Data Abstraction

Jia Xiaodong

September 6, 2021

Data Structures

For example, I want to sort a bunch of numbers:

```
// returns them in sorted order
function sort(a,b,c,d...) { ??? }
```

Better?

```
function sort(list) { ... return list; }
```


So far we have abstracted functions. However even then it might not be enough. A good example would be the rational number abstraction introduced in the lectures. That would improve productivity greatly compared to trying to manage a numerator and denominator on your own.

Here I also want to suggest another use case. Say we want to sort a few numbers into ascending order. If we wrote a function that could only take in numbers, that would be pretty difficult to write (though not impossible). It would all be so much simpler if there was a notion of a list, very much like a list of numbers on a sheet of paper. Then we would do our manipulations on that list and return the updated list.

What is data?

> *...we can think of data as defined by some collection of selectors and constructors, together with specified conditions that these functions must fulfil in order to be a valid representation.* — *SICP* §2.1.3

The point of this statement is that we need only be concerned with the outward appearance, or structure of our data, and not how it is made up of.

Church encoding *

Ex. 2.6

Give a definition for `plus`, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)
```

```
const two = succ(one);
f => x => f(one(f)(x))
f => x => f(((x => f(x))(x))
f => x => f(f(x))
```

```
const three = succ(two);
f => x => f(two(f)(x))
f => x => f((x => ffx)(x))
f => x => fffx
```

```
const four = succ(three);
f => x => f(three(f)(x))
f => x => f((x => fffx)(x))
f => x => ffffx
```


This is a good illustration of what the quote from the textbook means.

With 0 and the successor function the natural numbers can be defined. Let us first see what the numbers look like in this scheme. Applying the substitution model a few times we see that numbers in this scheme is defined as the number of self compositions of a function. (There is some funny business here regarding how we use "number of something" to define numbers, let us just ignore that.)

Church encoding *

Cont.

```
const one = f => x => fx
const three = f => x => fffx
const four = f => x => ffffx

three(f)(x) === fffx
one(f)(fffx) === f fffx

four(f)(x) === one(f)(three(f)(x))

function plus(a, b) {
    return f => x => a(f)(b(f)(x));
}
```

Now we can try coming up with the `plus` function. Look at a few examples. Say we want to do $1 + 3 = 4$, i.e. we want `plus(one, three) ~ four` (I use symbol ~ to mean congruency, because `===` would not work here. Can you explain why `===` doesn't work?).

What we want to accomplish is quite simple, if somehow we made the `x` in `one` to become `fffx` of `three`, we would get what we want. The way to "expose" `fffx` is by basically calling the numeral twice similar to what is done is `succ`.

Church encoding *

Cont.

Give a definition for pred.

```
const one = f => x => f(x);
const two = f => x => f(f(x));

function contain(n) { return p => p(n); }
function extract(c) { return c(u => u); }       // c is a container

function inc(c) { return h => h(c(f)); }
const init = u => x
inc(init) = h => h(x)           // contain(x)
inc(inc(init)) = i => i(f(x))   // contain(f(x))

function pred(n) { return f => x => extract(n(inc)(init); }

function pred(n) { return f => x =>
    extract(n(c => h => h(c(f)))(init); }
```


Let us take a look at some of our values again. What we want to achieve is to strip off an application of `f`. We can do so with the identity function `x=>x`. However we must apply this only at the right place. To control this, we wrap our numbers in a container (note the similarity to `pair`). Extract releases the container.

Take a look at this `inc` function. It takes a contained value and increments the value in the container. `init` is a base value. Now perhaps we notice something. Applying `inc` once gets us to `x` which is essentially what is inside `zero`. And so on. Hence what we do is to call `inc n` times on `init`, extract out the value, and we are done.

Note that in this slide I have used names like `f` as though they are fixed globally. This will not work, especially for `inc`. Please make sure you are clear which names are bound and which are not, especially here where there are a so many different names floating around.

Church encoding *

Quest

New definitions. Implement succ and pred.

