Midterm survey closes today 😊
Winner announcement of ‘Force Crystals’ and ‘Alien Pancakes’:
Lecture 16, 17/10
Re-Midterm

16 Oct 2012 (Tues)
7:30 pm, venue: TBC
Recap: Last Lecture

• Introduced assignment: $v = E$;

• By introducing state and assignment, the Revised Substitution Model (RSM) is broken

• Today, we need to learn how to cope with this new environment
Assignment statement

\(<\text{name}\> = \langle\text{new-value}\>;;

• \(<\text{name}\>\) is a symbol, not evaluated.

• \(<\text{new-value}\>\) is evaluated, then its value is assigned to the variable \(<\text{name}\>\) in the environment.

• That is, from now on, \(<\text{name}\>\) will evaluate to \(<\text{new-value}\>\)

• Compare with expression statement
  \(<\text{value-1}\> == <\text{value-2}\>;;\)
Today’s Agenda

• Environment Model
• Mutable Data
  – Mutating lists
  – Queues
  – Tables
Environment Model

• RSM doesn’t work anymore!

• An Environment
  – determines the context in which an expression should be evaluated
  – Every expression **HAS** an environment in which is evaluated. If none is created by the user, then the expression is evaluated in the **Global Environment**.
Environment Model

The Global Environment consists of a single *frame* with bindings of function variables to their implementation.

```
alert: impl
prompt: impl
pair: impl
list: impl
```
Environment Manipulation

• Every function stores the bindings of its global variables in its environment.
• Every function call *extends* this environment by a binding
  – of the parameter variables to the actual arguments, and
  – of the local variables to undefined.
Environment Extension

• So far, we have not spent much time worrying about how environments are extended.
• In the presence of assignment, we need to be clear about this.
Environment Model

- An environment is a *sequence of frames*.
- Each frame contains *bindings* of symbols and values.
- A frame points to its *enclosing environment*, the next one in the sequence.
- Extending an environment means *adding* a new frame inside the old one.
Accessing Environments

• To evaluate a symbol, look up its value in the current frame.
• If not found in current frame, then look in enclosing environment, and so on.
Example

• A, B, C, D are environments (C same as D)
• Value of x in environment A is: 7
• Value of x in environment B is: 3
• x in Frame II shadows x in Frame I
• Value of w in environment A? unbound
Evaluation Rules

• To evaluate an application:

\[ \text{fun}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \]

– Evaluate the subexpressions of the application \textit{in the current environment}.

– Apply the value of the function subexpression to the values of the argument subexpressions.
Evaluation Rules

• **Var declarations**: `var <name> = <exp>;`
  - Applying enclosing function def extends function environment by binding of `<name>`

• **Variable occurrence** `<name>` searches environment from current frame

• **Assignment statement**: `<name> = <exp>;`
  - searches environment for `<name>` from current frame, changes its binding to value of `<exp>`
Example

- Access \( x \) in A leads to 3
- Evaluate \( x = 13; \) in B
More rules

• Function expression:
  function(<var1>,<var2>,...) { <body> }
  – Creates a function object, represented as two circles.
  – One points to the code (text).
  – The other points to the environment in which the function expression was evaluated; called the *closing environment*.
  – A function therefore remembers the environment in which it was created.
Function application

- Create a new frame that points to the closing environment of the function.
- In this new frame, bind the formal parameters to the actual arguments.
- Bind variables declared with `var` to `undefined`.
- Then evaluate the body of the function in this new frame.
JavaScript toplevel

- Variable declaration outside any function:
  \[
  \text{var <name> = <expr>;} \\
  - \text{Evaluate <expr> in global environment} \\
  - \text{If <name> is not defined in global environment, define it, and bind it to undefined} \\
  - \text{Change binding of <name> to eval result}
  \]

- Function definition outside any function:
  \[
  \text{function <name>(args) \{ <body> \}} \\
  \text{similar to} \\
  \text{var <name> = function(args){<body>}};
  \]
After evaluating

```javascript
function square(x) { return x * x; }
```

in the global environment.
Example

- Now, evaluate \texttt{square(5)} in the global env.
- Create new frame that points to \texttt{square}'s closing environment.
- In this frame, bind \texttt{x} to 5, evaluate \texttt{x*x};
Example 2 (from last lecture)

function make_withdraw(balance) {
    return function(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "error";
        }
    }
}

var W1 = make_withdraw(100);
W1(50) → 50
W1(60) → "error"
W1(40) → 10
Example 2 (from last lecture)

After defining procedure `make_withdraw`

- other bindings...
- `make_withdraw`:
  - parameters: balance
  - body: return function ...

