Welcome

CS1101S Discussion Group Week 9: *Mutable Data & Array*

Niu Yunpeng

niuyunpeng@u.nus.edu

October 17, 2017

Overview

1. Mutable data
   - States & change of states
   - Environment model
   - Mutable data structure

2. Loop & array
   - while and for
   - Array

Mutable Data

Representation of data in Source

- Data is the storage of information.
- Two kinds of information: **states** & **procedures**.
- *To represent states*: use variables;
- *To manipulate states*: use functions.

Mutable Data

Before Week 8

- Pure functional programming.
- Substitution model.
- Return value do not change if values of arguments are the same.

After Week 8

- Stateful programming.
- Environment model.
- Return value may vary even if values of arguments are the same.

Mutable Data

For stateless programming...

- Once a variable has been defined, its value cannot be changed.
- If we really want to change its value, it has to be assigned to a new variable.

Mutable Data

The concept of memory allocation

- When we define a variable, the interpreter will allocate a position in memory (random access memory, RAM) randomly so that we can use it any time we want.
- The name is actually the reference to this position in memory.
- Whenever we call the name, the interpreter will just look for the value stored at that position in memory.

Understanding

- A variable is like a **changeable container**.

Mutable Data

Why can we change the value of a variable?

- Before, when we want to have a new value of a variable, we allocate a new position in memory.
- However, it is not necessary for us to do this at all (because this is in fact a waste of space in memory).
- We can just update the value stored at the original position. When we call that name after that, the interpreter will still look up for the same position and a new value will be found.

Mutable Data

Environment model

- Even though we supply the same values for all arguments, the return value of a function may still vary.
- Due to this, the substitution model breaks down.
- We have to introduce a new one and a better one:

  environment model
- It is an *upgrade* of substitution model + variable scoping.

Mutable Data

Frame

- Each function call creates a new frame (similar to scope for variables).
- The initial frame is called global frame (global scope).
- Each frame contains a series of bindings of names and values.

Environment

- In order to find the variable, it is possible to search starting from the current local scope up to the global scope.
- Thus, all these corresponding frames are deterministic to the value of the variable. They are called the environment, a sequence of frames.

Mutable Data

Frame & environment

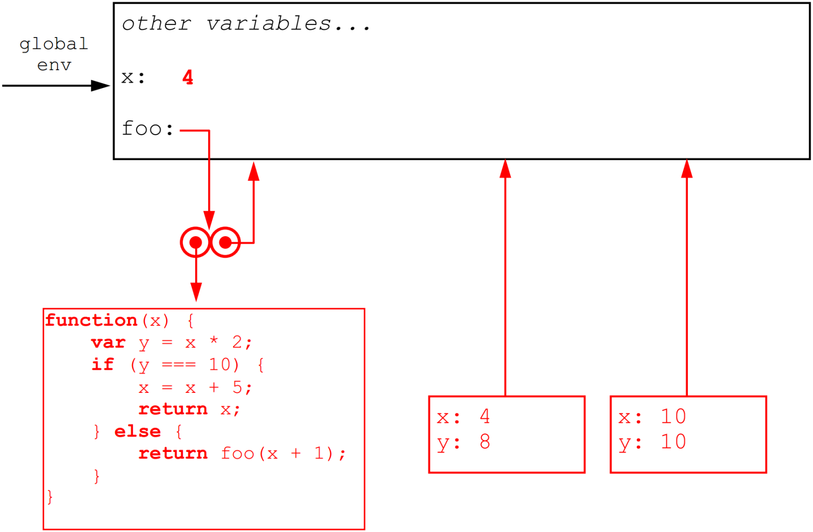
- Looks like a list.
- The head is the current frame, while the tail is pointing to the parent frames, called its **enclosing environment**.

Mutable Data

What happens when we call a function?

- Create a new frame to extend the current environment.
- Evaluate actual arguments and bind their values to formal parameters.
- Local variables are bound to `undefined`.
- Evaluate the function body and send the return value to the enclosing environment.

Mutable Data



```
function(x) {
    var y = x * 2;
    if (y === 10) {
        x = x + 5;
        return x;
    } else {
        return foo(x + 1);
    }
}
```

Mutable Data

Exercise 1

- Assume the program stops at the comment.
- Draw the environment model diagram gradually.
- Also, identify the value of x at the point of that comment.

