4A: Introduction to Data Abstraction

CS1101S: Programming Methodology

Low Kok Lim

August 31, 2016

Recap: Assessment Overview

- 35%: Missions
- 5%: Discussion group participation
- 15%: Midterm Assessment (28 Sep, Wed, 10am)
- 15%: Practical Assessment (10 Nov, Thu, 7pm)
- 30%: Final Assessment (23 Nov, Wed, 9am)

Reading

SICP 2.1, 2.2.1

Source Week 5

Identifiers

`Math.PI` is not *really* an identifier: The dot plays a peculiar role in JavaScript. *For now*, we treat `Math.PI` *as if it were an identifier*.

Source Week 5

Identifiers

`Math.PI` is not *really* an identifier: The dot plays a peculiar role in JavaScript. *For now*, we treat `Math.PI` *as if it were an identifier*.

New functions

Some we will cover today, more on Friday

Types in The Source

- Numbers: 1, -5.6, 0.5e-157
- Boolean values: `true`, `false`
- Functions: `function(x) { return x + 1; }`
- Strings: `"this is a string"`

Types in The Source

- Numbers: 1, -5.6, 0.5e-157
- Boolean values: `true`, `false`
- Functions: `function(x) { return x + 1; }`
- Strings: `"this is a string"`
- Today: ***empty list*** and ***pairs***

Equality

We allow `===` and `!==` on the following types:

- Boolean values — super easy
- Strings — straightforward
- Numbers — watch out! `3/5 === 2/5 + 1/5`
- Functions — ***All function values are created different.***

Highest Denomination

```
function highest_denom(kinds) {
    if      (kinds === 0) { return 0;          }
    else if (kinds === 1) { return 5;          }
    else if (kinds === 2) { return 10;         }
    else if (kinds === 3) { return 20;         }
    else if (kinds === 4) { return 50;         }
    else if (kinds === 5) { return 100;        }
    else                  { display('error'); }
}
```

Highest Denomination

```
function highest_denom(kinds) {
    if      (kinds === 0) { return 0;          }
    else if (kinds === 1) { return 5;          }
    else if (kinds === 2) { return 10;         }
    else if (kinds === 3) { return 20;         }
    else if (kinds === 4) { return 50;         }
    else if (kinds === 5) { return 100;        }
    else                  { display('error'); }
}
```

`highest_denom` stores six values and retrieves them when applied to the proper index.

Data Structure Example

`highest_denom` is a *data structure*.

- *Create* the data structure: Using function definition

  ```
  function highest_denom(kinds) {...}
  ```

- *Access* the data structure: Using function application

  ```
  ... highest_denom(3) ...
  ```

Data Structures in Math

- They are *everywhere*: tuples, sets, multisets, etc.

Data Structures in Math

- They are *everywhere*: tuples, sets, multisets, etc.
- What is the simplest data structure possible?

Data Structures in Math

- They are *everywhere*: tuples, sets, multisets, etc.
- What is the simplest data structure possible?
- A ***pair***:
  - Created in math using tuple notation, e.g. $(4, 3)$
  - Accessed in math using a pattern: Let p be (x, y), for some x and y ... (and now use x and y)

Pairs in The Source

```
function coordinates_4_3(dimension) {
    if (dimension === "x") {
        return 4;
    } else if (dimension === "y") {
        return 3;
    } else {
        return "error";
    }
}

/* Access: */ ... coordinates_4_3("y") ...
```

Works, but is it good?

Works, but is it good?

That’s quite good for access, but pretty bad for creating the data structure.

Works, but is it good?

That’s quite good for access, but pretty bad for creating the data structure.

```
function coordinates_4_3(dimension) {
    if (dimension === "x") {
        return 4;
    } else {
        return 3;
    }
}
```

Idea: Abstraction (what else?)

```
function make_coordinates(x, y) {
    return function (dimension) {
               if (dimension === "x") {
                   return x;
               } else {
                   return y;
               }
           };
}

var my_point = make_coordinates(4, 3); // create

... my_point("x") ...                  // access
```

Some Renaming

```
function pair(x, y) {
    return function (dimension) {
               if (dimension === "first") {
                   return x;
               } else {
                   return y;
               }
           };
}
function first(p) { return p("first"); }
function second(p) { return p("second"); }
```

Alternative Implementation

```
// make_one_out_of_two from Recitations
function pair(x, y) {
    return function (select) {
               return select(x, y);
           };
}
function first(p) {
    return p(function(x, y) { return x; });
}
function second(p) {
    return p(function(x, y) { return y; });
}
```

Checking `is_pair`: First Attempt

```
var tag = function(){ ; }; // created different
function pair(x, y) {
    return function (select) {
               return select(x, y, tag); };
}
function first(p) {
    return p(function(x, y, z) { return x; });
}
function second(p) {
    return p(function(x, y, z) { return y; });
}
function is_pair(p) {
    return p(function(x, y, z) { return z === tag; });
}
```

