A path-balanced tree such that the left sub-tree must be at least as big as the right sub-tree.

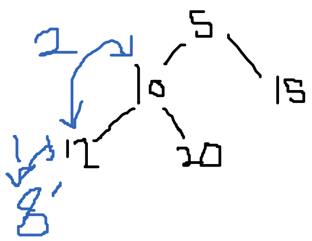
AVL Tree

A height-balanced BST

Heap (Min-Heap)

Complete tree + height ordered (not BST)
Every node is greater than its parents.

Deleting from a heap:
Remove top, promote the lowest node from its two children and repeat.
Finally, move leafs laterally until the heap is complete

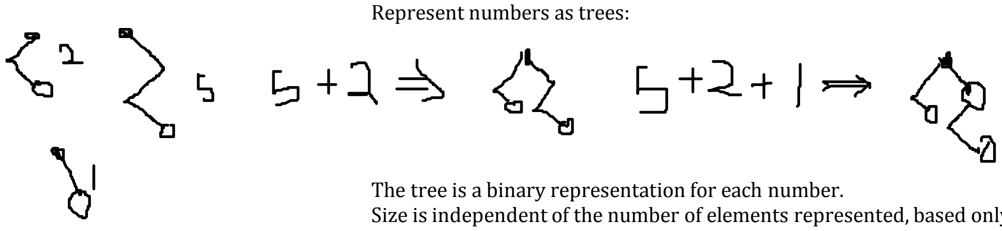


Inserting into a heap:
Insert into bottom and 'bubble' up

Heap allows Priority Queue

Digital Search Tree (Trie)

Represent numbers as trees:



The tree is a binary representation for each number.
Size is independent of the number of elements represented, based only on the height of the largest binary number stored.

Abstract Data Types

October-05-10 11:28 AM

Abstract Data Type

A data type where the actual mechanism for storing the data is unimportant.

Big-Oh Notation

$g \in O(f)$
g is asymptotically no larger than f

$f \in \Omega(g)$ means g ∈ O(f)
f is asymptotically no smaller than g

$f \in \Theta(g)$ means $f \in \Omega$(g) and $f \in \Omega(g)$

Definition of O(f)

f(n) ∈ O(g(n)) if
$\exists$ c > 0 $\exists$ $n_0 > 0$ s.t. (such that)
$\forall$ $n \geq n_0$ f(n) ≤ c × g(n)

AVL Tree

An AVL tree has the invariant:

- BST
- Height Balance

Modules

Imported via:
(require "avl-cs145.ss")
When building a module must use full language

(provide insert-avl delete-avl node-left ... etc)

The AVL Tree is opaque
Cannot print the tree
Cannot make-node (forgery)
Cannot change avl-tree (tampering)

Modules help reduce the complexity of your code by separating it into manageable chunks. For n lines want approximately $\sqrt{n}$ modules with $\sqrt{n}$

Testing

- Use beginning student with list abbreviations
- Use Version 1.1 of av1-cs145.ss
- Test!

Assignment - Implement a "set" abstract data type
~10 things to implement (functions + helpers)

Unit test:

Test each small section of code individually
Then start to test groups of code.

(check-expect (+ 1 2) (- 4 1)) <- Magic pseudo function

Order Notation for Efficiency

Course Webpage has a good document on efficiency

O(f) where f is a function.

- The set of all functions that are asymptotically no larger than f

Often instead of O(f) we say O(f(x)) or O(f(n)) or O(n^2) etc.

$g \in O(f)$ g is asymptotically no larger than f
$f \in \Omega(g)$ means g ∈ O(f) or f is asymptotically no smaller than g
$f \in \Theta(g)$ means $f \in \Omega$(g) and $f \in \Omega(g)$

Properties

If f(x) is O(g(x)) and h(x) is O(g(x))
Then
f(x)+h(x) is O(g(x))
c× f(x) is O(g(x))

Definition

f(n) is O(g(n)) means
There exists c > 0
There exists $n_0 > 0$
Such that for all $n \geq n_0$ $f(n) \leq c \times g(n)$

Example:

Prove $3n^2 + 6n$ is $O(n^2)$

$6 \leq n$
$6 \leq (4-3)n$
$6n \leq (4-3)n^2$
$3n^2 + 6n \leq 4n^2$
So
$n_0 = 6$
$c = 4$

Disproving:
$\exists$foo $\overline{\text{bar (For all variable boolean)}}$
$\Rightarrow \forall$foo $\overline{bar}$

Significant Orders

O(1) - Constant Time
O(n) - Linear Time

$O(n^2)$ - Quadratic Time
$O(n^p)$ - Polynomial Time
$O(\sqrt{n})$
$O(\log n)$ - log

Functional Abstraction

October-07-10 11:30 AM

Function expressions:

```
(define (double x) (+ x x))
```
Is equivalent to
```
(define double (lambda (x) (+ x x)))
```

Pass functions as arguments

```
(define (double x) (+ x x))
(define (square x) (* x x))
```
Can be expressed as one function with

```
(define (twice f x) (f x x))

;; takes two functions and a value, returns a value
(define (both f g x) (f (g x)))
```

Polymorphic type: α, β, γ, etc.

```
;; gcompose: (β -> γ) X (α -> β) -> (α -> γ)
;;takes two functions and returns a function
(define (gcompose f g) (lambda (x) (f (g x))))

(define (stutter f) (lambda (x) (f (f x))))
```

Lambda Calculus

```
(define mytrue (lambda x y)  x))
(define myfalse (lambda x y) y))
(define (myif b t f) (b t f))
```

Implementing a struct using lambda

```
(define mycons (lambda (car cdr) (lambda (b) (myif b car cdr))))
(define (myfirst mylist) (mylist mytrue))
(define (myrest mylist) (mylist myfalse))
```

```
(define (twice f x) (f x x))
(define (four f x) (twice f (twice f x)))

(define (gcompose f g) (lambda (x) (f (g x))))

(define (stutter f) (lambda (x) (f (f x))))

(define mytrue (lambda (x y) x))
(define myfalse (lambda (x y) y))
(define (myif b t f) (b t f))

(define mycons (lambda (car cdr) (lambda (b) (myif b car cdr))))
(define (myfirst mylist) (mylist mytrue))
(define (myrest mylist) (mylist myfalse))

;; with normal list
(define (doublel1 l) (if (empty? l)
            empty
            (cons (+ (first l) (first l))
               (doublel1 (rest l)))))

;; or, lambda expression to avoid calling twice
(define (doublel2 l) (if (empty? l)
            empty
            (cons ((lambda (x) (+ x x)) (car l))
               (doublel2 (rest l)))))

;; alternatively, can use let
(define (doublel3 l) (if (empty? l)
            empty
            (cons (let ((x (car l))) (+ x x))
               (doublel3 (rest l)))))

(define (myempty? l)
 (if (empty? l)
   mytrue
   myfalse))

;; map lambda list
(define (mymap f l) (myif (myempty? l)
             empty
             (mycons (f (myfirst l))
                 (mymap f (myrest l)))))

(define a (mycons 2 (mycons 3 empty)))
(mymap add1 a)
```

Lists and Graphs

October-14-10 11:34 AM

Can be represented by a list of pairs. For a directed graph the pair (A, B) means A → B

To remove duplicates

```
(define (dedupe l)
        (foldr (lambda (x y) (if (or (empty?  y) (not (symbol=? x (car y))))
                                        (cons x y)
                                        y))
                empty
                (sort l symbol<?)

(define (symbol<? a b)
        (string-cis? (symbol->string a)(symbol->string b)))
```

Syntax of foldl

```
(foldl [function] [identity] [list])
Eg. (foldl + 0 (list 1 2 3))
Eg. (foldl max -inf.0 (list 1 2 3))
```

Sorting

(sort ...)

