Welcome

CS1101S Studio Session Week 6: *List & Tree Processing*

Niu Yunpeng

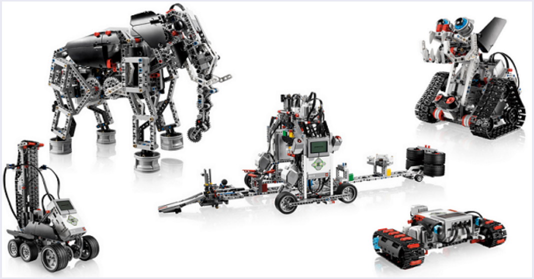
niuyunpeng@u.nus.edu

September 18, 2018

Before We Start

Robot - LEGO Mindstorms ev3

- Robot grouping done randomly in Week 6 Studio
- Robot kit issued in Week 7 Studio
- Robot mission assessment in Week 8 Studio



Before We Start

Robot grouping - work as a team!

Group 81:

- Chen Yuanbo
- Chong Zi Kang
- Lim Kang Yee
- Syed Muhammad Zain Alam

Group 82:

- Dorcas Tabitha Tan
- Eugene Tan Yew Chin
- Ng Jun Rong, Terence
- Shawn Chew

Overview

1. Identity & equality
    - Review
2. List processing
    - From last week
    - List library
3. Tree processing
    - Search
4. One more thing about recursion
    - Permutation
    - Combination

Identity & Equality

To compare identity in Source

- boolean: straightforward;
- string: straightforward;
- numeral: trivial for integers, non-deterministic for non-integers;
- function: two functions are always not identical;
- pair/list: two pairs/lists are always not identical.
- ...

Identity & Equality

To compare equality in Source

```
function equal(a, b) {
    if (is_empty_list(a) && is_empty_list(b)) {
        return true;
    } else if (is_list(a) && is_list(b)) {
        return equal(head(a), head(b)) &&
               equal(tail(a), tail(b));
    } else {
        return a === b;
    }
}
```

Identity & Equality

Exercises

Find out the result of the following statements:

```
true && false || true && false === true;

1 / 5 + 2 / 5 === 3 / 5;

math_pow(2, 53) === math_pow(2, 53) + 1;

equal(pair(1, x => x), pair(1, x => x));
```

Overview

List Processing

Revisit pair & list

- Pair is a simple data structure that stores a head and a list;
- A list is either an empty list or a pair whose tail is a list.

Three ways to represent `pair` and `list`

- Use your code in the Source language;
- Use box-and-pointer diagram (as the list visualizer);
- Use square brackets (as the output in the interpreter).

List Processing

Use `pair` as a data structure

The data structure should at least provide the functions below to use:

- `pair(x, y)`: construct a pair with two elements *a* and *b*;
- `head(some_pair)`: get the first element of a pair;
- `tail(some_pair)`: get the second element of a pair;
- `is_pair(some_pair)`: check whether an object is a pair.

List Processing

List library from last week

Up to last week, we have the following functions to use:

- `list(x, y, z, ...)`: construct a list with *n* elements;
- `head(lst)`: get the first element of a list;
- `tail(lst)`: get the remaining part of a list;
- `is_list(lst)`: check whether an object is a list;
- `is_empty_list(lst)`: check whether an object is a list and empty;
- `length(lst)`: count the number of elements in a list.

List Processing

New library functions for this week

Up to now, the list library supports different kinds of functions:

- List builder: `list`, `build_list`, `enum_list`;
- List getter: `head`, `tail`, `list_ref`, `member`, `is_member`;
- List information: `is_list`, `is_empty_list`, `length`, `equal`;
- List modifier: `append`, `reverse`, `remove`, `remove_all`, `filter`, `map`, `for_each`;
- List converter: `accumulate`, `list_to_string`.

List Processing

List builder

The following functions can be used to build a list:

- `list(x, y, z, ...)`: construct a list with n elements;
- `build_list(n, func)`: construct a list by applying a unary function `func` to every integer from 0 to $n-1$;
- `enum_list(x, y)`: construct a list composed of every integer from x to y (both inclusive).

