Done by Marcus Pang in 2021

Definitions

1. computational process: set of activities in a computer, designed to achieve a desired result.
2. primitive expressions: numerals, booleans, pre-declared functions
3. program: expressions with semicolons
4. pair: tuple of size 2 (array of size 2)
5. list: null or pair whose tail is a list (nested arrays)
6. tree of type T: null or pair whose head is of T and whose tail is a list of type T
7. binary tree: null or entry, left binary tree branch, and right binary tree branch
8. stream of type T: null or pair whose head is of type T and whose tail is a nullary function that returns a stream of type T
9. array: stores a sequence of data elements

Order of Growth

Consider a growth function f representing some program, then

1. $f(n)$ is big-theta of $g(n)$
   $$\iff f(n) \in \Theta(g(n))$$
   $$\iff f(n) \in \mathcal{O}(g(n)) \land f(n) \in \Omega(g(n))$$
   $$\iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ where } 0 < c < \infty$$
2. $f(n)$ is big-O of $g(n)$
   $\iff$ for sufficiently large n,
   $\exists c \in \mathbb{R}_{\geq 0}(f(n) \leq c \cdot g(n))$
3. $f(n)$ is big-Ω of $g(n)$
   $\iff$ for sufficiently large n,
   $\exists c \in \mathbb{R}_{\geq 0}(f(n) \geq c \cdot g(n))$

Substitution Model

Applicative-order reduction ("eager evaluation"):

1. primitive expressions: take the value
2. operator combinations: evaluate operands, apply operator
3. constant declaration: evaluate the value expression and replace the name by value
4. function application: evaluate component expressions
   (a) primitive function $\implies$ apply the primitive function
   (b) compound function $\implies$ substitute argument values for parameters in body of declaration

Normal order reduction ("lazy evaluation"): arguments are not evaluated, they are passed to functions as expressions, and the expressions are only evaluated when needed
Model breaks down when assignment statements are present as the model considers a constant/variable as just a name for the value and does not expect it to change.

Environment Model

Definitions:

1. environment: sequence of frames
2. frame: contains bindings of values to names, points to enclosing environment (next one in sequence unless it is the global frame)
3. extending environment: adding a new frame in existing frame
4. global environment: single frame with bindings of primitive and pre-declared functions and constants
5. program environment: directly extends global environment which contains all user programs

Evaluating a name:

1. check value in current frame
2. if not found, look in enclosing environment
3. if not found, then name is unbound

Evaluating a block:

1. if no constant and variable declaration, no frame is created
2. else extend environment (create frame)
3. store names of constants/variables with no values initially
4. evaluate declarations and bind values to name

Evaluating constant/variable declaration:

1. change binding of name to value in current frame

Evaluating assignment statements:

1. search for name similar to evaluating names
2. if name is constant/unbound return error

Evaluating function applications:

1. evaluate arguments first in current environment
2. extends environment in which function was created (create frame)
3. store parameters in this new frame
4. evaluate function body block

Evaluating lambda expressions/function declarations

1. create function object: 1 circle points to body, 1 circle points to environment where it was created

Use ":=" for constants and ":" for variables

Recursive Processes and CPS

All recursive processes can be converted to iterative processes

1. if recursive operations are always the same, apply the operation before the calling the recurisve function
2. if order of operations do not matter, use helper function
3. if order of operations matter, adjust to reverse order
4. if that's too difficult, use CPS

Continuation Passing Style: passing deferred operation as a function in extra argument to convert any recursive function

Interpreters and Compilers

Interpreter is a program that executes another program

$$\boxed{\begin{array}{c}\text{target}\\ \text{source}\end{array}}$$

Compiler is a program that translates one language to another

$$\boxed{\begin{array}{c}\text{target}_1 \implies \text{target}_2\\ \text{source}\end{array}}$$