Ex.

```
(require "avl-cs145.ss")
(define (mysortd l)
        (listavl (foldr (lambda (e s) (insertavl s e))
                    empty
```

Insertion Sort

```
(define (insert e l)
        (cond
                [(empty? l) (cons e l)]
                [(> e (car l)) (cons (car l) (insert e (cdr l)))]
                [true (cons e l)]))

(define (myinsertsort l)
        (foldr insert empty l))
```

Invariant for Insertion Sort:

1. Elements in current + result = elements in input
2. Result is ordered

Ex

Current	Result
(2 6 3 9)	()
(6 3 9)	(2)
(3 9)	(2 6)
(9)	(2 3 6)
()	(2 3 6 9)

$O(n^2)$

Selection Sort

Invariant - the same

Take the largest element in current and add it to the result

Current	Result
(2 6 3 9)	()
(2 6 3)	(9)
(2 3)	(6 9)
(2)	(3 6 9)
()	(2 3 6 9)

$O(n^2)$

Merge Sort

Invariant:

1. Union of the sub-lists is equal to the input list
2. Elements in each sub-list are sorted

Steps	Result

0	((2) (6) (3) (9))
1	((2 6) (3 9))
2	((2 3 6 9))

Log n steps, O(n) time per step
Sorting takes O(n log n)

Streams

(define ff (+ 1 2 3 ... 10000)) - is evaluated immediately
(define (f) (+ 1 2 3 ... 10000)) - is evaluated when (f) is called

(g lambda () (+ 1 2 3))) <- a 'promise' or passing something to a function that is to be evaluated later

Programs that do stuff in the real world

October-19-10 11:39 AM

Using gedit

RunC gedit
 Runtime plugin (ctrl-R)

"This software runs on Linux and it runs on Mac. It does not run on Windows. You say 'wait a minute, I'm addicted to Windows.' Get over it."

Evil

Pseudo-functions
 Ex. (random 10) ; magic
 -procedural, not functional programming

(read) - implements a stream: input
 Needs input to be of the form of a scheme expression

Test for end of file: (eof-object? x)

(display) - outputs the parameters and returns #<void>

Could output multiple values and hide output through
(define (discard x) (void))

(discard (list (display Hello) (newline)))

But this is almost the same as
(begin (display Hello) (newline)) - returns the result of the last evaluation

More Evil

Assignment
(define x 3)
(set! x 4) <- changing the value of the variable x

Pure Evil

Mutation

(define-struct foo(a b) #: mutable #:transparent)

(define f (make-foo 1 2))
(define q (list f f))
(foo-a f) ;1
(foo-b f); 2

f ; (foo 1 2)
q; (list (foo 1 2) (foo 1 2)

(set-foo-b! f 42)

f ;(foo 1 42)
q; (list (foo 1 42) (foo 1 42))

(set-foo-a! f f)

q; (list #0=(foo #0# 42) #0#)

Destructive Data Structures

Allows you to modify trees etc. without reconstructing the entire tree. Instead, changers the pointers of notes, or removes nodes.

Data Structures

October-21-10 12:36 PM

Copy

Exercise (define copy t)
    (generate
    ...
    ))

```
(define (tcopy t)
(cond
    [(empy? t) empty]
    [else
        (make-node (node->data t)
            (tcopy (node-left t))
            (tcopy (node-right t)))])
```

Without recursion, keep a list of all the nodes stepped through from the top to the bottom
This would be a stack

Models of Computation

Finite state machine - finite stream

Finite state machine + 1 stack ⇒ push down automaton

Finite state machine + 2 stacks ⇒ Turing machine

RAM can be used in place of stacks

Queue

Cons - front of list
Snoc - end of list

ADT Queue, will have mutable head pointer and mutable tail pointers

C - RunC

October-26-10 11:30 AM

Concrete approximation of the RAM model

C is not high level, just syntax for a computer

C

```
#include<stdio.h>

int main()
{
    int c;
    c = getchar();
    printf("The answer is %d.\n", c);
}
```

Character encoding in ASCII - 7bit
Type man ascii to see the table (in terminal)

Other character encoding:
BCD - 6bit
EBCDIC - 8bit
Ctrl-D is -1, or EOF

Comments

/* This is a comment
That spans multiple lines*/

// This is a single line comment

Functions

```
int mydouble(int x)
{
    return x * x;
}
```

Modules

Specification and Definition (header)

Specification

mydouble.c
int mydouble (int x)
{
    return x * x;
}

Header

mydouble.h
int mydouble(int x);

Including the Module

#include "mydouble.h"

RunC

Ctrl-R to run
Ctrl-V when you're done

Suppose file is called answer.c
Then when run in a folder with answer.in.something it will use this file as input and output to answer.out.something

If there is an answer.expect.something
It will compare the two in answer.check.something

Computer Architecture

October-26-10 12:14 PM

Byte

8 bits - each bit is 2 states

(Real) Machine Organization [Architecture]

RAM

Stack of bytes with addresses

Operations on RAM:
Fetch: Returns a byte stored at an address
Store: Stores a byte in an address.

Typically there are 2^b addresses
$2^{16} = 65536$ - old, 16 bit
$2^{32} \approx 4\ billion$, 32 bit
$2^{64} \approx \infty$, 64 bits

ram.ss

ram-init b ;; creates a new RAM with b-bit addreses
ram-fetch ram addr ;; fetches byte at addr from ram
ram-store ram addr byte ;; new ram with byte stored at addr

ram implementation is fairly similar to generate, but initial state is RAM and no result
use standard input/output

Byte Representation

Unsigned Integer 0 - 255
Signed - Two's Compliment -128 to 127

$unsigned(b) \equiv int(b)\ (mod\ 256)$

$129 \equiv -127\ (mod\ 256)$

Little Endian
1 0 1 0 1 0 1 0 $(1 + 4 + 16 + 64)$

Big Endian
1 0 1 0 1 0 1 0 $(128 + 32 + 8 + 2)$

When declaring variables, the names represent memory addresses. - Environment

In C a statement like
fib1 = tmp + fib1 + 1 is broken into:

tmp1 = tmp + fib1
tmp2 = tmp1 + 1 (where 1 is saved as a literal in ram)
fib1 = tmp2

When using a 4-byte word to represent an integer, you essentially have a base 256 number.
Can store values from $0\ to\ 2^{32} - 1$

C Program

October-28-10 12:15 PM

Libraries

#include <stdio.h>
#include <stdlib.h>

malloc is included in stdlib

A C program is a set of compilation units.