List Processing

List getter

The following functions can be used to get the element in a list:

- `head(lst)`: get the first element of a list;
- `tail(lst)`: get the remaining part of a list;
- `list_ref(lst, n)`: return the n^{th} element in a list, where the index starts from 0;
- `member(x, lst)`: return the first sublist whose head is identical to x, or an empty list if x if not in the list;
- `is_member(x, lst)`: returns whether x is in the list.

List Processing

List information

The following functions can be used to check the information of a list:

- `is_list(lst)`: check whether an object is a list;
- `is_empty_list(lst)`: check whether an object is a list and empty;
- `length(lst)`: count the number of elements in a list;
- `equal(lst1, lst2)`: check the equality of two pairs/lists/trees.

List Processing

List modifier

The following functions can be used to modify a list:

- `append(xs, ys)`: return a new list that *ys* is appended to *xs*;
- `reverse(lst)`: return a new list in the reverse order of *lst*;
- `remove(x, lst)`: return a new list by removing the first element in the list which is identical to *x*;
- `remove_all(x, lst)`: return a new list by removing all elements in the list whichever is identical to *x*;
- `filter(func, lst)`: apply a unary function *func* to every element in the list, and return a new list which only contains elements whose return value of *func* is true;
- `map(func, lst)`: return a new list by element-wise applying a unary function *func*.

List Processing

List converter

The following functions can be used to convert a list to other formats:

- `accumulate(func, base, lst)`: recursively apply a binary function *func* to every element in a list from right to left. Start from *base* and return the final result. The return value of the binary function *func* should be in the same type as *base* so that we can convert the list into the type of *base*.
- `list_to_string(lst)`: return a string that represents the list in the format of square brackets.

List Processing

Notice

- In the following slides, you are going to see a straightforward version for implementation of the list library.
- You should be aware this implementation is only for demonstration purpose, the actual implementation in Source is different.
- Also, we will consider empty list `[]`, `is_pair`, `is_empty_list` and `list` as built-in system functions.

List Processing

List library implementation

```
// Straightforward implementation for list library in Source
// Niu Yunpeng @ CS1101S 2016 - 2018
function pair(x, y) {
    return oper => oper(a, b);
}

function head(my_pair) {
    return my_pair((m, n) => m);
}

function tail(my_pair) {
    return my_pair((m, n) => n);
}
```

List Processing

List library implementation

```
// This version gives rise to a recursive process.
function build_list(n, func) {
    function build(x) {
        return x === n ? [] : pair(func(x), build(x + 1));
    }
    return build(0);
}

// This version gives rise to an iterative process.
function build_list(n, func) {
    function iter(x, lst) {
        return n < 0 ? lst : iter(x - 1, pair(func(x),lst));
    }
    return build(n - 1, []);
}
```

List Processing

List library implementation

```
// This version gives rise to a recursive provess.
function enum_list(x, y) {
    return x > y ? [] : pair(x, enum_list(x + 1, y));
}

// This version gives rise to an iterative process.
function enum_list(x, y) {
    function iter(n, lst) {
        return n < x ? lst : iter(n - 1, pair(n, lst));
    }
    return iter(y, []);
}

function list_ref(lst, n) {
    return n === 0 ? head(lst) : list_ref(tail(lst), n - 1);
}
```

List Processing

List library implementation

```
function member(x, lst) {
    if (is_empty_list(lst)) {
        return [];
    } else {
        return head(lst) === x ? lst
                                   : member(x, tail(lst));
    }
}

function is_member(x, lst) {
    return !is_empty_list(member(x, lst));
}
```

List Processing

List library implementation

```
function is_list(lst) {
    if (is_empty_list(lst)) {
        return true;
    } else {
        return is_pair(lst) && is_list(tail(lst));
    }
}

function is_empty_list(lst) {
    // Built-in system function
}

function is_pair(lst) {
    // Built-in system function
}
```

List Processing

List library implementation

```
// This version gives rise to a recursive process.
function length(lst) {
    return is_empty_list(lst) ? 0 : 1 + length(tail(lst));
}

// This version gives rise to an iterative process.
function length(lst) {
    function iter(lst, len) {
       return is_empty_list(lst) ? len
                                 : iter(tail(lst), len + 1);
    }

    return iter(lst, 0);
}
```