Sorting

```
function bubblesort(xs) {
  const len = length(xs);
  for (let i = len - 1; i >= 1; i = i - 1) {
    let p = xs;
    for (let j = 0; j < i; j = j + 1) {
      if (head(p) > head(tail(p))) {
        const temp = head(p);
        set_head(p, head(tail(p)));
        set_head(tail(p), temp);
      }
      p = tail(p);
    }
  }
}
function selection_sort(xs) {
  if (is_null(xs)) {
    return xs;
  } else {
    const x = smallest(xs);
    return pair(
        x,
        selection_sort(remove(x, xs))
    );
  }
}
function smallest(xs) {
    return is_null(tail(xs))
        ? head(xs)
        : head(xs) > smallest(tail(xs))
            ? smallest(tail(xs))
            : head(xs);
}
function merge_sort(xs) {
  if (is_null(xs) || is_null(tail(xs))) {
    return xs;
  } else {
    const mid = math_floor(length(xs) / 2   );
    return merge(
        merge_sort(take(xs, mid)),
        merge_sort(drop(xs, mid))
    );
  }
}
function merge(xs, ys) {
  if (is_null(xs)) {
    return ys;
  } else if (is_null(ys)) {
    return xs;
  } else {
    const x = head(xs);
    const y = head(ys);
    return x < y
        ? pair(x, merge(tail(xs), ys))
        : pair(y, merge(xs, tail(ys)));
  }
}
function take(xs, n) {
  return n === 0
      ? null
      : pair(head(xs), take(tail(xs), n - 1));
}
function drop(xs, n) {
  return n === 0
      ? xs
      : drop(tail(xs), n - 1);
}
function quicksort(xs) {
  return is_null(xs)
    ? null
    : is_null(tail(xs))
    ? xs
    : accumulate(
        append,
        null,
        list(
          quicksort(head(
            partition(tail(xs), head(xs)))
          ),
          list(head(xs)),
          quicksort(tail(
            partition(tail(xs), head(xs)))
          )
        )
      );
}
function partition(xs, p) {
  return pair(
    filter((a) => a <= p, xs),
    filter((a) => a > p, xs)
  );
}
function bubblesort_array(A) {
  const len = array_length(A);
  for (let i = len - 1; i >= 1; i = i - 1) {
    for (let j = 0; j < i; j = j + 1) {
      if (A[j] > A[j + 1]) {
        const temp = A[j];
        A[j] = A[j + 1];
        A[j + 1] = temp;
      }
    }
  }
}
function selection_sort_array(A) {
  const len = array_length(A);
  for (let i = 0; i < len - 1; i = i + 1) {
    let min_pos = find_min_pos(A, i, len - 1);
    swap(A, i, min_pos);
  }
}
function find_min_pos(A, low, high) {
  let min_pos = low;
  for (let j = low + 1; j <= high; j = j + 1) {
    if (A[j] < A[min_pos]) {
      min_pos = j;
    }
  }
  return min_pos;
}
function insertion_sort(A) {
  const len = array_length(A);
  for (let i = 1; i < len; i = i + 1) {
    let j = i - 1;
    while (j >= 0 && A[j] > A[j + 1]) {
      swap(A, j, j + 1);
      j = j - 1;
    }
  }
}
function merge_sort_array(A) {
  merge_sort_helper(A, 0, array_length(A) - 1);
}
function merge_sort_helper(A, low, high) {
  if (low < high) {
    const mid = math_floor((low + high) / 2);
    merge_sort_helper(A, low, mid);
    merge_sort_helper(A, mid + 1, high);
    merge(A, low, mid, high);
  }
}
function merge(A, low, mid, high) {
  const B = []; // temporary array
  let left = low;
  let right = mid + 1;
  let Bidx = 0;
  while (left <= mid && right <= high) {
    if (A[left] <= A[right]) {
      B[Bidx] = A[left];
      left = left + 1;
    } else {
      B[Bidx] = A[right];
      right = right + 1;
    }
    Bidx = Bidx + 1;
  }
  while (left <= mid) {
    B[Bidx] = A[left];
    Bidx = Bidx + 1;
    left = left + 1;
  }
  while (right <= high) {
    B[Bidx] = A[right];
    Bidx = Bidx + 1;
    right = right + 1;
  }
  for (let k = 0; k < high - low + 1; k = k + 1) {
    A[low + k] = B[k];
  }
}
```