A procedure in C:

```
int foo (int x){
	int a = x + 1
	return x + a;
}
```

Frame or Activation Record - stores the variables used in a procedure
(Also known as a stack frame)

Parameters are copied into the frame, computation is performed on the frame, the return value is copied back, and then the frame is destroyed.

When using recursion or calling other functions, a frame is built for each new function call. This operates as a stack.

C Memory Organization

Given a RAM
C divides the RAM into 4 sections:

- Literal Pool
- Static Variables
- Stack
- Heap

Data Storage

Literal Pool

Is where literals (e.g. numbers and explicit strings) are stored for computation

Static Variables

Location of variables that are not declared inside any procedure. The variable does not move, it is persistent.

int a; when outside of a function/procedure will be saved here

Stack

Location for frame allocation. In the stack, frames are allocated sequentially.

int a; when inside a function/procedure will be saved here

Heap

General purpose data structures.
Accessed using malloc
ex: a = malloc(10)
    -finds 10 bytes and returns the address
Memory is returned by free(a)

Memory must be freed!

int * a = malloc(sizeof(int)); will be saved in the heap

Program

Translation of C code.

Pointers

int * x; x is a pointer to an integer
x = malloc(sizeof(int));

*x = 42; (sets 42 to the memory location of x)
x = 42; (sets x to 42 - in other words *x will point to the memory location of 42)

*x = *x +1, will add 1 to the memory location at x

x + 1 will in fact increment x by 1 × (sizeof (*x))

int y;
& y is the pointer to y

Creating an array

int *w = malloc(1000 * sizeof (int))
*w is the first integer in the array
*(w+1) is the second integer in the array etc.

Syntactic sugar: x[i] means *(x+i) , which is also equal to i[x]

Dangling reference - a pointer to deallocated memory.

Tutorial

October-29-10 2:44 PM

Parts of a C program

Directive

#include <stdio.h>
Compiler pastes stdio.h into your program

Declarations

int x;
Static storage in RAM

static int x;
If don't use static, if you compile the file with static int x; with another file, when you declare int x; in the new file they will be treated as separate variables - if don't use static, they will be the same
Similar in idea to (do not provide x) - if such a function existed

Good practice to use static

Compilation

Compilation takes a bunch of .c files and runs them through a compiler (GCC) and outputs and executable image. (.exe on windows) - Machine Code or Binary Code
The executable image is run through a loader and loaded into RAM. Then the CPU executes it.

Linking

Combining multiple executable image files.

Bootstrapping (Booting)

Aka IPL - Initial Program Load

The loader is run by a further smaller loader, that is loaded by a further smaller one, etc. Until get to a hardcoded loader (in ROM often)

What happens when you hit Ctrl-R

Comes across #include statements in the code.

Uses recursion to include all reachable #include files in all the files accessible from the initial file.

Image files created from all the .c and .h files reachable through include.
Image files is saved as a, but can be overridden with -o [name]

gcc bar.c foo.c bif.c baf.c -o x
Will create x, a executable image file with bar.c, foo.c, bif.c, and baf.c
Run with ./x

Other gcc flags: --std=c99

With RunC , after loaded into RAM, valgrind loads the file
Valgrind will prevent your program from using uninitialized RAM

Valgrind runs much slower than bare machine code, the price of security.

Computer History

Stored program computer

John van Neumann
The concept of saving the program in RAM

Jaquard, invented the programmable loom using punched holes
Player Pianos operate on a similar concept

Charles Babbage, Ada Lovelace

- Creator of the Difference Engine and the Analytical Engine
- Ada Lovelace programmed for these engines.

1920-1930
Telephones / Switches
Discovered you could construct mechanical relays. Used for amplification - low amount of power to an electromagnet could close a switch allowing more power through the relay.

Latch, relay which stays in the last position it was set to. A bit

Delay line: A relay connected to a loop, sends current through the loop, when the current returns it triggers the repeater and sends another pulse of current through the loop. Also known as a shift register.

A modern possible alternative for solid-state RAM is to use a light-based delay line.

Claude Shannon
At MIT. For master's degree he discovered a way to simplify switches with boolean algebra.
Went to work for AT&T, and WWII happened. He invented a science known as information theory.

RAM storage with CRT. When the electron strikes the phosphorous screen, it will dislodge an electron which is collected. If, however, light is shining on the phosphorous, the electron will not be emitted.

Display all the dots with the CRT - light or dark. Then repeat to check to where the screen is lit.

CORE RAM

A grid of wires with a magnetic core at each intersection. Can pass current through the wires to magnetize the magnetic core in one of two directions.

Modern Ram

Static Ram

A huge array of latches
Expensive but fast

Dynamic

Similar to CORE RAM, but uses capacitors instead of magnetics cores.
Dynamic because the memory is not saved long, instead the memory is constantly read and re-written.

More C

November-02-10 11:32 AM

Output Format Specifiers

%d - integer
%p - pointer
%x, %X - hexadecimal

Variables

.c and .h

In .c have
directives and declarations

int x; 32 bit word in static area of RAM [2's compliment number]; sizeof(int) = 4
regardless of whether the computer is 32 or 64 bits
int *p; 32 bit word in static area of RAM [address] sizeof(int *) = 4
on a 64 bit machine this would be a 64 bit word; sizeof(int *) = 8

Static area is initialized to zero.

x	0
p	0 or NULL

NULL is a macro in stdlib.h (#define NULL 0)

*p = malloc(sizeof(int));
malloc returns uninitialized memory in the heap

Procedures

```
int foo(int a, int b, int *c)
{
        [local declarations and statements]
        int q;          -4 bytes on the stack (in a frame)
        q = 42;
        printf("%d\n",q);
        static int z;   z is stored in static memory, however it is still local, can only be accessed by foo
                          z is initialized to 0. So doing something like this lets you keep a counter for how
                          many times a function is called or keep a value between multiple calls of a function.

        int *w;
        w = &z;
        printf("%p\n", w);
```

Statements (Imperative)

Assignment: x = e;
Evaluates the expression e and stores it in the memory location of x
e can be a constant, a variable name, a function call, expressions combined by operators

Boolean: false = 0, true = not zero
$e_1 == e_2$ equality check (NOT $e_1 = e_2$)

Logical:
&& - AND operator. Uses short-circuit evaluation
$e_1 \&\& e_2 \&\& e_3$
|| - OR operator. Also uses short-circuit evaluation

Bitwise:
& - Bitwise AND
| - Bitwise OR
^ - Bitwise XOR
<< - left shift, $e_1 \ll e_2$ shifts e_1 e_2 bits to the left
>> - right shift, $e_1 \gg e_2$ shifts e_1 e_2 bits to the right.
When shifting to the right, the signed bit is replicated on the left as new bits
The type of shifting depends on whether the number is signed or unsigned.