List Processing

List library implementation

```
// Notice: Week 6 still does not support set_tail yet.
// This version gives rise to a recursive process.
function append(xs, ys) {
    if (is_empty_list(xs)) {
        return ys;
    } else {
        return pair(head(xs), append(tail(xs), ys));
    }
}
```

List Processing

List library implementation

```
// This version gives rise to a recursive process.
function reverse(lst) {
    if (is_empty_list(lst)) {
        return lst;
    } else {
        return append(reverse(tail(lst)), list(head(lst)));
    }
}
```

List Processing

List library implementation

```
// This version gives rise to an iterative process.
function reverse(lst) {
    function iter(origin, reversed) {
        if (is_empty_list(origin)) {
            return reversed;
        } else {
            return iter(tail(origin),
                        pair(head(origin), reversed));
        }
    }

    return iter(lst, []);
}
```

List Processing

List library implementation

```
// Notice: Week 6 still does not support set_tail yet.
// This version gives rise to a recursive process.
function remove(x, lst) {
    if (is_empty_list(lst)) {
        return lst;
    } else if (head(lst) === x) {
        return tail(lst);
    } else {
        return pair(head(lst, remove(x, tail(lst))));
    }
}
```

List Processing

List library implementation

```
// Notice: Week 6 still does not support set_tail yet.
// This version gives rise to a recursive process.
function remove_all(x, lst) {
    if (is_empty_list(lst)) {
        return lst;
    } else if (head(lst) === x) {
        return remove_all(x, tail(lst));
    } else {
        return pair(head(lst, remove_all(x, tail(lst))));
    }
}
```

List Processing

List library implementation

```
// Notice: Week 6 still does not support set_tail yet.
// This version gives rise to a recursive process.
function filter(func, lst) {
    if (is_empty_list(lst)) {
        return lst;
    } else if (func(head(x))) {
        return filter(x, tail(lst));
    } else {
        return pair(head(lst, filter(func, tail(lst))));
    }
}
```

List Processing

List library implementation

```
// Notice: Week 6 still does not support set_head yet.
// This version gives rise to a recursive process.
function map(func, lst) {
    if (is_empty_list(lst)) {
        return lst;
    } else {
        return pair(func(head(lst)), map(func, tail(lst)));
    }
}
```

List Processing

List library implementation

```
// This version gives rise to a recursive process.
function accumulate(func, base, lst) {
    if (is_empty_list(lst)) {
        return base;
    } else {
        return func(head(lst),
                    accumulate(func, base, tail(lst)));
    }
}
```

List Processing

List library implementation

```
// This version gives rise to an iterative process.
function accumulate(func, base, lst) {
    function iter(lst, result) {
        if (is_empty_list(lst)) {
            return result;
        } else {
            return iter(tail(lst), func(head(lst), result));
        }
    }

    return iter(reverse(lst), base);
}
```

Overview

Tree Processing

From list to tree

- The definition of list is: *A list is either an empty list or a pair whose tail is a list*.
- Therefore, the head of a list does not have to be a simple item.
- Indeed, the head of a list may be a list as well.

Tree Processing

Tree Processing

Trees in Computer Science

- Binary Search Tree (BST)
- Minimum Spanning Tree (MST)
- Shortest Path Tree
- AVL Tree
- Red-black Tree
- Skip List
- van Emde Boas Tree
- B Tree
- Fibonacci Tree
- ...

Tree Processing

To use tree as a data structure

The tree library is different from list library:

- `count_leaves(tree)`: count the number of leaves in a tree;
- `tree_map(tree)`: element-wise map on a tree;
- `tree_reverse(tree)`: reverse the order of all leaves in a tree;
- ...