```
const zero = f => x => x;
const one = f => x => f(zero, () => zero(f)(x));
const two = f => x => f(one, () => one(f)(x));
```

```
function succ(n) { f => x => f(n, () => x); }

function pred(n) { f => x => n((m, n) => m)(zero); }

function plus(a, b) { a((m, n) => succ(n())) (b); }
```


The last 2 quest questions revolves around speeding up the `pred` operation through feeding the next numeral information about the predecessor as well.

Notice for `pred` we return the first item and not the second. Do you know why?

Try `plus` out for yourself and confirm that it works.

Pair

A pair is a collection of two items. We assign one to the *head*, and the other to the *tail* of the pair.

Ex. 2.4 *

Possible implementation:

```
function pair(x,y) { return f => f(x, y); }
function head(p) { return p((x, y) => x); }
function tail(p) { return p((x, y) => y); }
```

The simplest structure you can come up is perhaps grouping two things together.

As mentioned earlier, it does not matter how this is done. Here is a possible implementation using functions. It does not look like what you might expect from "a pair of two things". Another implementation may be more common-sense, using *arrays* in native JavaScript (not available in the current Source chapter you are using).

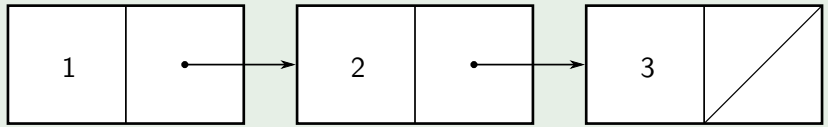
In any case, these things can be changed at the whim of the implementer and if you base your own programs on the assumption that it's going to be implemented in a certain way (breaking abstraction), your program will fail once this assumption fails to hold true.

List

`null` is a (empty) list. A list is a pair whose tail is a list.

Example

`list(1, 2, 3) === pair(1, pair(2, pair(3, null)))` [1]



Predeclared functions:

- LISTS documentation
- S2 Language Spec

[1] This actually evaluates to `false`. What I mean by `===` here, for the lack of a better way to write it, is that they mean the same thing.

There is no limit on what you pair. A sequence can be made using a chain of pairs. We recursively define a list to be a pair of something with another list.

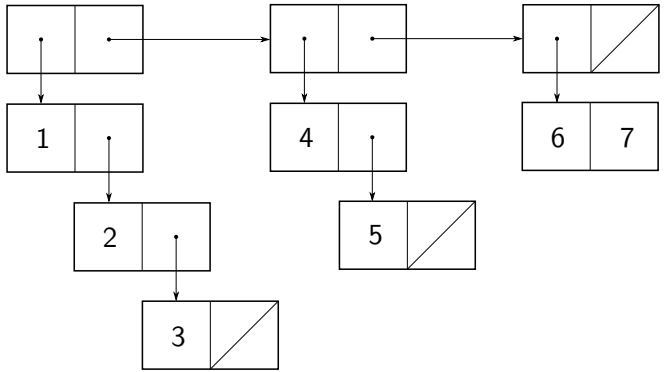
Graphically we use "box and pointer" diagrams to illustrate how these look like.

This definition also means that a list is just a big pair and we can continue using the pair operations on lists. We will be seeing plenty of examples of what we can do with lists.

For a reference of all the predeclared list functions and what they do, refer to the two links provided.

S5 Q1

Draw box and pointer diagram and give the printed representation for `list(list(1, 2, list(3)), list(4, 5), pair(6, 7));`



One tip when trying to parse these expressions: draw whatever you can first.

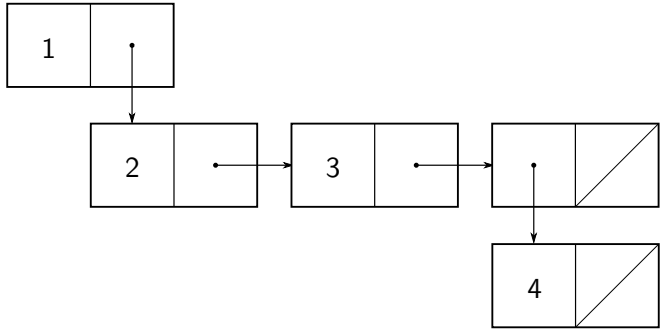
The only thing you need to remember is the difference between a pair and a list. A pair puts 2 things in both of its boxes. A list only has 1 data element in its left box, the right box points to the rest of the list.

As to when you write data elements in, and when you draw an arrow out, this is (to my knowledge) not really important. The guideline is for simple values like `null`, numbers, strings, you write them in, and for objects like lists, functions, you draw an arrow out.

S5 Q1

Cont.

Draw box and pointer diagram and give the printed representation for `pair(1, list(2, 3, pair(4, null)));`

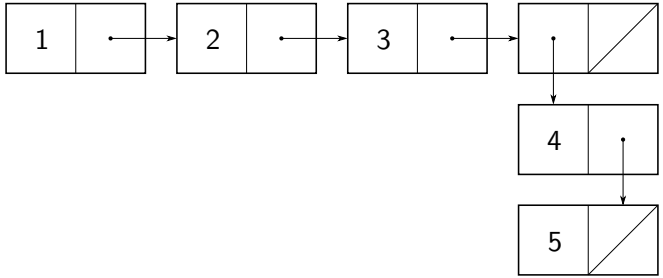


`pair(4, null)` is the same as `list(4)`. Also note that `list(1, list(2))` is not `list(1, 2)`. Also `pair(1, list(2))` is the same as `list(1, 2)`.

S5 Q1

Cont.

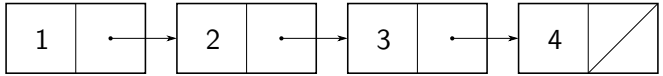
Draw box and pointer diagram and give the printed representation for `pair(1, pair(2, list(3, list(4, 5))));`



S5 Q2