Expressions

*e pointer dereference
&e address of
+e
-e
!e
$e_1[e_2]$ $e_1[e_2] \equiv *(e_1 + e_2)$

Control Structures

If

if (e) [statement]
else if (e_2) [statement]
else if (e_3) [statement]
else [statement]

Each successive if is a new if/else statement

However, suppose you wanted one of the [statements] to be an if statement. Where does the next else match up to?

if (e) if(foo) [statement] else; // else; does nothing but clarifies the usage of else
else if (e_2) {if (bar) [statement]} // enclosing in braces also work.
else if (e_3) [statement]
else [statement]

If/Else if structure:

```
if (e ) {

} else if(){

} else if (){

}else {

}
```

While

while (e) [statement]
works like a generate statement over all of RAM

Documentation: Describe an invariant, understand what is happening to the variables.

For

for (e_1; e_2; e_3) [statement]

Equivalent to
e_1
while (e_2){
    [statement]
    e_3
}

Common for idioms:

```
for (i=0; i < 10; i ++){
}
```

Do While

do [statement]
while (e)

Structures

```
struct foo{
    int a;
    int *p;
}; //MUST HAVE A SEMICOLON AT THE END
```

global declaration:
struct foo f;
 - creates a structure in the static memory location.

f.a = 42;
f.a = f.a + 1; or f.a += 1

```
int bar(){
    struct foo y;
    y.a ...
}
```

If want to be lazy:
typedef struct foo myfoo;
myfoo ; ≡ struct foo x;

myfoo *z = malloc(sizeof(myfoo)); or malloc(sizeof(struct foo));

(*z).a = 42;
(*z).p = malloc(sizeof(int))
*((*z).p)=45;

struct foo g = *z;

Instead of (*z).p, can say x->p

saying
struct foo{

```
    int a;
    int *p;
} a, b, c;
```
will declare a, b, and c to be struct foo in whatever location the code is written (static or frame)

Structures in C

November-04-10 11:31 AM

Linked List

```
struct node{
	int data;
	struct node *rest;
};
```

It is possible to declare
struct node *p; without the declaration of struct node in the given file.

Tree

```
struct node{
	int data;
	struct node *left, *right;
};

struct node *a = malloc(sizeof(struct node));
struct node *b = malloc(sizeof(struct node));
struct node *c = malloc(sizeof(struct node));

a->data = 3;
a->left = b;
a->right = c;
b->data = 2;
c->data=4
b->left = b->right = c->left = c->right = NULL;
```

could made this easier with:

```
struct node * make_node (int d, struct node *l, struct node *r){
	struct node *temp = malloc(sizeof(struct node)); if returning memory use malloc otherwise the
	frame will disappear and the memory will go with it. temp is on the frame, malloc is on the heap.
	temp->data=d;
	temp->left=l;
	temp->right=r;
	return temp;
}

struct node * t = make_node(3, make_node (2, NULL, NULL), make_node(4, NULL, NULL));

void delete_tree(struct node *t)
{
	if (!t) // aka t==NULL
		return;
	delete_tree(t->left);
	delete_tree(t->right);
	free(t);
}

delete_tree(t); // must delete the tree before the program ends

struct node *x = make_node(42, NULL, NULL);
struct node *y = make_node (65, x, x); called sharing or aliasing
```

What happens when you try to delete y? x will be freed twice = bad

Can avoid this problem by not sharing or fixing delete

Fixing Delete:

Find all of the nodes that are reachable from t with no duplicates and delete those

Garbage Collection

Copy Collection

Break RAM into two parts, allocating new memory into one. Once that gets full copy everything into the other - of course only that is reachable will be copied. Erase everything in the in the original area, repeat.

Requires identifying the root pointers (in C the root pointers are in the static and stack memory locations. If something in the heap isn't reachable from those locations it isn't reachable at all)

In C there is no garbage collection - cannot move memory around.
In general, half of RAM will be usable before it starts to become slow to find new memory.

Object Oriented

```
struct foo { // or class
	int x, y, z;
	int double(int x) {return x + x} // Cannot say in C, can say in c++, called a method
}

struct bar extends foo{ //not real C/C++ syntax
	int w;
```

}

Can do "object oriented" in scheme

```
(define (make-automobile steel plastic)
        (define c (fabricate steel plastic))
        (define (driveto city) ...)
        (define (crush) ...)
        (define (fill gas)...)
        (list driveto crush fill))

(define mycar s p)
((first mycar) 'Toronto)
```

Polymorphism

Templates, allowing you to use various types for the same purpose

Continuation Passing

November-05-10 2:34 PM

Program in Scheme but every function must yield void

(define (double x) (+ x x) (void)) // returns void but no useful information
(define (double x) (display (+ x x))) // returns void
cannot use (double (double x))

(define (double x f) (f (+ x x)))
(double 4 display)
(double 4 (lambda (x) (double x display)))

f is a continuation. A lambda that describes everything that remains to be done in the program.
CPS - Continuation Passing Style
    (double 4 (lambda (x) (double x display)))

Steele's Masters Thesis → Rabbit

Scheme turns every function you create into one of the CPS. It is possible to access the hidden continuation function
(define (double x) (call/cc (lambda (c) (+ x x))))
(double 4)
>> 8

Call/cc takes the continuation function and replaces c with it.

You can stick c in the lambda and it'll end the computation earlier. c acts as return
(define (double x) (call/cc (lambda (c) (c 42) (+ x x))))
(double 4)
>> 42
Does not compute (+ x x)

(define (foo X C) (if (= X 6) (C 42) X))
(define (double x) (call/cc (lambda (c) (foo x c) (+ x x))))
(double 4)
>> 8
(double 6)
>> 42
(double (double 3))
>> 42

Version of this in C is setjump and longjump

(define q 34)
(define (foo X C) (set! q C) (if (= X 6) (C 42) X))
(define (double x) (call/cc (lanbda (c) (foo x c) (+ x x))))
(double 4)
>> 8
(q 33)
>> 33

Data Types

November-09-10 11:30 AM

Types in C
short, long / int, long long, char
signed, unsigned

short - 16 bits / 2 bytes / -65536 to 65535
int - 32 bits / 4 bytes
long - 32 bits on 32-bit machine
- 64 bits on 64-bit machine

long long - 64 bits / 8 bytes
char - 8 bits / 1 byte / may be signed
char, signed char, unsigned char are all different types

getchar(); return int, not char
Returns -1 or something in 0-255

Casting
char c;
long i;
(long)c + i;

Floating Point (aka inexact)

float, double

float: 32 bits: IEEE standard

1bit	8 bits	23 bits
sign	exponent -128 to 127	binary fraction

Specific values used to represent 0, -infinity, infinity, NaN

double: 64 bits:

1 bit	11 bits	52 bits
sign	exponent	binary fraction

Register

Can declare variables to be in the register
register int x;

Strings

'a' is an int
sizeof('a'); returns 4

char x = 'a';
char *s = malloc(10 * sizeof(char))
s[0] = 'h';
s[1] = 'i';
s[2] = 0;

h	i	\0							

Must null terminate the string.
Characters are ASCII stored in an integer

Unicode
About 1 million different code points, 0-127 is ASCII

naïve 1 int / code point
observation: all the "real" stuff is < 65536
So most of Unicode can be represented in one short.