Tree Processing

Search

We shall introduce two algorithms for searching:

- **linear search**: based on list;
- **binary search**: based on tree;

Tree Processing

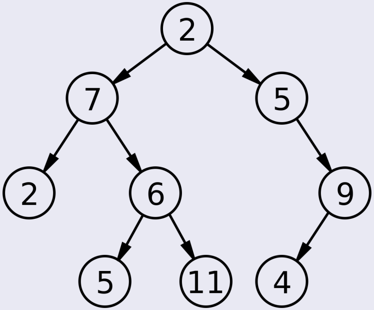
Linear search

```
function linear_search(xs, x) {
    if (is_empty_list(xs)) {
        return false;
    } else {
        return head(xs) === x ? true
                              : linear_search(tail(xs), x);
    }
}
```

Tree Processing

Binary Tree

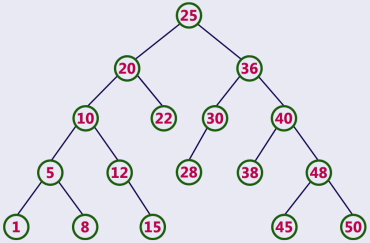
- Each node has two children.



Tree Processing

Binary Search Tree

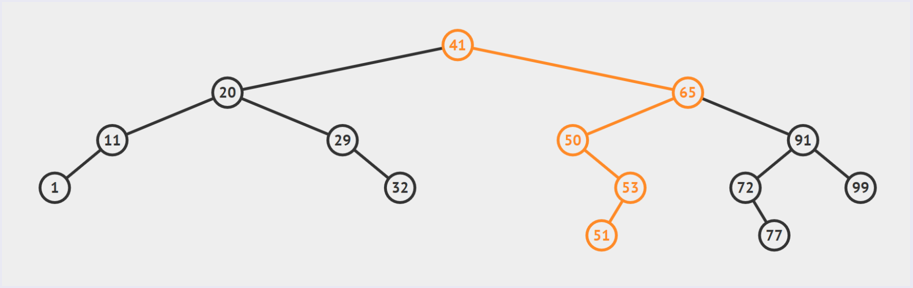
- Each node has two children;
- Left child is always smaller than right child.



Tree Processing

Binary Search

- Decide to go left or right.
- Let's search for 52.



Overview

Recursion

Classical examples of recursion

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)
- Hanoi tower
- Coin change
- Permutation / combination
- ...

Recursion

Examples that we have already covered before...

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)
- Hanoi tower
- Coin change

Recursion

Last things about recursion...

- Permutation
- Combination

Recursion

Permutation

- In mathematics, the notion of **permutation** relates to the act of arranging all the members of a set into some sequence or order.
- Here, we care about how to list all the permutations of a given set.

Example

- Given a set $S = \{1, 2, 3\}$, then:
- The permutation of S is

$$\{\{1,2,3\},\{1,3,2\},\{2,1,3\},\{2,3,1\},\{3,1,2\},\{3,2,1\}\}$$

- The number of permutation of S is 6.

Recursion

Idea about permutation

- There is only 1 permutation of [] - itself.
- For each element x in S:
  - Generate all permutations of $S - x$ recursively;
  - Prepend x in front of each one of them.
- Join all results together.

Recursion

Permutation

```
function permutation(lst) {
    if (is_empty_list(lst)) {
        return list([]);
    } else {
        return accumulate(
            append, [],
            map(x => map(other => pair(x, other),
                         permutation(remove(x, lst))),
                lst));
    }
}
```

Recursion

r-Permutation

- In elementary combinatorics, **r-permutation** usually refers to the act of arranging k elements taken from a set of size n into some order or sequence, where $k \leq n$.

Example

- Given a set $S = \{1, 2, 3\}$, then:
- The 2-permutation of S is

$$\{\{1,2\}, \{2,1\}, \{1,3\}, \{3,1\}, \{2,3\}, \{3,2\}\}$$

- The number of 2-permutation of S is 6.

Recursion

r-Permutation

```
function r_permutation(lst, r) {
    if (r === 0) {
        return list([]);
    } else if (is_empty_list(lst)) {
        return [];
    } else {
        return accumulate(
            append, [],
            map(x => map(other => pair(x, other),
                         r_permutation(remove(x, lst),
                                       r - 1)),
                lst));
    }
}
```

Recursion

k-Combination

- In mathematics, a combination is a way of selecting items from a set such that the order of selection does not matter. A **k-combination** of a set S is a subset of k distinct elements from S.
- The number of k-combinations is equal to the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Example

- Given a set $S = \{1, 2, 3\}$, then:
- The 2-combination of S is

$$\{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$$

- The number of 2-combination of S is 3.