Mutable Data

Exercise 1.1

```
var x = 0;

function environmentalist() {
    x = x + 1;

    function model(x) {
        x = x + 2;
        return x;
    }

    return model(x);
}
// Here
environmentalist();
x = environmentalist();
```

Mutable Data

Exercise 1.2

```
var x = 0;

function environmentalist() {
    x = x + 1;

    function model(x) {
        x = x + 2;
        // Here
        return x;
    }

    return model(x);
}

environmentalist();
x = environmentalist();
```

Mutable Data

Exercise 1.3

```
var x = 0;

function environmentalist() {
    x = x + 1;

    function model(x) {
        x = x + 2;
        return x;
    }

    return model(x);
}

environmentalist();
// Here
x = environmentalist();
```

Mutable Data

Exercise 1.4

```
var x = 0;

function environmentalist() {
    x = x + 1;

    function model(x) {
        x = x + 2;
        return x;
    }

    return model(x);
}

environmentalist();
x = environmentalist();
// Here
```

Mutable Data

Exercise 2

- The whole program has been evaluated.
- Draw the environment model diagram.

Mutable Data

Exercise 2.1

```
var x = 4;

function foo(x) {
    var y = x * 2;

    if (y === 10) {
        x = x + 5;
        return x;
    } else {
        return foo(x + 1);
    }
}

foo(x);
```

Mutable Data

Exercise 2.2

```
function alpha(x) {
    var y = 3;

    function beta(x) {
        y = y + x;
        return y;
    }

    return beta;
}

var haha = alpha(5);
haha(1);
```

Mutable Data

Before today - immutable data structure

- A collection of data into one object.
- Data inside cannot be changed.
- Constructor, accessor, predicate, printer, ...

After today - mutable data structure

- A collection of data into one object.
- Data inside can be changed.
- Constructor, accessor (getter), mutator (setter), predicate, printer, ...

Mutable Data

Mutable pair/list

- set_head(pr, x): set the head of a pair to become x;
- set_tail(pr, y): set the tail of a pair to become y.

Caution

- Remember identity & equality;
- Remember the concept of memory allocation.

Mutable Data

Things you can do for pair/list

- Re-write some parts of the list library;
- Create a cycle for a list.

Your task today

- Can you write a program to detect the number of cycles in a given list (or return 0 if none)?

Mutable Data

Mutable data structure

- Linked list
- Double-way linked list
- Queue
- Stack
- Table
- ...

Mutable Data

Linked list / double-way linked list 1

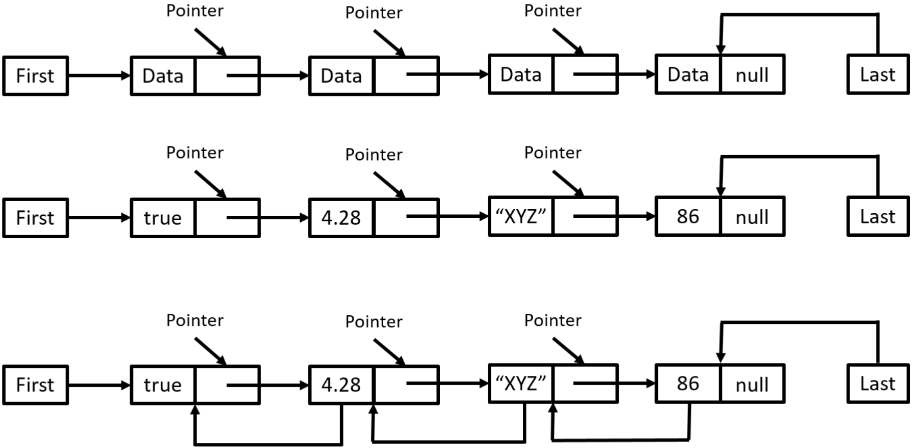
- `make_linked_list()`: create an empty linked list;
- `get_first(lst)`: get the first node of the linked list;
- `get_last(lst)`: get the first node of the linked list;
- `get_next(node)`: get the next node in the linked list;
- `get_prev(node)`: get the last node in the linked list;
- `get_data(node)`: get the data stored in the current node.

Mutable Data

Linked list / double-way linked list 2

- `prepend(lst, x)`: add `x` to the front of the linked list;
- `append(lst, x)`: add `x` to the rear of the linked list;
- `add_before(node, x)`: add `x` before the node;
- `add_after(node, x)`: add `x` after the node;
- `remove_first(lst)`: delete the first node in the linked list;
- `remove_last(lst)`: delete the last node in the linked list;
- `delete(node)`: delete the selected node in the linkd list;
- `empty(lst)`: delete all items in the linked list;
- `is_empty_linked_list(lst)`: check if a linked list is empty.