UTF-8

Variable length byte encoding for Unicode
8 bits, if first bit is 0 then the remaining 7 are an ASCII character.
If the first bit is 1, then there are more bytes, each byte has a 1 in front if there is another byte used in the representation.

char *t = "hi";
this puts 'h', 'i', 0 in the literal pool

t points to the literal pool.
t[1] = 'z'; <- usually not allowed by the compiler. If it works then printing "hi" would print "hz"

Stack Allocation

char x[10];
int y[3];
Will create memory locations in the stack
x is a char* and a pointer to 10 chars
char *p = x;

x[0] = 35; or 0[x] = 35; or *(x+0) = 35; or *x = 35;

The size of the stack frame is variable. Allowed to say something like int y[n];

Works also in other memory locations:

global

char x[10];
static char x[10];

Initialize Arrays (on static/stack)

int x[3] = {10, 20, 30};
char c[3] = {'h', 'i', '/0'}; also char c[] = {'h', 'i', '/0'};
char s[3] = "hi"; <- this makes no sense, does not treat "hi" like normal
Completely different from char *z = "hi"; Creates z on stack, creates "hi" in literal pool, points z to hi.

```
struct f {
	int x;
	char y;
};

struct f p = {100, 'q'};
```

Strings

November-11-10 11:30 AM

Language

A set of strings

Have a string, is this string in the language L?

Finite state machines recognize regular languages.
Linear Grammar - have only left recursion or only right recursion.
$A \rightarrow A$ a b c d or $A \rightarrow A$ a b c d
regexp - regular grammar

If can answer with a stack, the language is called a context-free language (push down automaton)
Remove the restrictions of recursion on the right

Context-sensitive language. Phrase structured grammar
$N \rightarrow A$ hello B (replace then with A and B)
Unrestricted but size of left must be smaller than size of right

Two stacks: Turing machine, can do anything.
Unrestricted Language
Stick whatever on the left or on the right
N a B b $\rightarrow$ C b N a

Noan Chonsky 1959

Strings

```
#include <string.h> // C strings, strings in general
```

```
char *x = "abc"; // literal pool
char q[] = "abc"; // stack
char *p = malloc(100)
p[0] = 'a';
p[1] = 'b';
p[2] = 'c';
p[3] = '\0';
```

Create string with 1000 a's

```
char *s = malloc(1001);
for (int i = 0; i < 1000; i++) s[i] ='a';
s[1000] = 0;
```

Read from input

```
char *s = malloc(1001);
int i, c;
for (i = 0; EOF != (c = getchar()); i++) s[i] = c;
s[i] = 0;
free(s);
// BAD, messes up with input greater than 1000 characters.
```

```
foo *q = realloc(p, n)
// p is a pointer, n is an integer - size of bytes
// malloc(n), copies everything in p to the beginning of q, frees p
```

in the previous for loop could say

```
if (i%1000 == 999)
    s = realloc(s, i+1001)
```

However far more efficient to resize when $i = 2^k \Rightarrow 2^{k+1}$

String Library Functions

Can access help through man [library function]

```
strlen(s); // the length of the string s not counting the null terminator
        // O(n) time, where n is the length of the string
strcpy(a, b); // copies the contents of b into a. A must be big enough or it will overflow the array
        // O(n) time
strncpy(a, b, n); // copies the first n bits of b into a, copies 0's after reaching null in b
        // does not copy a null terminator if n stops it before reaches the null in b
strcat(a, b); // concatenates b onto a. a had better be big enough
    strcpy(a + strlen(a), b);
char * x= strchr(a, c); //returns a pointer to the first occurrence of c in a.
        // aka returns a suffix of a beginning with the first occurrence of c
```

//a suffix of a string is itself a string.
// returns null if not found
strchr_n(a, c);// like strchr but returns a pointer to the null terminator of a if c is not found.
strtok(s, p); // returns char* , s, p are strings
// returns a prefix of s terminated before any character in p
// if called again it starts from where it left off and returns the next bit terminated before a character in p
// Useful for braking up a set of strings separated by delimiters.
char *x strstr(a, b); //returns a pointer to the first occurrence of b in a, or NULL if not found
// O (strlen(a)*strlen(b))
memcpy(a, b, n); //copies the first n bytes of b into a
memset(s, c, n); // copes c into s n times
ex. memset(s, 'a', 1000); s[1000] = 0;

*SML (Standard Meta Language)

November-16-10 11:31 AM

ML

Stands for "Meta Language"
Designed for automatic theorem proving
    Designed for LCF theorem prover
    Robin Milner

Standardized in 1990, SML (Standard ML)
Forked in 1985: CaML - Categorical abstract Machine Language
        Basis for F#

Implementation SML/NJ

Expressions

SML compiler has a REPL like DrRacket
Semicolons are used to tell the interpreter to evaluate the expression
They are used much more infrequently in programs

1+2;
> val it = 3 : int
Note the inference of the type

Declarations

val x = 1;
> val x = 1 : int

val y = 1.2;
> val y = 1.2 : real

x+3
> val it = 4 : int

fun sqr x = x*x;
> val xqr = fn : int → int
We can add type annotations as needed
fun sqr (x : real) = x*x;
> val sqr = fn : real → real

Ascription is useful in clarifying intent and debugging type errors.
ML has a fixed set of overloaded operators.

ML will use type variables for polymorphic functions
fun id x = x;
> id = fn : 'a -> 'a

val y = id 3;
> y = 3 : int

Functions in ML have exactly one parameter ("curried")
fun sumSqrs x y = x * x + y * y;
> val sumSqrs = fn : int → int → int

Tuples

val x = (true, #"z");
> val x = (true, #"z") : bool * char

There are selector functions for tuples, but they are usually deconstructed using petterns
Tuples can be used to group multiple parameters to a function
fun add(x, y)= x + y;

- val add = fn: int * int → int

Here (x, y) is a pattern
add is the same as op +
There aren't 1-tupes but the 0-tuple () handy
It's the sole value of the type unit, the equivalent of #<void> in Scheme

Binary tupled functions can be made into infix operators.
infix add;
> infix add

3 add 4;
> val it = 7 : int

We could also have done it this way
infix add

fun (x add y) = x+y;

infixr makes the operator right-associative.

