**Definitions**

**List:** A list is either null or a pair whose tail is a list.

**Tree:** A tree of a certain data type is a list whose elements are of that data type or trees of that data type. (No tree of null or tree of pairs)

**Binary tree:** A binary tree is the empty tree, or it has an entry, a left branch or left subtree, a right branch or right subtree.

**BST:** A BST is a binary tree where all entries in the left subtree are smaller than the entry, and all entries in the right subtree are larger than the entry.

**Arrays:** An array is a data structure that stores a sequence of data elements. Arrays can be read and written in constant time. *Exception: Assigning to an array element A[i], where index I >= array\_length(A), takes Θ(i – array\_length(A)) time.*

**Streams:** A stream is either the empty list, or a pair whose tail is a nullary function that returns a stream.

**Recursive vs iterative:** Recursive means there are increasing deferred operations with respect to n, iterative means there are constant deferred operations. (Can be 0)

**Environment model**

**3 things to check for when creating a frame for a function call**

1. Extend frame from where function is created. (unless the function comes from the global env)
2. Are there function params? If so, extend frame for params. If not, don’t.
3. Are there **new** name declarations in the function body? If so, extend another frame. If not, don’t.

*If function has no params and no new name declarations in the body, no new frame should be extended.*

*An environment is a sequence of frames. Each frame contains bindings of values to names.*

**Searching & Sorting Algorithms**

**Linear search**: O(n) time

**Binary search**: O(log n) time, requires the input to be sorted, which probably takes another O(n log n) time (Merge sort).

**Selection sort:** *Θ* **(n^2)**

Lists: Find the smallest element x and remove it from the list. Sort the remaining list and put the x in front.

Arrays: Build the sorted array from left to right. For each remaining unsorted portion to the right of position i, find the smallest element and swap it into position i.

**Insertion sort:** *Θ* **(n^2)**

Lists: Sort the tail of the given list using wishful thinking. Insert the head into the right place.

Arrays: Move a pointer i from left to right. The array to the left of i is sorted already. Swap the value at i with its neighbor to the left, until the neighbor is smaller.

**Merge sort:** *Θ* **(n log n)**

Lists: Split the list in half, sort each half using wishful thinking. Merge the sorted lists together.

Arrays: Sort the halves. Merge the halves (using temporary arrays).

**Quicksort:** *Θ* **(n log n)**

Partition the list about a pivot. Append left and right of the pivot. Left portion should have numbers smaller than the pivot, vice versa for right portion.

**Common Recurrence Relations**

T(n) = T(n-1) + O(1) = O(n)

T(n) = T(n/2) + O(1) = O(log n)

T(n) = T(n-1) + O(log n) = O(n log n)

T(n) = T(n-1) + O(n) = O(n^2)

Generally, T(n) = T(n-1) + O(n^k) = O(n^k+1)

T(n) = 2 T(n/2) + O(n) = O(n log n) 🡪 Merge sort

T(n) = T(n/2) + O(n) = O(n)

T(n) = 2 T(n/2) + O(1) = O(n)

T(n) = 2 T(n-1) + O(1) = O(2^n)

The space complexity of a program refers to the additional space required by the execution of the program. You need time to create space. But taking time, does not necessarily mean you create space. Theta(g(n)) time 🡪 O(g(n)) space. Theta(g(n)) space 🡪 Omega(g(n)) time

**Continuous Passing Style**

1. Look at the recursive solution and duplicate it

2. Add a parameter, c.

3. Slightly modify the base case so that it actually calls c.

4. Initially, start with wish => c(wish).

5. Treat wish as the result of map\_cps(f, tail(xs), x => x) which contains the list we want to return except the mapped head of the list. Then, we just need to pair f(head(xs)) with wish to get the result.

function map(f, xs) {

return is\_null(xs)

? null

: pair(f(head(xs)), map(f, tail(xs)));

}

function map\_cps(f, xs, c) { // step 2

return is\_null(xs)

? c(null) // step 3

// : map\_cps(f, tail(xs), wish => c(wish)) // step 4

: map\_cps(f, tail(xs), wish => c(pair(f(head(xs)), wish))); // step 5

}

**Metacircular evaluator**

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

Parse 🡪 Evaluate 🡪 Apply

**Programming Language Processing**

Graphical user interface, application

Description automatically generated

**Interpreter (Represented as a square):** Interpreter is a program that executes another program. The interpreter’s source language is the language in which the interpreter is written. The interpreter’s target language is the languages in which the programs are written which the interpreter can execute.

JavaScript

X86-64

Chrome browser for PC, seen as interpreter for JavaScript, written in x86-64 machine code.

Table

Description automatically generatedProgram BrowserGame is written in JavaScript, which is then put through a JavaScript to x86-64 interpreter. This x86-64 program is then run by the x86-64 processor.

**Compiler:** A compiler is a program that translates from one language to another language.

Diagram

Description automatically generated

TypeScript-to-JavaScript compiler written in x86-64 machine code.

