L5: Introduction to Data Abstraction

SWS3012: Structure and Interpretation of Computer Programs

Martin Henz (adapted from slides by Low Kok Lim)

July 8, 2023

Outline

- Data abstraction (2.1)
- Case study: rational numbers (2.1.1)
- Making lists with pairs (2.2.1)

Where Are We Now?

- **Module overview**
  - Unit 1 — Functional abstraction: SICP Chapter 1
  - **Unit 2 — Data abstraction: SICP Chapter 2**
  - Unit 3 — State: SICP Chapter 3
  - Unit 4 — Beyond: SICP Chapters 4 & 5
- **New language: Source §2**
  - See language documentation at https://docs.sourceacademy.org/source_2/
  - Source Academy Playground is using **Source §2** from now on

Outline

- Data abstraction (2.1)
- Case study: rational numbers (2.1.1)
- Making lists with pairs (2.2.1)

Types of Values in Source

- **Numbers:** `1`, `-5.6`, `0.5e-157`
- **Boolean values:** `true`, `false`
- **Strings:** `"this is a string"`
- **Functions:** `x => x + 1`
- **Some others** (imported from modules): `heart`, `make_point(0.5, 0.25)`
- **Today:** ***pairs*** and ***empty list***

Data Structures in Mathematics

- They are **everywhere**: tuples, sets, matrices, etc.
- What is the simplest data structure possible?
- A ***pair***
  - **Constructed** in math using tuple notation, e.g. (0.5, 0.25)
  - **Selected** in math using a pattern:
    - Let *p* be (*x*, *y*), for some *x* and *y* ... (and now use *x* and *y*)

Pairs so far — Points in Curve Missions

```
// Construct:
const p = make_point(0.5, 0.25);

// Select:
const x = x_of(p);
const y = y_of(p);
```

We can Define the Point Abstraction (2.1.3)

```
function make_point(x, y) {
    return component =>
        component === 0 ? x : y;
}

function x_of(p) {
    return p(0);
}

function y_of(p) {
    return p(1);
}
```

Show in Playground

Using More Generic Names

```
function pair(x, y) {
    return component =>
        component === 0 ? x : y;
}

function head(p) {
    return p(0);
}

function tail(p) {
    return p(1);
}
```

Show in Playground

Another way to define pair, head, tail

```
const pair = (x, y) => f => f(x, y);

const head = p => p((x, y) => x);

const tail = p => p((x, y) => y);
```

Show in Playground

Outline

- Data abstraction (2.1)
- Case study: rational numbers (2.1.1)
- Making lists with pairs (2.2.1)

Case Study: Rational Numbers (2.1.1)

- What is a ***rational number***?
  - A **pair**, consisting of a **denominator** and a **numerator**
- In Source:

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

Addition and Subtraction of Rational Numbers

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

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

Multiplication and Division of Rational Numbers

```
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 of Rational Numbers

- **First attempt:**

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

- **Second attempt:**

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

Printing Rational Numbers

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

Playing with Rational Numbers

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

const one_third = make_rat(1, 3);

rat_to_string(add_rat(one_half, one_third));
➔ "5 / 6"

rat_to_string(mul_rat(one_half, one_third));
➔ "1 / 6"

rat_to_string(add_rat(one_third, one_third));
➔ "6 / 9"
```

[Show in Playground]

Making Reduced Rational Numbers

```
// Compute the greatest common divisor (GCD) of
// two numbers using Euclid’s algorithm
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}

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

Show in Playground

Playing with Rational Numbers Again

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

const one_third = make_rat(1, 3);

rat_to_string(add_rat(one_half, one_third));
→ "5 / 6"

rat_to_string(mul_rat(one_half, one_third));
→ "1 / 6"

rat_to_string(add_rat(one_third, one_third));
→ "2 / 3"
```

Show in Playground

Summary of Case Study on Rational Numbers

- **Pairs** can be used to represent **rational numbers**
- Operations are implemented using **constructor** and **selector** functions
- A library **hides** the **internal representation** of the data
  - Implementation details remain invisible to the user of the library
  - Provides a higher-level abstraction

Outline

- Data abstraction (2.1)
- Case study: rational numbers (2.1.1)
- Making lists with pairs (2.2.1)

Making Lists with Pairs: Motivation

- Want to put the coin denominations 100, 50, 20, 10, 5 in a “list” constructed from pairs

```
const first_denomination =
    pair(...100...50...20...10...5...);
```

- Many possible ways, examples:
  - `pair(pair(100, 50), pair(20, pair(10, 5)))`
  - `pair(100, pair(pair(50, 20), pair(10, 5)))`
  - `pair(pair(pair(100, 50), pair(20, 10)), 5)`
  - `pair(100, pair(50, pair(20, pair(10, 5))))`