Recursion

Idea abou k-combination

- Instead of arranging elements into a specific order, we need to select a certain number of elements now.
- For each element, we have two choices: to select or to not select.

Hint

- Similar to the coin change problem.
- Instead of counting the number of leaves in the decision tree, we want to list all possible paths from the root to every leaf.

Recursion

k-Combination

```
function k_combination(lst, k) {
    if (k === 0) {
        return list([]);
    } else if (is_empty_list(lst)) {
        return [];
    } else {
        const with_head =
                    map(other => pair(head(lst), other),
                        k_combination(tail(lst), k - 1));
        const without_head = k_combination(tail(lst), k);

        return append(with_head, without_head);
    }
}
```

Recursion

Examples that we have already covered so far...

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)
- Hanoi tower
- Coin change
- Permutation/combination

Recursion

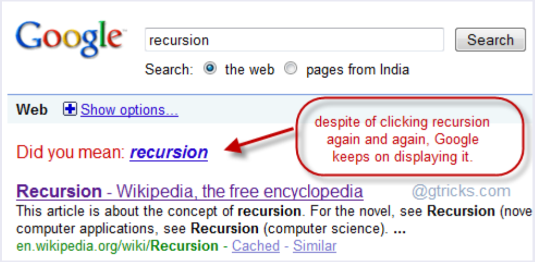
Congratulations!

- You have finished the course from *Department of Recursion, Faculty of Abstraction, University of Wishful Thinking*!

Recursion

Recursion in Google Search

- Try to search for "recursion" in Google:



recursion

Search: ● the web ○ pages from India

Did you mean: *recursion*

despite of clicking recursion again and again, Google keeps on displaying it.

Recursion - Wikipedia, the free encyclopedia

This article is about the concept of **recursion**. For the novel, see **Recursion** (nove computer applications, see **Recursion** (computer science). **...**

en.wikipedia.org/wiki/**Recursion** - Cached - Similar

Thus...

- Now, you know why "*Google is always your best friend*", right?

Studio Group Problems

Let’s discuss them now.

Studio Group Problems

Let's work together!

- We will use a realtime collaborative platform "SourceMD", which uses "CodiMD", which in turn is based on "HackMD".
- For this week, click `https://tinyurl.com/cs1101s-w6`

Studio Group Problems

Question 1

Write the function `map` using `accumulate`. In order to define your `map` function in the Source, you need to give it a name different from the pre-declared name `map`, for example `map_`.

Hint: the function body should have only one line.

Studio Group Problems

Question 2

Write a function called `remove_duplicates` that takes in a list as its only argument and returns a list with duplicate elements removed. *The order of the elements in the returned list does not matter.*

Please provide two different implementations of `remove_duplicates`:

- Use `filter` only
- Use `accumulate` only

Studio Group Problems

Question 3

Write a function makeup_amount which takes as parameters the amount x and a list l of all the coins available, and returns a list of lists, such that each sub-list of the result contains a valid combination to make up x.

Notice:

- l is a list of coins, not a list of kinds of coins. For instance, if there are two 5's in the list, that means we have two 5-cent coins.
- A combination may appear more than once, since it may be using different coins of the same denomination.

Studio Group Problems

Question 4

Implement a function `tree_accumulate` that behaves like `accumulate` but can also work on trees. You may want to use `accumulate` to implement it.

Studio Group Problems

Question 5

Implement a function `accumulate_n` similar to `accumulate` except that it takes as its third argument a list of lists, which are all assumed to have the same number of elements. It applies the designated accumulation function to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a list of the results.

For instance, if we have `const lst = list(list(1, 2), list(3, 4), list(5, 6))`, the return value of `accumulate_n((x, y) => x + y, 0, lst)` is `list(9, 12)`, where $9 = 1 + 3 + 5$ and $12 = 2 + 4 + 6$.

Studio Group Problems

Question 6

Let's use list to represent sets. Each element of the set appears exactly once in its list representation, and the order does not matter. Write a function `subsets` that takes a list as the only parameter, and returns a list of lists, each representing a unique subset of the given set.

End

The End

Copyright