- After defining procedure `make_withdraw`
Example 2

Let’s evaluate

\[
W_1 = \text{make\_withdraw}(100);
\]

Evaluate this:

```javascript
return function(amount) {
  if (balance >= amount) ... 
}
```
Example 2

Let’s evaluate

\[ W1 = \text{make\_withdraw}(100); \]

- global env
- `make_withdraw` function
  - `W1`:
  - `balance: 100`
  - parameters: `amount`
  - body: `if (balance >= amount) ...`

- parameters: `balance`
- body: `return function ...`
Example 2

Evaluating \( W_1(50) \)

```
make_withdraw: ...

W1:

balance: 100

amount: 50

if (balance >= amount) {
    balance = balance - amount;
    return balance;
} else {
    return "error";
}
```
Example 2

Environment after evaluating \texttt{W1(50)}
Example 2

Environment after evaluating

```javascript
var W2 = make_withdraw(100)
```

W2 has its own state variable.
Mutable Data

• Assignment gives us the ability to create *mutable* data, i.e. data that can be modified.
  – e.g. bank account

• In SICP Chap. 2, all our data were *immutable*. We had
  – Constructors, selectors, predicates, printers
  – But no mutators.
Mutable Data

After creating a pair with `pair`, the head and tail can be changed using `set_head` and `set_tail`
Mutable Data

• \texttt{set\_head(p, x)} changes head pointer of \texttt{p} to point to \texttt{x}.

• \texttt{set\_tail(p, y)} changes tail pointer of \texttt{p} to point to \texttt{y}.
Initially, \( x, y \) are as shown.

Effect of:
\[
\text{set\_head}(x, y)
\]
Example

Effect of: \texttt{set\_tail}(x, y)

Initially, x, y are as shown.
Warning!
• Be careful when using mutators!

```
var a = list(1, 2, 3);
var b = list(6, 5);
var c = append(a, b);
c  ➔ [1, [2, [3, [6, [5, []]]]]]

set_head(b, 100);

b  ➔ [100, [5, []]
c  ➔ [1, [2, [3, [100, [5, []]]]]]
```

Mutating b changes c as well!
Initially,

Structure of c

Mutating b
Another Example

```javascript
var a = list(1, 2, 3);
set_head(tail(tail(a)), a);
```

**Question**

What is `length(a)`?

Cyclical structure!
Data Structure: Queue

- A Queue is a **sequence of items** with the **first-in-first-out (FIFO)** property.
  - Items are removed in the order they were inserted.
- Items are inserted at the **rear** of the queue, and removed from the **front** (or **head**) of the queue.
Queue: Specs

• Constructor
  – make_queue: function
    • Arguments: none
    • Returns: a new, empty queue

• Predicate
  – is_empty_queue: function
    • Arguments:
      – Q: a queue
    • Returns: true if Q is empty, false otherwise
Queue: Specs

- Selector
  - `Front_queue` : function
    - Arguments:
      - Q: a queue
    - Returns: the item at the front of Q. Q is unchanged. Signals an error if Q is empty.
Queue: Specs

• Mutator
  – \texttt{Insert\_queue}: function
    • Arguments:
      – Q: a queue
      – x: an item to be inserted.
    • Effects: Q is mutated so that x is added to the rear of Q.
    • Returns: the modified Q
Queue: Specs

• Mutator
  – delete_queue: function
    • Arguments:
      – Q: a queue
    • Effects: Q is mutated so that the item at the front is removed. An error is signaled if Q is empty before the deletion.
    • Returns: the modified Q.
Queue: implementation

- The items are stored in a list.
- But a pair is used to keep track of the front and rear of the queue.
  - This is for efficiency reasons.
Queue: implementation

- Adding an item.
Queue: implementation

• Removing an item.

```
front-ptr

a -> b -> c -> d -> rear-ptr
```
Queue: implementation

First, define helper functions:

```c
front_ptr = head;
rear_ptr  = tail;
set_front_ptr = set_head;
set_rear_ptr  = set_tail;
```
function is_empty_queue(q) {
    return is_empty_list(front_ptr(q));
}

function make_queue() {
    return pair([], []);
}

function front_queue(q) {
    if (is_empty_queue(q)) { "error"
    } else {
        return head(front_ptr(q));
    }
}
Queue: implementation

function delete_queue(q) {
    if (is_empty_queue(q)) { "error"
    } else {
        set_front_ptr(q, tail(front_ptr(q)));
        return q;
    }
}
Queue: implementation

```javascript
function insert_queue(q, item) {
    var new_pair = pair(item, []);
    if (is_empty_queue(q)) {
        set_front_ptr(q, new_pair);
        set_rear_ptr(q, new_pair);
    } else {
        set_tail(rear_ptr(q), new_pair);
        set_rear_ptr(q, new_pair);
    }
    return q;
}
```
Useful mutator

• Remove front item, and return it.

```javascript
function dequeue(q) {
    var result = front_queue(q);
    delete_queue(q);
    return result;
}
```
Using a Queue

• Printing documents

Print Queue

- Lee’s document
- Mary’s document
- Lee’s email
- Tom’s paper
- Joe’s picture
Data Structure: Table

• A table is a sequence of key-value pairs.

