CS1101S Studio Session Week 5:
Data Abstraction & List Processing

Niu Yunpeng

niuyunpeng@u.nus.edu

September 11, 2018

Overview

1. Data abstraction
   - What is data
   - To understand data structure
   - To use data structure
2. Pair & list
   - Pair processing
   - An "insider" problem
   - List processing
   - Exercises
3. Identity & equality
   - Identity in Source
   - Equality in Source

Data Abstraction

What is data?

- Data is the storage of information.
- Two kinds of information: **states** & **procedures**.
- Procedures are the manipulation of states.

Data in the Source

- *To represent states*: use variables;
- *To represent procedures*: use functions.

Data Abstraction

Still remember value() from coin_change?

```
function value(kind) {
    return kind === 1 ?   5 :
           kind === 2 ?  10 :
           kind === 3 ?  20 :
           kind === 4 ?  50 :
           kind === 5 ? 100 :
           0;
}
```

Data Abstraction

What is `value()` about?

- We want to know the value for each kind of coins. We certainly can store them in variables like `coinA`, `coinB`, `coinC`, etc.

What if we have too many kinds of coins?

- We then need a ***well-organized structure*** to store all the data.

Data Abstraction

What is *data structure*?

- Data structure provides us with a *well-organized* way to store all related information as a collection.
- Data structure should provide functions so that we can arbitrarily get/change the values inside.
  - `getters`
  - `setters`
  - ...

Data Abstraction

Data structure & black-box abstraction

- Data structure is a black-box.
- We can use it to store and retrieve data without knowing things inside.

Data Abstraction

Use data structure with `value()`

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

- `initialize()`: to initialize a new data structure to store different kinds of coins and their respective values;
- `add_new_kind(id, value)`: to add a new kind of coins to an existing data structure with a unique identifier and its value;
- `get_value(id)`: to get the corresponding value of a certain kind of coins by its unique identifier.

Data Abstraction

To use data structure

- Revisit the example on the lecture notes - rationals.
- Try to understand how to *design and build* a *tailor-made data structure* for a specific problem.

Data Abstraction

Rational numbers

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

- `make_rat(num, denom)`: make a rational number with its numerator and its denominator;
- `get_num(rat)`: get the numerator of a rational;
- `get_denom(rat)`: get the denominator of a rational;
- `add_rat(a, b)`: add two rationals *a* and *b*;
- `sub_rat(a, b)`: subtract two rationals *a* and *b*;
- `mul_rat(a, b)`: multiply two rationals *a* and *b*;
- `div_rat(a, b)`: make a division of two rationals *a* and *b*;
- `equal_rat(a, b)`: check whether two rationals are equal;
- `rat_to_string(rat)`: convert a rational to a string.

Data Abstraction

Make a rational number

```
function make_rat(num, denom) {
    const divider = gcd(num, denom);
    return pair(num / divider, denom / divider);
}

function get_num(rat) {
    return head(rat);
}

function get_denom(rat) {
    return tail(rat);
}
```

Data Abstraction

Rational number calculation

```
function add_rat(a, b) {
    return make_rat(get_num(a) * get_denom(b) +
                    get_num(b) * get_denom(a),
                    get_denom(a) * get_denom(b));
}

function sub_rat(a, b) {
    return make_rat(get_num(a) * get_denom(b) -
                    get_num(b) * get_denom(a),
                    get_denom(a) * get_denom(b));
}
```

Data Abstraction

Rational number calculation

```
function mul_rat(a, b) {
    return make_rat(get_num(a) * get_num(b),
                    get_denom(a) * get_denom(b));
}

function div_rat(a, b) {
    return make_rat(get_num(a) * get_denom(b),
                    get_denom(a) * get_num(b));
}
```

Data Abstraction

Others

```
function equal_rat(a, b) {
    return get_num(a) === get_num(b) &&
           get_denom(a) === get_denom(b);
}

function rat_to_string(rat) {
    return get_num(rat) + "/" + get_denom(rat);
}
```

Overview

Pair & 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.

Three ways to represent a `pair`

- 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).

Notice

- The same applies to `list` later.

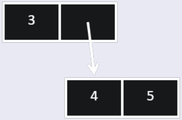
Pair & List Processing

Three ways to represent a pair

- 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).