```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```

Evaluate `reverse(list(1, 2, 3, 4));`.

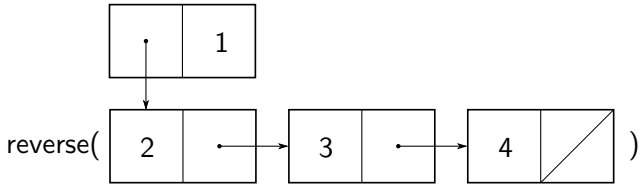


This is a wrong implementation for reverse, and we will see why by evaluating it for an example list. Start with the list. This list is not null, so we evaluate the alternative.

S5 Q2

Cont.

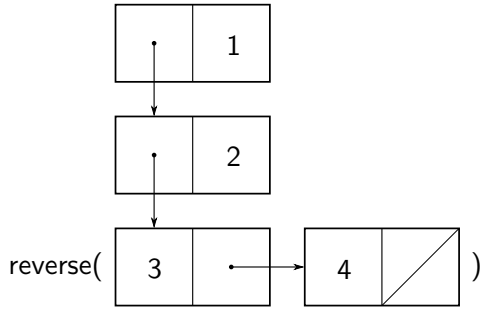
```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```



S5 Q2

Cont.

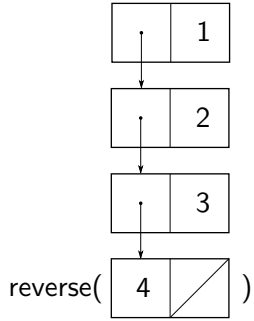
```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```



S5 Q2

Cont.

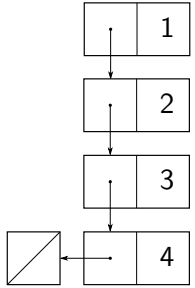
```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```



S5 Q2

Cont.

```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```



We will just do the last two steps in one go. We end up with a list that is not reversed, but with pairs that have been reversed, and since it does not conform with our definition of a list, the output is incorrect.

Now, do you know how to implement a reverse function that works?

S5 Q3

Write expressions using `lst`, `head`, `tail` that will return `1` with `lst = list(7, list(6, 5, 4), 3, list(2, 1));`

- `head tail tail tail` gets us to `list(2, 1)`.
- Then, `head tail` gets us 1.

```
head(tail(head(tail(tail(tail(lst))))));
```


This is fairly straightforward. It's like navigating a maze. Only make sure to match the brackets carefully and know which term belongs where.

Also note that `tail`ing any list gives a list by the definition of a list.

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return `1` with `lst = list(list(7), list(6, 5, 4), list(3, 2), 1);`

```
head(tail(tail(tail(lst))))
```

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return `1` with

```
lst = list(7, list(6), list(5, list(4)),
       list(3, list(2, list(1))));
```

- `head tail tail tail` gets us to `list(3, list(...))`.
- Then `head tail` gets us to `list(2, list(1))`.
- Then `head tail` gets us to `list(1)`.
- Then `head` gives us 1.

```
head(head(tail(head(tail(head(tail(tail(tail(lst)))))))))
```

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return `1` with

```
lst = list(7,
       list(list(6, 5), list(4), 3, 2), list(list(1)));
```

- `head tail tail` gets us to `list(list(1))`.
- Then `head head` gets us 1.

```
head(head(head(tail(tail(lst)))))
```

S5 IC-Q1

Write function `every_second` that takes in a list and returns a list containing every other element, starting from the first element.

```
function every_second(lst) {
    function h(res, n, max) {
        return n >= max
            ? res
            : h(pair(list_ref(lst, n), res), n + 2, max);
    }
    return h(null, 1, length(lst));
}
```

This is actually very easy using the hint regarding `list_ref`. Simply count up from 0 to `length(lst) - 1`, appending to a temporary list that forms our result.