Lists

[1, 2, 3];
> val it = [1, 2, 3] : int list

1 :: 2 :: 3 :: [];
> val it = [1, 2, 3] : int list

[1] @ [2, 3];
> val it [1, 2, 3] : int list

Ampersand is append, :: is concatenate

```
fun append([], ys) = ys
| append(x::xs, ys) = x:: append(sx, yx);
> val append fn: 'a list * 'a list ⇒ 'a list
```

Append is the same function as op @

An alternative:

```
fun append(xs, yx) =
      case sx of
            [] ⇒ yx
            (x::xs) ⇒ x :: append(xs, y);
```

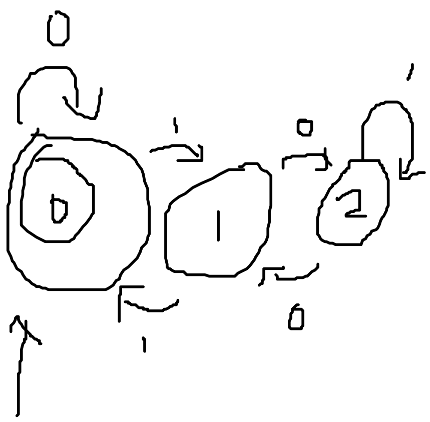
_ acts as a wildcard in pattern matching ("don't care")
(M as m::ms)
Will allow you to use either M or m, ms

Local definitions

```
fun split []  = ([], [])
|     split [a] = ([a], [])
|     split (a::b::cs) =
        let
          val (mx, nx) = split cs
        in
          (a::ms, b::ns)
        end
```

Mutual recursion

```
fun
      oddlen [] = false
|      oddlen (x::xs) = evenlen xs
and
      evenlen [] = true
|      evenlen(x::xs) = oddlen xs
```



Finite State Machine

Want to implement the finite state machine shown
In this implementation, there is a function corresponding to each state

```
val myNumbers =
      let fun
            zero [] = true
            | zero (#"0"::xs) = zero xs
            | zero (#"1"::xs) = one xs
            | zero _ = false
      and
            one [] = false
            | one(#"0"::xs) = two xs
            | one(#"1"::xs) = zero xs
            | one _ = false
      and
            two [] = false
            | two(#"0"::xs) = one xs
            | two(#"1"::xs) = two xs
            | two _ = false
      end
```

User Defined Types

type intpair = int * int
intpair is a type synonym (like typedef in C)

datatype suit = Heart | Spade | Diamond | Club

datatype value = Ace | One | … | Jack | Queen | King
type card = suit * value

datatype 'a tree = Empty
                  | Node of 'a * 'a tree * 'a tree
tree is a type constructor (as are &, -> , list)
Empty and Node are data constructors

```
fun contains Empty _ = false
|    contains (Node(x, lt, rt)) y =
         x = y
         orelse contains lt y
         orelse contains rt y;
> val contains = fn : ''a tree → ''a → bool
```

''a representa an edgetype

Exceptions
The handler must produce a value of the same type as the expression to which it is attached

```
exception DivByZero;

fun safeDif (_, 0) = raise DivByZero
|    safeDiv (x, y) = x div y;

val quot = safeDiv (3, 0) handle DivByZero => 0
```

A similar mechanism is provided in Racket

I/O
the ML standard basis library proveds many I/O functions.
We will mention only one here:
print "Testing...\n";

The string concatenation operator ^ is handy

Reference Types
Like Scheme, ML is not pure, but mutation is used sparingly
the data constructor ref provides the equivalent of boxes in Scheme

```
val x = ref 5
> val x = ref 5 : int ref

val y = !x
> val y = 5 : int

x := y + 1;
> val it = () : uint
```

Types
Type errors in ML often appear incomprehensible.
Sometimes we are prevented from writing code in a fashion that seems natural to us because of restrictions in the type system.
In order to understand how the ML compiler does type inference, we add types to the lambda calculus.
This was first done by Church in the 1930's, but we use a style closer to that of Curry's work in the same period

Untyped Lambda Calculus
An expression is either:
- A variable (x, y, z), etc)
- An abstraction λV.E where V is a variable and E is an expression
- Applications E1 E2 where E1 and E2 are expressions

Computation in the untyped lambda calculus proceeds by substitutions
(λx.y)z → y

We now add types to get the simply-typed lambda calculus
The only difference is that the variables used by abstractions are annotated with types
Computation is unchanged

Simply-Typed Lambda Calculus
A type is either:
- A base type (t1, t2) or
- T1 → T2 where G1 and T2 are types

The type constructor → is right associative

Our goal is to be albe to make type judgements
What is the type of $\lambda x{:}t1.x$? $t1 \to t1$
What about $\lambda x{:}t1.\lambda y{:}t1{\to}t2.yx$? $t1 \to ((t1 \to t2) \to t2)$

Logic

$\Gamma \vdash \alpha$ the statement "From the set of propositions Γ we can prove the proposition α
A proof is a tree built from applications of inference rules.

The Curry-Howard Correspondence

Logic corresponds to Programming Languages
The proof of α is the typing of α
For more: see Philip Wadler, "Proofs are Programs."

Theorems of Simply-Typed Lambda Calculus

A term is closed if it has no free variables
A term $T{:}\tau$ is well-typed if it can bet shown to have that type

Strong Normalization

When we try to type the Y-combinator, we run into problems
Consider typing $\lambda x.xx$
Just because we cannot type self-application doesn't mean we can't do recursion in some other fashion.
However, the strong normalization theorem suggests that the simply-typed lambda calculus is a weak model of computation - not Turing Complete

Theorem

Every reduction sequence of every well-typed term of the simply-typed lambda calculus is of finite length

To gain more power, we must extend the simply-typed lambda calculus with a construct for recursion (which breaks strong normalization).

Extensions

We can extend the simply-typed lambda calculus to bring it closer to typed functional languages such as ML.
For example, we can add Bool and Nat as base types, and constructs such as let and if.
We need inference rules for these.

Progress and preservation theorems can be proved for these extensions.
We still have not modelled type inference when annotations are absent or optional, as in ML
We also need many versions of, e.g., the identity function.

See Wikipedia on "type inference"

*Haskell

November-18-10 11:30 AM

Glasgow Haskell Compiler

Whitespace is significant, block syntax like python
Can override with {} and ;

Value definitions: var = expr
Function definitions: fname par1 par2 ... = expr

Types: Int, Real, Char, Bool
type variabes are in lower case
:: "has type" and : means "cons"
Type constructors →, [], (,)
String is a list of characters, [Char]
Lambda expressions \x → x*x

Comments --

Code Example, Permute List

```
perms1 :: [a]→[[a]]
perms1 [] = [[]]
perms1 (x:xs) = addtoAll x (perms1 xs)

addToAll x [] = []
addToAll x (p:ps) = addToOne x p ++ addtoAll x ps

addToOne x [] = x
addToOne x (y:ys) = (x:y:ys) : consOnEach y (addToOne x ys)

consOnEach y [] = []
consOnEach y (p:ps) = (y:p) : consOnEach y ps
```

Running Haskell

Interpreter ghci, similar to sml
ghc resembles gcc

ghc expects main to be defined:
main :: IO()
main - print (perms [1, 2, 3, 4])

To avoid parantheses, the function application operator $ (with lowest precedence) is used:
main = print $ perms [1, 2, 3, 4]

Any two-parameter curred function can be used as an operator:
5 'div' 2

Any operator can be used as a function: (*) 3 4

One argument can be supplied: (3:) - function conses 3 onto argument
(:[7]) - appends 7 to the arguments

perms1 = foldr addToAll [[]]

addToAll x concat . map (addToOne)
. operator composes two functions f . g ⇒ f(g(x))

addToOne x [] = [[x]]
addToOne x (y::ys) = (x:y:ys) : map (y:) (addToOne x ys)