Example

- `const x = pair(3, pair(4, 5));`



- [3, [4, 5]]

Pair & List Processing

Consider: `make_one_out_of_two`

```
function make_one_out_of_two(a, b) {
    return oper => oper(a, b);
}

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

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

const my_pair = make_one_out_of_two(1, 2);
first(my_pair);
```

Pair & List Processing

From pair to list

- Sometimes, we need to store more than 2 variables in a data structure.
- Without list, we have to

```
pair(3, pair(1, pair(4, pair(1, pair(5, ...)))));
```

- With list, we only need to

```
list(3, 1, 4, 1, 5, ...);
```

Pair & List Processing

Formal definition

- A list is either an empty list or a pair whose tail is a list.

Pair & List Processing

Use `list` as a data structure

Up to now, 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.

Pair & List Processing

Recap: 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).

Pair & List Processing

Exercise 1

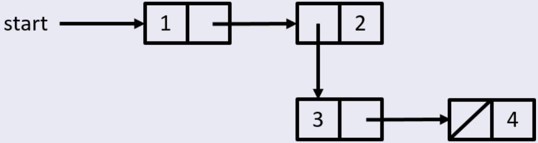
Draw the box-and-pointer diagrams for each one of them below:

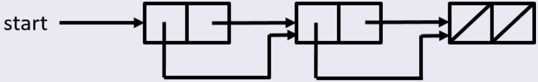
```
const lstA = list(list([], 1, list([], 2, [])),
                  3,
                  list([], 4, []));

const p1 = pair(4, []);
const p2 = pair(3, p1);
const lstB = list(1, pair(2, p2));

const z1 = pair(1, 3);
const z2 = list(3, z1);
const lstC = list(tail(z2), z1, head(z1));
```

Pair & List Processing

Exercise 2

Write Source programs which can produce the box-and-pointer diagrams below (*The head of the whole list should be pointing to "start"*):





Pair & List Processing

Exercise 3

Given two lists of the same length `xs` and `ys`, try to construct a 3^{rd} list of the same length in which each element is a pair composed of the element on the same position from `xs` and `ys`. Your function name should be `make_pairs`.

Example

For example, for `make_pairs(list(1, 2, 3), list(11, 12, 13))`, it should return `list(pair(1, 11), pair(2, 12), pair(3, 13))`.

Pair & List Processing

Exercise 3

Now, generalize this concept by defining a new function. Given two lists of the same length `xs` and `ys`, try to construct a 3^{rd} list of the same length in which each element is the result of applying a certain zip function to the two elements on the same position from `xs` and `ys`. Your function name should be `zip`.

Example

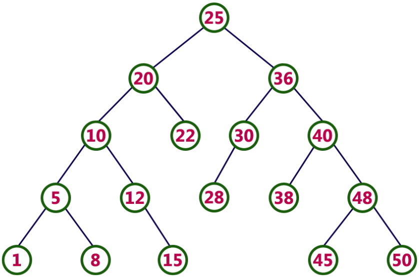
For example, if we apply

```
zip((x, y) => x * y,
    list(1, 2, 3),
    list(11, 12, 13));
```

it will return `list(11, 24, 39)`.

Exercise 4 - BST

A binary search tree (BST) is either an empty list or a list with three elements: a left child BST, a number x, and a right child BST. Notice that every number in the left BST is smaller than the number x, and every number in the right BST is larger than the number x.



Pair & List Processing

Exercise 4 - BST

The first step to understand how to use BST is to have a try. Given 5 numbers 1...5, try to store them in a BST. Then, you should use the 3 ways to represent this list (notice: BST is just a special kind of list). The answer may not be unique.

Exercise 4 - BST

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

- `get_min(tree)`: get the smallest element in a BST;
- `get_max(tree)`: get the largest element in a BST;
- `search(tree, x)`: check whether a number exists in a BST;
- `height(tree)`: get the height of a BST;
- `bst_to_list(tree)`: convert a BST into a list.

Task

Implement all these functions mentioned above and other necessary functions that should be supported by a BST library.

Overview

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;
```

Identity & Equality

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);
```

Identity & Equality

Exercise 3

Find out the result of the following statements:

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

const 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);

const my_pair = pair("NUS", "CS1101S");
const list1 = list(1, my_pair, 2);
const 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, x => x),
      pair(1, x => x));
```

Let's discuss them now.

The End

Copyright

This work is mostly an original by *Niu Yunpeng*. It may either directly or indirectly benefit from the previous work of Martin Henz, Cai Deshun. For illustration purposes, some pictures in the public domain are used. Upon request, detailed acknowledgments will be provided.