However, this is extremely inefficient. Look up the time complexity of `list_ref`. What is the overall time complexity of this implementation of `every_second`? For a problem like this, we should be striving for $\Theta(n)$.

S5 IC-Q1

Cont.

Write function every_second that takes in a list and returns a list containing every other element, starting from the first element.

```
function every_second(lst) {
    return is_null(lst) || is_null(tail(lst))
        ? null
        : pair(head(tail(lst)), every_second(tail(tail(lst))));
    }
}
```


We do not need `list_ref` at all. `list_ref` has to run through the list from the start each time we call it. We can simply go through the list one by one, eliminating the repetitive work `list_ref` does.

The base case occurs when it is a list of 1 element, or no elements. Then there is no second element and we return `null`.

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

```
function every_second_odd(lst) ...
function every_second_even(lst) ...
function sum(lst) ...
```


One way to do it is to use the results from the previous question and sum over the even and the odd lists. However you should learn to feel a bit odd about doing things this way. For one, it is wasted work since you traverse the list twice (yes it is still linear time, but if you do double the work, and run twice as slow, then you are still very slow). It also wastes space by creating new lists. Furthermore, `every_second_*` operates by skipping over elements. Why don't we compute a sum while doing the skipping?

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

```
function sum(lst){
    function s(e, o, lst, iseven) {
        return is_null(lst)
            ? list(e, o)
            : iseven
                ? s(e + head(lst), o, tail(lst), false)
                : s(e, o + head(lst), tail(lst), true);
    }
    return s(0, 0, lst, true);
}
```


We can ignore the list part first. The requirement of the two sums being in a list might as well not be a requirement because it is easy to put two things in a list. So let us make a helper function keeping track of running sums of even and odd items. We also keep a flag `iseven` that tells us are we at an even-ranked element or not.

In this case the base case is fairly simple, if we hit an empty list then we return the running sums.

Otherwise, if we are at an even item, we add to the even sum, otherwise, we add to the odd sum, and move on to the rest of the list, flipping the flag so we track the parity correctly.

Q6

Lexicographic order

Write a function `lexico(xs, ys)` that takes in lists of characters `xs` and `ys` and returns `true` iff `xs > ys` lexicographically.

```
function lexico(xs, ys) {
    if (is_null(xs)) {
        return false;
    } else if (is_null(ys)) {
        return true;
    } else if (head(xs) === head(ys)) {
        return lexico(tail(xs), tail(ys));
    } else if (head(xs) > head(ys)) {
        return true;
    } else {
        return false;
    }
}
```


If `xs` ends prematurely then there is no chance it can be larger than `ys`. Similarly, if `ys` ends prematurely then it must be smaller than `xs`.

After we confirm that the strings are not empty, we check their `heads`. If they are the same then we will delegate the check to the rest of the string. Otherwise we can make a decision now and return the result.

Q7

Substring

Write a function `substr(xs, ys)` that takes in lists of characters `xs` and `ys` and returns `true` iff `ys` is a substring of `xs`.

```
function substr(xs, ys) {
    function trial(xs, ys) {
        if (is_null(xs)) { return is_null(ys); }
        else if (is_null(ys)) { return true; }
        else if (head(xs) === head(ys)) {
            return trial(tail(xs), tail(ys));
        } else { return false; }
    }
    function step(xs) {
        if (is_null(xs)) { return false; }
        else {
            return trial(xs, ys) || step(tail(xs));
        }
    }
    return step(xs);
}
```


This is a little trickier than first meets the eye. It is not enough to check the `head`s and then move to the `tail`s, for example for `"hhello"` and `"hell"`, this will result in checking `"hello"` and `"ell"` and so on which will be wrong. Furthermore, doing so might cause an error where `"hell"` is a substring of `"helllo"` since they are only in the right sequence but not contiguous.

The correct (and naive) way is to just run a test for every character, one by one. The test will go step through both the string and `ys` and confirm that they match. We return `true` if any one of the tests pass.