Haskell has overloaded operators, and users can define their own
sqr x = x*x
:type sqr
> sqr :: (Num a) ⇒ a→ a

:type is a command to ghci
Num is a type class

Haskell has if-then-else with Boolean literals True and False, and logical connectives &&, ||, and not

Guards are a convenient alternative in definitions
abs x | x ≥ 0 = 0 x
| otherwise = -x

Haskell has let expressions similar to ML:
let
    sqr1 = x*x
    sqr2 = y*y
int
    sqr1 + sqr2

The use of where is more restricted, but it can scope across guards

List comprehension:
myMap f sx = [f x | x <- xs]
myFilter p sx = [x | x <- sx, p x]

cross xs ys = [(x, y) | x <- xs, y <- ys]

perms2 [] = [[]]
perms2 xs = [y:p | (y, ys) <- sels xs, p <- perms2 ys]

sels [] = []
sels (x:xs = (x, xs) : [(y, x:xs) |(y, ys) <- sels xs]

Haskell allows type synonyms using type and algebraic data types declared using data
Unlike ML, data constructors may be curried

Laziness

Scheme and ML use eager evaluation: the leftmost, innermost expression is evaluated.
This means, for example, that all arguments to a function are evaluated before the function is applied to them.

Haskell uses lazy evaluation: the leftmost, outermost expression is evaluated.
In effect, expressions are not evaluated until necessary.

As a simple example of laxiness, short-circuiting Boolean operators can be functions

myAnt :: Bool → Bool → Bool
myAnd False _ = False
myAnd _ x = x

Typing myAnd False undefined into GHCi produces False as expected
In fact, undefined is defined in the Prelude as undefined = error "Prelude.undefined"

```
ones = 1 : ones
```
could also say
```
ones = [1, 1..]
```

```
nats = 0 : map (+1) nats
```
or
```
mats = [0, 1..]
```

```
odds = filter odd nats
```
or
```
odds = [1, 3..]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

primes1 = sieve[2..]
sieve (p::ns) = p : sieve [ n | n <- ns, n 'mod' p /= 0]

primes2 = 2 : oprimes
    where
        oprimes = 3 : filter isPrime[5, 7,..]
        possDivz n = takeWhile (\p → p*p ≤ n) oprimes
        notDiv n p = n 'mod' p \= 0
        isPrime n = all (notDiv n) (possDivs n)
```

Immutable Arrays

Here's an example of crating a one-dimensional immutable array from a list of (index, value) pairs
sqrx = array (1, 100) [(I, i*i_ | i <- [1..100])]

Terminal Command Equivalents

-- interact :: (String → String) → IO ()

```
--UNIX 'cat'
main = interact id

-- UNIC 'wc -1'
showln = (++) . show
main = interact $ showln . length . lines

linemap f = interact $ unlines . f . lines

-- Unix 'head -10'
main = linemap $ take 1-

-- Unix 'grep a'
main = linemap $ filter $ elem 'a'
```

Type Classes

Type classes offer a controlled approach to overloading

The Eq class

Types in this class provide == and /=

Can create a member of eq with derive Eq

The Ord Class

Ordering

Monads

The Monad type class abstracts a common computational pattern.
One of the simplest instances is the Maybe type/
We can start by thinking of it as:
data Maybe a = Nothing | Just a

The idea is to use Nothing when an error occurs, and Just to wrap a value for continued computation.

When composing functions with maybe

```
chain :: Maybe Int → (Int → Maybe Int) → Maybe Int
chain Nothing _ = Nothing
chain (Just r) f = f  r
```

The chain function is already specified by the Monad typeclass and Maybe provides the above definition
There, chain is called >>= (pronounced "bind")

The State monad

Monads hides plumbing
The plumbing hidden by the Maybe monad is the wrapping/unwrapping of the value.
What plumbing is involved in manipulating state?
If the computation is tail-recursive, we can put state in an extra parameter (an accumulator).
It's harder with a more general computation (e.g. on trees)

Testing in Haskell

HUnit: unit testing (modelled on JUnit)

QuickCheck: Randomized testing of properties of code

Vectors/ADTs in C

November-23-10 11:29 AM

Vector

ADT for a fixed length sequence
Operations:
- Create
- Index - select element
- Set - change element

Scheme

(define v (vector 10 20 30))
(vector-ref v 2) ; 30 O(1) time
(vector-size v); 3
(vector-set! v 2 42) O(1) time
(vector-rev v 2) ; 42
(vector->list v)

Strings in scheme are vectors of characters

C

vector_user.c
```
#include "vector.h"

vector foo = vector_create(10, 20, 30);
int j = vector_ref(foo, 2);
vector_set(&foo, 2, 42);
```

vector.h
```
// 3D vectors of ints
typedef struct vector vector;
struct vector {
    int i, j, k;
};
```

Vector.h allows you to help prevent tampering. If you move the definition of struct vector to vector.c and make vector a pointer it is difficult to access the workings of the structure.

For added security can hash the vector with a signed cookie. Makes it almost impossible to create a vector with different data but the same signature.

vector.c
```
#include "vector.h"

vector vector_create(int a, int b, int c){
    vector r;
    r.i = a;
    r.j = b;
    r.k = c;
    return r;
}

int vector_ref(vector v, int i) {
    if (!i) return v.i;
    if (i == 1) return v.j;
    return v.k
}

// wrong - v is passed by value not reference
void vector_set (vector v, int i, int e){
    if (!i) v.i = e;
    else if (i==1) v.j = e;
    else v.k = e;
    return;
}

// correct version - pass the pointer of v
vector * vector_set (vector * v, int i, int e){
    if (!i) v->i = e;
    else if (i==1) v->j = e;
    else v->k = e;
    return;
}
```

vector.h
```
// 3D vector of ints
typedef struct vector * vector;
// Hands off!

vector vector_build(int, int, int);
```

vector.c
```
#include "vector.h"

struct vector {
    int x[3];
};

int vector_ref (vector v, int i) {
    return v->x[i];
}

vector vector_build (int a, int b, int c) {
    vector r = malloc (sizeof (struct vector));
    r->x[0] = a;
    r->x[1] = b;
    r->x[2] = c;
    return r;
}

void vector_delete (vector v) {
    free(v);
}
```