Diagram

Description automatically generated

Program RunicCurves is written in Source. It is then compiled by a Source-to-JavaScript compiler. This gives a JavaScript program. This compiler program is run on a JavaScript interpreter written in x86-64 machine code, which is run on the x86-64 processor. The program RunicCurves in JavaScript in interpreted by the JavaScript interpreter written in x86-64 machine code, which is run on the x86-64 processor.

**Some useful functions**

function mutable\_reverse(xs) {

**// version 1**

if (is\_null(xs)) {

return xs;

} else if (is\_null(tail(xs))) {

return xs;

} else {

let temp = mutable\_reverse(tail(xs));

set\_tail(tail(xs), xs);

set\_tail(xs, null);

return temp;

}

**// version 2**

function helper(prev, xs) {

if (is\_null(xs)) {

return prev;

} else {

let rest = tail(xs);

set\_tail(xs, prev);

return helper(xs, rest);

}

}

return helper(null, xs);

}

**// merge two ordered lists using recursion without creating new pairs**

function mergeB(xs, ys) {

if (is\_null(xs) && is\_null(ys)) {

return null;

} else if (is\_null(xs)) {

set\_tail(ys, mergeB(xs, tail(ys))); return ys;

} else if (is\_null(ys)) {

set\_tail(xs, mergeB(tail(xs), ys)); return xs;

} else if (head(xs) <= head(ys)) { set\_tail(xs, mergeB(tail(xs), ys));

return xs;

} else {

set\_tail(ys, mergeB(xs, tail(ys))); return ys;

}

}

**// accumulate\_tree**

function accumulate\_tree(f, op, initial, tree) {

return accumulate((x, y) => is\_list(x)

? op(accumulate\_tree(f, op, null, x), y)

: op(f(x), y),

null, tree);

}

**// flatten tree to list**

function flatten\_tree(T) {

return accumulate((x, y) => is\_list(x)

? append(flatten\_tree(x), y)

: append(list(x), y),

null,

T);

}

OR

function flatten\_tree(t) {

return accumulate\_tree(x => list(x), append, null, tree);

}

**// reverse tree**

function transform\_tree(t) {

return reverse(map(x => is\_list(x)

? transform\_tree(x)

: x,

t));

}

**// d\_append**

function d\_append(xs, ys) {

if (is\_null(xs)) {

return null;

} else {

set\_tail(xs, d\_append(tail(xs), ys));

return xs;

}

}

**// d\_map**

function d\_map(f, xs) {

if (!is\_null(xs)) {

set\_head(xs, f(head(xs)));

d\_map(f, tail(xs));

}

}

**// d\_filter (note the first element is never filtered)**

function d\_filter(pred, xs) {

if (is\_null(xs)) {

return null;

} else if (pred(head(xs))) {

set\_tail(xs, d\_filter(pred, tail(xs)));

return xs;

} else {

return d\_filter(pred, tail(xs));

}

}

**// count looped pairs**

function correct\_count\_pairs(x) {

let pairs = null;

function check(y) {

if (!is\_pair(y)) {

return undefined;

} else if (!is\_null(member(y, pairs)) {

return undefined;

} else {

pairs = pair(y, pairs);

check(head(x));

check(head(y));

}

check(x);

return length(pairs);

}

**// rotate\_matrix (transpose and then swap items)**

function rotate\_matrix(M) {

const n = array\_length(M); // M is assumed n x n.

function swap(r1, c1, r2, c2) {

const temp = M[r1][c1];

M[r1][c1] = M[r2][c2];

M[r2][c2] = temp;

}

// transpose matrix first

for (let i = 0; i < n; i = i + 1) {

for (let j = i + 1; j < n; j = j + 1) {

swap(i, j, j, i);

}

}

// swap the items in the same row

for (let i = 0; i < n; i = i + 1) {

for (let j = 0; j < math\_floor(n / 2); j = j + 1) {

swap(i, j, i, n - j - 1);

}

}

}

**// hold\_stream(list(2,4,6)) returns a stream 2,4,6,6,6,6…**

function hold\_stream(xs) {

return is\_null(tail(xs))

? pair(head(xs), () => hold\_stream(xs))

: pair(head(xs), () => hold\_stream(tail(xs)));

}

**//search\_stream**

function search\_stream(xs, pos, x) {

return is\_null(xs) || pos < 0

? false

: head(xs) === x

? true

: search\_stream(stream\_tail(xs), pos – 1, x);

}

**// memoized streams (may not actually memoize)**

function memo(fun) {

let already\_run = false;

let result = undefined;

return () => {

if (!already\_run) {

result = fun();

already\_run = true;

return result;

} else {

return result;

}

};

}

**// fast\_and\_furious (iterative, using cps)**

function fast\_and\_furious\_expt(b, n) {

function ffe(bb, nn, cont) {

return nn === 0

? cont(1)

: is\_even(nn)

? ffe(bb, nn / 2, x => cont(square(x)))

: ffe(bb, nn – 1, x => cont(b \* x)); } return ffe(b, n, x => x);