Checking `is_pair`: Second Attempt

```
function pair_lib(fun_name) {
  var tag = function() { return true; };
  function pair(x, y) { ...tag... }
  function first(p) { ... }
  function second(p) { ... }
  function is_pair(p) { ...tag... }
  return (fun_name === "pair") ? pair
         : (fun_name === "first") ? first
           : (fun_name === "second") ? second
             : (fun_name === "is_pair") ? is_pair
               : error("illegal access");
}
```

How to use `pair_lib`?

```
var pair = pair_lib("pair");
var first = pair_lib("first");
var second = pair_lib("second");
var is_pair = pair_lib("is_pair");
...
var my_pair = pair(1, 2);
is_pair(my_pair);  // returns true or false?
```

How to use `pair_lib`?

```
var pair = pair_lib("pair");
var first = pair_lib("first");
var second = pair_lib("second");
var is_pair = pair_lib("is_pair");
...
var my_pair = pair(1, 2);
is_pair(my_pair);  // returns true or false?
```

The flaw

Every call of `pair_lib` creates its own `tag`. The functions `pair` and `is_pair` become *incompatible*.

The Correct Version

```
function make_pair_lib() {
    var tag = function() { return true; };
    function pair(x, y) { ... }
    function first(p) { ... }
    function second(p) { ... }
    function is_pair(p) { ... }

    return function(fn) {
        return (fn === "pair") ? pair
               : (fn === "first") ? first
                 : (fn === "second") ? second
                   : (fn === "is_pair") ? is_pair
                     : error("illegal access");
    };
}
```

How to use it?

```
var pair_lib = make_pair_lib();
var pair = pair_lib("pair");
var first = pair_lib("first");
var second = pair_lib("second");
var is_pair = pair_lib("is_pair");
...
var my_pair = pair(1, 2);
is_pair(my_pair);  // returns true
```

Case Study: Rational Numbers

What is a rational number?

Case Study: Rational Numbers

What is a rational number?

A pair, consisting of a denominator and a numerator

Case Study: Rational Numbers

What is a rational number?

A pair, consisting of a denominator and a numerator

```
function make_rat(n, d) {
    return pair(n, d);
}
function numer(x) {
    return first(x);
}
function denom(x) {
    return second(x);
}
```

Addition of Rational Numbers

```
function add_rat(x, y) {
    return make_rat(numer(x) * denom(y) +
                    numer(y) * denom(x),
                    denom(x) * denom(y));
}
```

Subtraction of Rational Numbers

```
function sub_rat(x, y) {
    return make_rat(numer(x) * denom(y) -
                    numer(y) * denom(x),
                    denom(x) * denom(y));
}
```

Multiplication and Division

```
function mul_rat(x, y) {
    return make_rat(numer(x) * numer(y),
                    denom(x) * denom(y));
}
function div_rat(x, y) {
    return make_rat(numer(x) * denom(y),
                    denom(x) * numer(y));
}
```

Equality: First Attempt

```
function equal_rat(x, y) {
    return numer(x) === numer(y) &&
           denom(x) === denom(y);
}
```

Equality: Second Attempt

```
function equal_rat(x, y) {
    return numer(x) * denom(y) ===
           numer(y) * denom(x);
}
```

Printing

```
function rat_to_string(x) {
    return numer(x) + " / " + denom(x);
}
```

Playing with Rational Numbers

```
var one_half  = make_rat(1, 2);

var one_third = make_rat(1, 3);

rat_to_string(add_rat(one_half, one_third));

rat_to_string(mul_rat(one_half, one_third));

rat_to_string(add_rat(one_third, one_third));
```

Playing with Rational Numbers

```
var one_half  = make_rat(1, 2);

var one_third = make_rat(1, 3);

rat_to_string(add_rat(one_half, one_third));

rat_to_string(mul_rat(one_half, one_third));

rat_to_string(add_rat(one_third, one_third));
```

Returns `"6 / 9"`!

First Approach: Reduce When Making a Rational

```
function make_rat(n, d) {
    var g = gcd(n, d);
    return pair(n / g, d / g);
}
```

Second Approach: Reduce When Accessing

```
function make_rat(n, d) {
    return pair(n, d);
}
function numer(x) {
    var g = gcd(first(x), second(x));
    return first(x) / g;
}
function denom(x) {
    var g = gcd(first(x), second(x));
    return second(x) / g;
}
```

Third Approach: Reduce When Displaying

```
function make_rat(n, d) { return pair(n, d); }
function numer(x) { return first(x); }
function denom(x) { return second(x); }

function rat_to_string(x) {
    var g = gcd(numer(x), denom(x));
    return (numer(x) / g) + " / " + (denom(x) / g);
}
```

Summary of Case Study on Rationals

- Pairs can be used to represent rational numbers
- Operations are implemented using constructor and accessor functions
- A library hides the internal representation of the data
- Implementation details remain invisible to the user of the library

Making Lists with Pairs: Motivation

```
var highest_denom = pair( pair(50, 20),
                          pair(10, 5) );
```

Making Lists with Pairs: Motivation

```
var highest_denom = pair( pair(50, 20),
                          pair(10, 5) );
```

What if we want the values except the first?