Mutable Data



Mutable Data

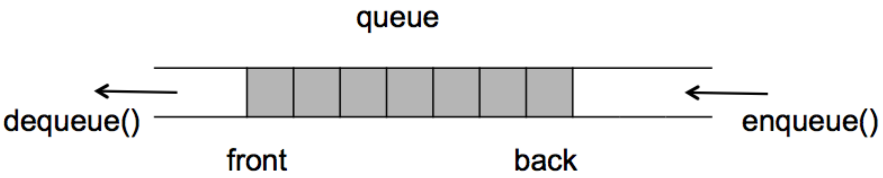
Queue - first in first out (FIFO)

- `make_queue()`: create an empty queue;
- `enqueue(queue, x)`: add `x` to the end of the queue;
- `dequeue(queue)`: delete the first item of the queue;
- `peek(queue)`: retrieve the value the first item of the queue;
- `empty(queue)`: delete all items in the queue;
- `is_empty_queue(queue)`: check if a queue is empty.

Notice

- `dequeue(queue)` and `peek(queue)` will raise an error if the queue is empty.

Mutable Data



Mutable Data

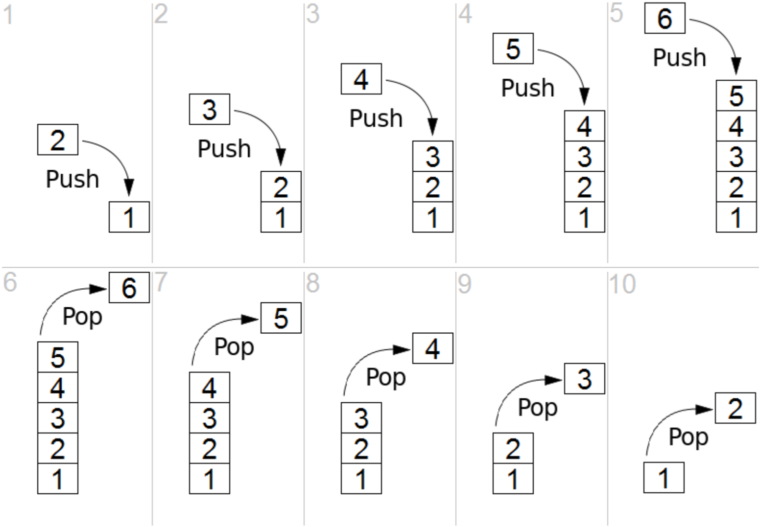
Stack - first in last out (FILO)

- `make_stack()`: create an empty stack;
- `push(stack, x)`: add `x` on the top of the stack;
- `pop(stack)`: delete the first item on the top of the stack;
- `peek(stack)`: retrieve the first value on the top of the stack;
- `empty(stack)`: delete all items in the stack;
- `is_empty_stack(stack)`: check if a stack is empty.

Notice

- `pop(stack)` and `peek(stack)` will raise an error if the stack is empty.

Mutable Data



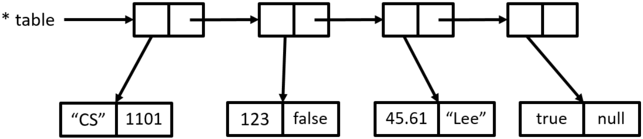
Mutable Data

Table

- `make_table()`: create an empty table;
- `contains(key, table)`: check if the table contains this key;
- `put(key, value, table)`: insert a new entry to the table;
- `lookup(key, table)`: return the value corresponding to the specified key in the table, or `undefined` if the key is not found;
- `empty(table)`: delete all entries in the stack;
- `is_empty_table(table)`: check if a table is empty.

Mutable Data

key	value
“CS”	1101
123	false
45.61	“Lee”
true	null
…	…



Mutable Data

Usage of mutable data structure

- Stack:
  - The interpreter uses stack to implement recursion.
- Table:
  - The binding between names and values in a frame is a table;
  - Later, we will use table to implement memoization.

Overview

1. Mutable data
   - States & change of states
   - Environment model
   - Mutable data structure

2. Loop & array
   - while and for
   - Array

Loop & array

while and for loop

- There are two kinds of loops available in Source:

  while and **for**

- They can be converted to each other.

```
for (E1; E2; E3) {
    // ...
}

E1;
while (E2) {
    // ...
    E3;
}
```

Loop & array

continue and break

- continue: terminates the current round of the loop and continues the loop with the next round.
- break: terminates the current round of the loop and also terminates the entire loop.

Loop & array

Array

- Array is effectively the same as list.
- Empty array: `[]`
- Array with n element: `[1, 2, ..., n]`
- Access m^{th} element: `arr[m]`
- Array assignment: `arr[m] = "cs"`
- Array length: `array_length(arr)`

Loop & array

Array and list

- List can be implemented using array.
- `pair(a, b)` is just `[a, b]`
- `list(a, b, c, d)` is just `[a, [b, [c, [d, []]]]]`

Loop & array

How to use array

- Implement data structure
- Implement sorting algorithm
- Use together with loop

Discussion Group Problems

Let's discuss them now.

End

The End