- Different ways of representations require different ways of retrieval

Idea: Introduce Some *Discipline*

- **Principle**
  - Make sure that `head(p)` always has the **data**, and `tail(p)` always has the **remaining elements**

- **Example:**

```
const denoms =
    pair(100, pair(50, pair(20, pair(10, 5))));

head(denoms) ➔ 100
tail(denoms) ➔ pair(50, pair(20, pair(10, 5)))
```

Special Case

- What if

```
const denoms = pair(10, 5);
```

- Then the program

```
const rest = tail(denoms);
```

gives us a value 5, not the remaining ***list of elements***

Idea: Introduce a *Base Case*

- How to represent the **empty list**?
  - It doesn't really matter!

- In Source, we use the special value `null` to represent the empty list

first_denomination using an Empty List

```
const first_denomination =
    pair(100,
        pair(50,
            pair(20,
                pair(10,
                    pair(5,
                        null)))));
```

List Discipline in Source

- **Definition:**

  A ***list*** is either `null` or a **pair whose tail is a list**

- **Examples:**

```
const my_list = null;

const your_list = pair(8, null);

const first_denomination =
    pair(100,
         pair(50,
              pair(20, pair(10, pair(5, null)))));
```

Retrieving Data from a List

- **Example:**

```
const denoms =
    pair(100,
         pair(50,
              pair(20, pair(10, pair(5, null)))));
```

```
head(denoms) → 100
head(tail(denoms)) → 50
head(tail(tail(denoms))) → 20
head(tail(tail(tail(denoms)))) → 10
head(tail(tail(tail(tail(denoms))))) → 5
tail(tail(tail(tail(tail(denoms))))) → null
```

LISTS Functions in Source §2 (link to doc)

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

Variadic Function `list`

- It can be tedious to write a list manually with `pair` and `null`

```
// Example:
const first_denomination =
    pair(100,
        pair(50, pair(20, pair(10, pair(5, null)))));
```

- Using the `list` function

```
// Example:
const first_denomination = list(100, 50, 20, 10, 5);
// the same, but shorter
```

Box Notation

- In ***box notation***
  - `pair(x, y)` is printed as `[x, y]`
  - Empty lists are printed as `null`

- **Example:**

```
pair(1, pair(2, pair(3, null)));
```

is printed as

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

Show in Playground

List Notation

- ***List notation***
  - Same as box notation, but any sub-structure that is a list is nicely formatted and printed as `list(...)`
  - Use predeclared function `display_list(x)` to show `x` in list notation

- **Example:**

```
display_list(
    pair(pair(pair(7, 8), pair(1, pair(2, null))),
         6));
```

prints

```
[list([7, 8], 1, 2), 6]
```

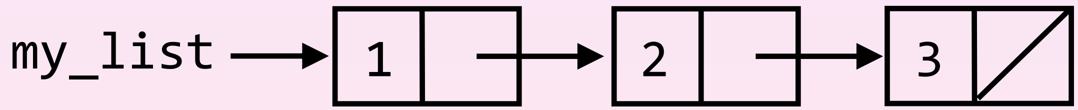
Show in Playground

Box-and-Pointer Diagrams

- **Box-and-pointer** diagrams are graphical representations of data structures made of pairs

- **Example:**

```
const my_list = pair(1, pair(2, pair(3, null)));
```



- **Data Visualizer** tool generates such diagrams in Playground
  - Use `draw_data` pre-declared function
  - **Example:** `draw_data(pair(1, pair(2, pair(3, null))));`

Show in Playground

Error Reporting

- The functions that query the structure of lists have **expectations** for their **arguments**:
  - `head(xs)`: expects `xs` is a pair
  - `tail(xs)`: expects `xs` is a pair
- Otherwise, a nice error message gets printed

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

- **Examples:**
  - Length of `null` is 0
  - Length of `pair(10, null)` is 1
  - Length of `pair(10, pair(20, pair(30, null)))` is 3

Computing the Length of a List

```
function length(xs) {
    return is_null(xs)
           ? 0
           : 1 + length(tail(xs));
}
```

Show in Playground

- Does it give rise to a **recursive** or **iterative** process?

Computing the Length of a List

- **Iterative version:**

```
function length_iter(xs) {

    function len(ys, counted_so_far) {
        return is_null(ys)
            ? counted_so_far
            : len(tail(ys), counted_so_far + 1);
    }

    return len(xs, 0);
}
```

Show in Playground

Summary

- **Data structures** are everywhere
- **Pairs**
- Case study: **rational numbers**
- **List discipline**
- **LISTS functions** in Source §2
- **Box notation**, **List notation**, and **Box-and-pointer diagrams**