Making Lists with Pairs: Motivation

```
var highest_denom = pair( pair(50, 20),
                          pair(10, 5) );
```

What if we want the values except the first?

```
var highest_denom_2 =
    pair( second(first(highest_denom),
          second(highest_denom) );
```

Making Lists with Pairs: Motivation

```
var highest_denom = pair( pair(50, 20),
                          pair(10, 5) );
```

What if we want the values except the first?

```
var highest_denom_2 =
    pair( second(first(highest_denom),
          second(highest_denom) );

// "equal" to pair( 20, pair(10, 5) );
```

Problem: Removing Next Value Works Differently!

```
var highest_denom = pair( pair(50, 20),
                          pair(10, 5) );

var highest_denom_2 =
    pair( second(first(highest_denom)),
          second(highest_denom) );

var highest_denom_3 =
    second(highest_denom_2);
```

Idea: Introduce Discipline

Principle

Make sure that `first(p)` always has the data, and `second(p)` always has the remaining elements.

Idea: Introduce Discipline

Principle

Make sure that `first(p)` always has the data, and `second(p)` always has the remaining elements.

Example

```
var highest_denom =
    pair(50, pair(20, pair(10, 5)));
```

Special Case

What if we are at the end?

Special Case

What if we are at the end?

```
var highest_denom = pair(10, 5);
```

Special Case

What if we are at the end?

```
var highest_denom = pair(10, 5);
```

Now the program

```
var rest = second(highest_denom);
```

Special Case

What if we are at the end?

```
var highest_denom = pair(10, 5);
```

Now the program

```
var rest = second(highest_denom);
```

gives us a value, not the remaining elements!

Idea: Introduce a *Base Case*

How to represent the empty list?

Idea: Introduce a *Base Case*

How to represent the empty list?

It doesn't really matter!

Idea: Introduce a *Base Case*

How to represent the empty list?

It doesn't really matter!

One way

Recall how *tags* were represented in the pair library

Representing the Empty List

```
function empty_list() { return true; }
```

highest_denom using the Empty List

```
var highest_denom =
pair(50, pair(20, pair(10, pair(5, empty_list))));
```

The Same “In Style”

```
var highest_denom =
    pair(50,
         pair(20,
              pair(10,
                   pair(5,
                        empty_list))));
```

Lists in The Source: Naming

- `first` renamed to `head`
- `second` renamed to `tail`

- Special symbol for `empty_list`: the empty box symbol `[]`

List Discipline in The Source

Definition

A list is either empty `[]` or a pair whose tail is a list.

highest_denom using The Source

```
var highest_denom = pair(50,
                         pair(20,
                              pair(10,
                                   pair(5, [])))));

... head(highest_denom)...

... head(tail(tail(highest_denom)))...
```

Source Week 5's List Library

- `pair(x, y)`: returns pair made of `x` and `y`
- `is_pair(p)`: returns `true` iff `p` is a pair
- `[]`: represents the empty list
- `is_empty_list(xs)`: returns `true` iff `xs` is the empty list
- `head(xs)`: returns the head (first component) of list `xs`
- `tail(xs)`: returns the tail (second component) of list `xs`
- `list(x1,...,xn)`: returns list whose first element is `x1`, second element is `x2`, etc. and last element is `xn`

Printing Pairs in Source IDE Interpreter

- `pair(x, y)` is printed as `[`*x*`,` *y*`]`
- The empty list is printed as `[]`

Example

```
pair(1, pair(2, pair(3, [])));
```

is printed as

```
[1, [2, [3, []]]]
```

Error Reporting

The functions that query the structure of lists have *expectations* for their arguments:

- `head(xs)`: expects a pair as `xs`
- `tail(xs)`: expects a pair as `xs`

Otherwise, a nice error message gets printed

Computing the Length of a List

Definition

The length of the empty list is 0, and the length of a non-empty list is one more than the length of its tail.

```
function length(xs) {
    if (is_empty_list(xs)) {
        return 0;
    } else {
        return 1 + length(tail(xs));
    }
}
```

Can we do this Iteratively?

Can we do this Iteratively?

```
function length_iter(xs) {
    function len(ys, counted_so_far) {
        if (is_empty_list(ys)) {
            return counted_so_far;
        } else {
            return len(tail(ys),
                       counted_so_far + 1);
        }
    }
    return len(xs, 0);
}
```

Summary

Summary

- The Source (equality)

Summary

- The Source (equality)
- Data structures: an old hat, really

Summary

- The Source (equality)
- Data structures: an old hat, really
- Case study: Rational numbers

Summary

- The Source (equality)
- Data structures: an old hat, really
- Case study: Rational numbers
- Lists in Source Week 5