CS1101S Discussion Group Week 6: *List & Tree Processing*

Niu Yunpeng

niuyunpeng@u.nus.edu

September 19, 2017

Overview

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

Identity & Equality

Identity vs Equality

- Identity means exactly the same thing. Usually, they represent just the different namings for the same object.
- Equality means two things hold the same value (or have the same structure). They are two different things, however, their value is equal.

Identity & Equality

Twins...

- Are they the same person?
- Do they look the same?

Think about it...

- Identity?
- Equality?

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

Exercise 1

Find out the result of the following statements:

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

'Source' === "Source";

1101 === "1101";

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

1 / 5 + 1 / 5 + 1 / 5 === 3 / 5;
```

Exercise 2

Find out the result of the following statements:

```
function plus(a, b) {
    return a + b;
}

function add(a, b) {
    return a + b;
}

plus === add;

plus(2, 3) === add(2, 3);
```

Exercise 3

Find out the result of the following statements:

```
function plus(a, b) {
    return a + b;
}

var add = plus;

plus === add;

plus(2, 3) === add(2, 3);
```

Identity & Equality

Exercise 4

Find out the result of the following statements:

```
function plus(a, b) {
    return a + b;
}

function add() {
    return plus;
}

plus === add;

plus === add();
```

Identity & Equality

Exercise 5

Find out the result of the following statements:

```
[] === [];

pair(2, 3) === pair(3, 4);

var my_pair = pair("NUS", "CS1101S");
var list1 = list(1, my_pair, 2);
var list2 = list(3, 4, my_pair);
head(tail(list1)) === head(tail(tail(list2)));
```

Identity & Equality

To compare equality in Source

Two objects are equal in Source if and only if (iff)

- they have the same structure;
- their constituent primitives are identical.

Specification

- boolean, string, numeral: the same as identity;
- empty list: always equal;
- pair, list: equal iff their head and tail are both equal.

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

Exercise

Find out the result of the following statements:

```
equal(1 / 5 + 1 / 5 + 1 / 5, 3 / 5);

equal(list(1, 2), list("1", 2));

equal(list([]), pair([], []));

equal(list(), tail(list([])));

equal(pair(1, function(x) { return x; }),
      pair(1, function(x) { return x; }));
```

Overview

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

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

List library 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`;
- 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.

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 @ CEG NUS 2017
function pair(x, y) {
    return function (m) { return m(x, y); }
}

function head(z) {
    return z(function (p, q) { return p; });
}

function tail(z) {
    return z(function (p, q) { return q; });
}
```

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 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 xs;
    } 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 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

Tree in Computer Science

- Binary Search Tree (BST)
- Minimum Spanning Tree (MST)
- Shortest Path Tree
- AVL Tree
- Red-black Tree
- Skip List
- Fibonacci Tree

Therefore...

- Tree is a very important data structure.

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;

Missions 11 (new this year)

- About **binary search**.

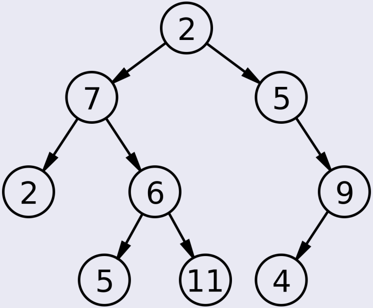
Linear search

```
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

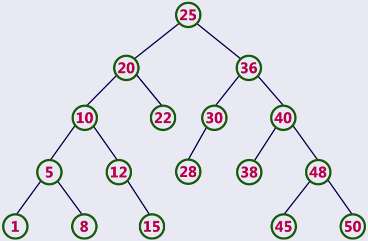
Binary Tree

- Each node has two children.



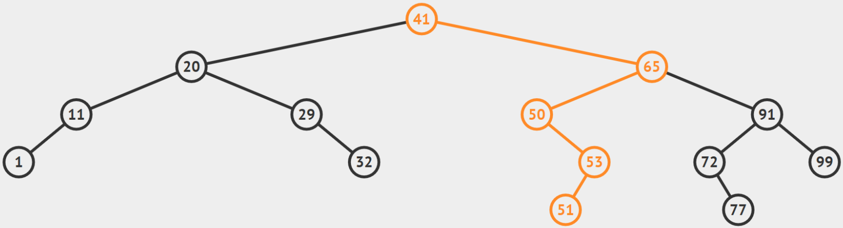
Binary Search Tree

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



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

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.

Idea about permutation

- There is only 1 permutation of [] - itself.
- For each element x in S:
  - Generate all permutations of $S - x$ recursively;
  - Prepand 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(function (x) {
                            return map(function (other) {
                                return pair(x, other);
                            }, permutation(remove(x, lst)));
                          }, lst));
    }
}
```

r-Permutation

- In elementary combinatorics, **r-permutation** usually refers to the act of arranging k elements taken from a set length of 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.

r-Permutation

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

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.

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.

k-Combination

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

        var 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

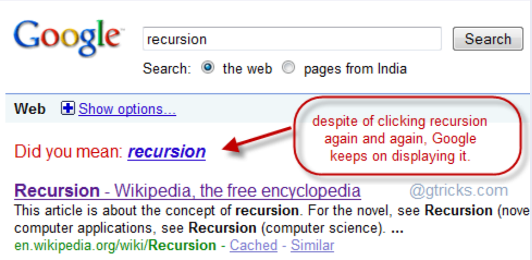
Congratulations!

- 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:



Thus...

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

Let's discuss them now.

The End