• The key is used to index the table, and the value is associated with the key.

• A key-value pair is also called an entry.
Table: Specs

- **Constructor**
  - `make_table`: function
    - Arguments: none
    - Returns: a new, empty table

- **Predicate**
  - `is_empty_table`: function
    - Arguments: T, a table
    - Returns: `true` if T is empty, `false` otherwise
Table: Specs

• Selector
  – lookup (get) : function
  • Arguments:
    – K: a key to search for an entry
    – T: the table to be searched
  • Returns: the key-value pair in T whose key matches K. If no entry found, error
Table: Specs

• Mutator
  – `insert` (put) : function
    • Arguments:
      – K: a key
      – V: a value to be associated with K
      – T: the table to be inserted
    • Effects: if an existing entry in T matches K, then the entry’s value is changed to V. Otherwise, a new entry K-V is added to T. T is modified.
    • Returns: undefined
Table: implementation

• Key-value entries are stored as pairs.
• Entries are stored in a list.
• List has a head, to allow first entry to be changed.
Table: implementation

function make_table() { return list('table'); }

function empty_table(table) {
    return equal(table, list('table')); }

function lookup(key, table) {
    return tail(assoc(key, tail(table))); }

function assoc(key, records) {
    if (is_empty_list(records)) { "error";
    } else if (equal(key, head(head(records)))) {
        return head(records);  
    }  
}
Table: implementation

```javascript
function check_assoc(key, records) {
    if (is_empty_list(records)) { return false;
    } else if (equal(key, head(head(records)))) {
        return true; }
    } else {
        return check_assoc(key, tail(records)); } }

function insert(key, value, table) {
    if (check_assoc(key, tail(table))) { 
        var record = assoc(key, tail(table));
        set_tail(record, value);
    } else {
        set_tail(table, pair(pair(key, value),
                        tail(table))); } }
```
Table: implementation

function delete(key, table) {
    Homework!
}
2D tables

- Two-dimensional tables are possible:
  - Two keys needed
  - Entry: key1-key2-value

- Implementation:
  - Nested 1D tables (subtables)
  - key1 used to locate a 1D subtable
  - key2 used to index within this subtable

- put, and get are simply lookup and insert on a 2D table
# 2D tables

- **table**
  - letters
    - a 97
    - b 98
  - math
    - + 43
    - - 45
    - * 42
Using Tables

• Problem: determine word frequency in a text message.
• e.g. given

```r
list('mary', 'had', 'a', 'little', 'lamb', 'little', 'lamb', 'little', 'lamb', 'mary', 'had', 'a', 'little', 'lamb', 'whose', 'fleece', 'was', 'white', 'as', 'snow')
```

• Determine how many times each word occurs.
  
mary: 2, little: 4, lamb: 4, ...
Solution

• Maintain a table
  – Word is the key
  – Count is the value

• cdr down the message list
  – For each word w
    • If w exists in table, then increment its value
    • Otherwise, add entry (w,1) to the table
  – Ignore punctuations
Queues using tables

• empty-Q = empty-table
• To insert item $M$, insert entry $(N, M)$ into table.
  – $N$ is a running number that we automatically generate. This serves as a unique key in the table.
  • when table is first created, $N=0$
  – After inserting, we increment $N$.
  – $N$ also indicates the order in which the items were inserted.
Queues using tables

• To delete an item from the front of the queue,
  – Lookup key $S$ in the table
  – $S$ is the smallest key
  – Delete that entry
  – Increment $S$

• To think about:
  – How to implement \textit{front\_queue}?
  – What is the time complexity of these functions?
Questions of the Day

• Can you implement tables using queues?
  – Hint: consider using 2 queues.
  – Think of how to represent an empty table.
  – How to implement table operations (e.g. lookup, insert!) using queue operations?

• Can you implement table more efficiently?
Summary

- **Environment Model** replaces RSM.
  - Frames are local repositories of state

- To evaluate a function in the Environment Model:
  1. Create a **NEW** frame in the closing environment (second pointer of function value)
  2. Bind variables in new frame
  3. Evaluate body of the function (first pointer of function value) in the new frame
Summary

• Assignment allows us to create mutable data structures.

• `set_head, set_tail` mutate (mutable) lists.

• Be careful when mutating things!

• Queues and tables are useful data structures.

• Data structures can be used to implement other data structures!