Make the Vector more general

vector.h
```
typedef struct vector * vector;
vector vector_build(int, int);
void vector_delete(vector);
int vector_len(vector);
```

vector.c
```
#include "vector.h"

//Approach 1
struct vector {
    int len;
```

```c
typedef struct vector * vector;
vector vector_build(int, int);
void vector_delete(vector);
int vector_len(vector);
int vector_create(int, int);
```

```c
#include "vector.h"

//Approach 1
struct vector {
    int len;
    int x[]; // woah, this has no memory malloced for it.
    // It is a pointer to zero integers
    // It will be at the end of the vector struct, although
    // maybe not guaranteed to be so.
}
vector vector_built (int len, int *e) {
    vector r = malloc(sizeof(struct vector) + len * sizeof
    (int));
    for (int i = 0; i < len; i++) {
        r->x[i] = e[i];
    r->len = len;
    return r;
}

void vector_delete(vector v) {
    free(v);
}

//Approach 2
vector vector_build(int len, int *e) {
    vector r = malloc(sizeof(struct vector));
    r->x = malloc(len * sizeof(int));
    for (int i = 0; i < len; i++)
        r->x[i] = e[i];
    r->len = len;
    return r;
}

void vector_delete(vector v) {
    free(v->x);
    free(v);
}

// For both approaches

int vector_len (vector v) {
    return v->len;
}

// bounds checking
int vector_set(vector v, int pos, int e) {
    if (pos >= 0 && pos < len) {
        v->x[pos] = e;
        return 0; /succeed
    }
    return 1; //fail
}
// or could use exit(0) - good
// exit(number ≠ 0) is bad
// ends the program, in unix can write echo $?
// to find the error code
// exit is equivalent to return from main

// alternate
#include <stdlib.h>
// has the global value errno
void vector_set (vector v, int pos, it e) {
    if (pos >= 0 && pos < len) {
        v->x[pos] = e;
        errno = 0;
        return;
    }
    errno = 1;
}
// perror prints an error message

int vector_ref (vector v, int pos) {
    if (pos >= 0 && pos < len) {
        return v->x[pos];
    }
} // how do you indicate error?
// could request a flag from the user and use that

typdef struct {
    int r;
    int status;
} int_status;

int_status vector_ref (vector v, int pos) {
    int_status stat;
    if (pos >= 0 && pos < len) {
        stat.status = 0;
        stat.r = v->x[pos];
        return stat;
```

```
    }
    stat.status = 1;
    return stat;
} // probably the best approach to deal with errors
```

But what if you want to store things in the array other than ints

Generic
(labmda (something)
struct vector {
	int len = 3;
	something x[];
}
)

Instantiation
(vector char) foo;
(vector avltree) bar;

Could build a generic as:
#define vec(t, n) \
typedef struct { \
	int len = 3 \
	t x[]; \
} n

vec(char, vecchar);
vec(int, vecint);
// defines vecchar a vector of chars
// defines vecint, a vector of ints

```
// Another Way
#include "t.h"

struct vector {
    int len;
    t *x;
}
```

t.h

```
typedef struct t_struct * t;

//operations on t
t_create;
t_create_array;
t_copy;
t_delete;
```

t.c

```
struct t_struct {
    int i;
    char c;
}; // more space than needed

union t_struct {
    int i;
    char c;
}// union stores i and c
// to the same place
```

Polymorphism

November-30-10 11:29 AM

Many Form Ism

Strong/Weak Typing

Scheme has strong typing, types are enforced by the compiler

C has weak typing, types can be treated as other types.

Many Types

Polymorphic Function - Works on many types
Polymorphic ADT - data structure and functions on many types

Scheme

Dynamic (Run-Time) Polymorphism
Dynamic Typing

C

Static typing
Loopholes for polymorphism

Polymorphism

Parametric Polymorphism (templates, generics)

(lambda (t) ...) where t is a type

Ad Hoc Polymorphism

Finite number of alternatives, each coded separately

- Unions in C
- Overloading
  - foo(int)
  - foo(char)
- Pascal → variant records
- (void *)

```
void * x;
(char *)x[i]; // will work... if you declared memory anyway
int foo (int *q) {
	bar;
}
foo(x); // will also work
```

Inclusion Polymorphism

- **Inheritance**

ADT_1 with ops f, g, h
ADT_2 with ops f g h j
$ADT_2 \subseteq ADT_1$

C polymorphism

```
int sumto (int n ) {
	int r = 0;
	for (int i  = 0; i ≤ n ; i++)
		r += 1;
	return r;
}
```

Specifically for integers

```
int sumto(int n, int *a) {
	int r = 0;
	for (int i = 0; i < n ; i++)
		r += i[a];
	return r;
}
int x[] = {10, 20, 30};
sumto(3, x);

double xx[] = {1.2, 2.3, 3.4}
sumto(3, xx); //type error
```

int sumto (int n, void *a) ... but how do you fill out the body

```
int sumto (int n, int ref(int)) {
	int r = 0;
	for (int i = 0; i < n ; i++)
		r += ref(i);
	return r;
}
```

ref is a pointer to a function in codespace

```
int x[] = {10, 20, 30};
```

```
int foo(int i ) {
	return x[i];
}
sumto (3, foo);

#include "elem.h"
elem sumto (int n, elem ref(int) {
	elem r = zero;// zero must be defined in elem.h (or you could pass zero as a parameter)
	for (int i = 0; i < n ; i++)
		r = plus(r, ref(i));
	return r;
}
```

Void pointers!

```
void * sumto (int n, void * ref(int), void * accum, void * plus (void *, void *), void copy (void ** to,
void * from)) {
	void * r;
	copy (&r, accum);
	for (int i = 0; i < n; i++)
		copy(&r, plus(r, ref(i)); // copy should be destructive
	return r
}
```

Mabel

December-02-10 11:32 AM

Manitoba Beginner's Language

1974 4th year class project
- Elementary and Intermediate examples
- Design criteria
- Language definition
- Compiler
- (Minimal) deployment experience

2006 perspective
- What's changed?

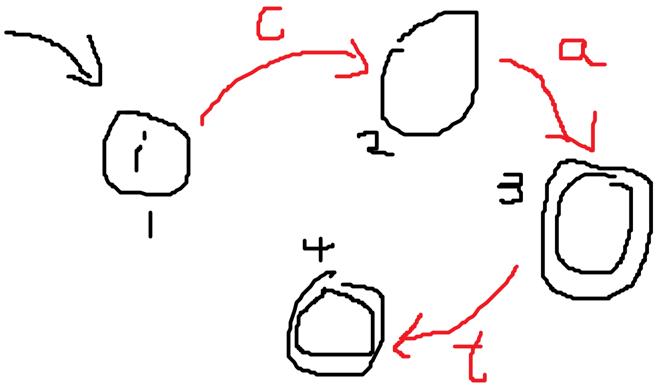
Finite State Machine

December-02-10 12:18 PM

(Finite Automaton)

S: A finite set of states
$i \in S$: Initial state
$f \subseteq S$: Final states
Σ: finite set of symbols (alphabet)
$T: S \times \Sigma \rightarrow S$

Bubble Diagram



The state machine M:

$S = \{1, 2, 3, 4\}$
$i = 1$
$\Sigma = \{c, a, t\}$
$f = \{3, 4\}$
T =

	c	a	t
1	2	-	-
2	-	3	-
3	-	-	4
4	-	-	-

Language: L(m) ={ca, cat}

If change T to

	c	a	t
1	2	-	-
2	-	3	-
3	-	-	4
4	-	2	-

L(m) is now infinite
L(m) = {ca, cat, cataa, cataat, caataataa, ... }
ca|ca(taa)*

Spam