Searching

```
function binary_search_array(A, v) {
  let low = 0;
  let high = array_length(A) - 1;
  while (low <= high) {
    const mid = math_floor((low + high) / 2);
    if (v === A[mid]) {
      break;
    } else if (v < A[mid]) {
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }
  return low <= high;
}
```

Memoization

```
function memoize(f) {
  const mem = [];
  function mf(x) {
    if (mem[x] !== undefined) {
      return mem[x];
    } else {
      const result = f(x);
      mem[x] = result;
      return result;
    }
  }
  return mf;
}
function mchoose(n, k) {
  if (read(n, k) !== undefined) {
    return read(n, k);
  } else {
    const result =
      k > n
        ? 0
        : k === 0 || k === n
        ? 1
        : mchoose(n - 1, k) + mchoose(n - 1, k - 1);
    write(n, k, result);
    return result;
  }
}
```

Miscellaneous

```
function get_sublist(start, end, L) {
  function helper(pos, ys, result) {
    if (pos < start) {
      return helper(pos + 1, tail(ys), result);
    } else if (pos <= end) {
      return helper(pos + 1, tail(ys),
                    pair(head(ys), result));
    } else {
      return reverse(result);
    }
  }
  return helper(0, L, null);
}
function insertions(x, ys) {
  return map(
    (k) => append(take(ys, k), pair(x, drop(ys, k))),
    enum_list(0, length(ys))
  );
}
function permutations(ys) {
  return is_null(ys)
    ? list(null)
    : accumulate(
        append,
        null,
        map((x) => map((p) => pair(x, p),
            permutations(remove(x, ys))), ys)
      );
}
function are_permutation(xs1, xs2) {
  return is_null(xs1) && is_null(xs2)
    ? true
    : !is_null(xs1) && !is_null(xs2)
    ? !is_null(member(head(xs1), xs2)) &&
      are_permutation(
        tail(xs1),
        remove(head(xs1), xs2)
      )
    : false;
}
function combinations(xs, k) {
  if (k === 0) {
    return list(null);
  } else if (is_null(xs)) {
    return null;
  } else {
    const s1 = combinations(tail(xs), k - 1);
    const s2 = combinations(tail(xs), k);
    const x = head(xs);
    const has_x = map((s) => pair(x, s), s1);
    return append(has_x, s2);
  }
}
function remove_duplicates(lst) {
  return accumulate(
    (x, xs) => (is_null(member(x, xs))
              ? pair(x, xs)
              : xs
    ),
    null,
    lst
  );
}
function subsets(xs) {
  return accumulate(
    (x, ss) =>
      append(
        ss,
        map((s) => pair(x, s), ss)
      ),
    list(null),
    xs
  );
}
function is_subset(S, T) {
  if (is_null(S)) {
    return true;
  } else if (is_null(T)) {
    return false;
  } else if (head(S) < head(T)) {
    return false;
  } else if (head(S) === head(T)) {
    return is_subset(tail(S), tail(T));
  } else {
    return is_subset(S, tail(T));
  }
}
function sort_ascending(A) {
  const len = array_length(A);
  for (let i = 1; i < len; i = i + 1) {
    const x = A[i];
    let j = i - 1;
    while (j >= 0 && A[j] > x) {
      A[j + 1] = A[j];
      j = j - 1;
    }
    A[j + 1] = x;
  }
}
function list_to_array(L) {
  const A = [];
  let i = 0;
  for (let p = L; !is_null(p); p = tail(p)) {
    A[i] = head(p);
    i = i + 1;
  }
  return A;
}
function array_to_list(A) {
  const len = array_length(A);
  let L = null;
  for (let i = len - 1; i >= 0; i = i - 1) {
    L = pair(A[i], L);
  }
  return L;
}
function 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(y));
      check(tail(y));
    }
  }
  check(x);
  return length(pairs);
}
```