L5: List and Tree Processing

CS1101S: Programming Methodology

Martin Henz

September 8, 2021

Where are we?

Module overview

Where are we?

Module overview

- Unit 1—Functional abstraction: SICP Chapter 1

Where are we?

Module overview

- Unit 1—Functional abstraction: SICP Chapter 1
- Unit 2—Data abstraction: SICP Chapter 2

Where are we?

Module overview

- Unit 1—Functional abstraction: SICP Chapter 1
- Unit 2—Data abstraction: SICP Chapter 2
- Unit 3—State: SICP Chapter 3

Where are we?

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 Chapter 4

Unit 2

Lectures and Briefs

Unit 2

Lectures and Briefs

- Intro to data abstraction: L4

Unit 2

Lectures and Briefs

- Intro to data abstraction: L4
- List and Tree Processing: L5

Unit 2

Lectures and Briefs

- Intro to data abstraction: L4
- List and Tree Processing: L5
- Language Processing: B5

Unit 2

Lectures and Briefs

- Intro to data abstraction: L4
- List and Tree Processing: L5
- Language Processing: B5
- Symbolic Processing: L6

Unit 2

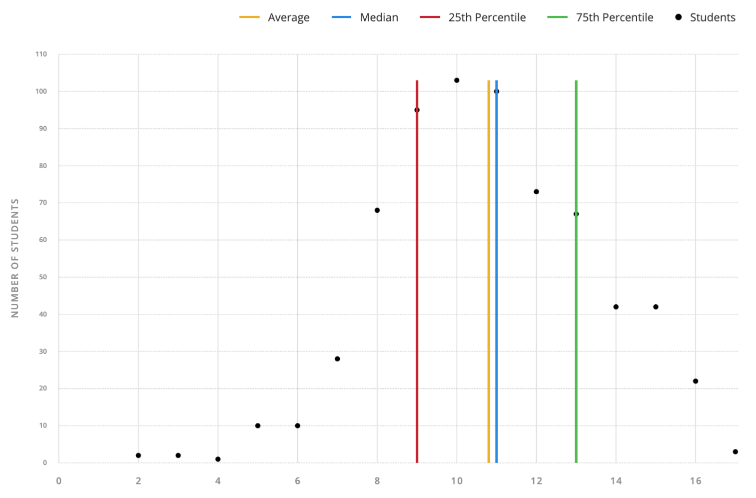
Lectures and Briefs

- Intro to data abstraction: L4
- List and Tree Processing: L5
- Language Processing: B5
- Symbolic Processing: L6
- Guest Lecture by Joel Low: B6

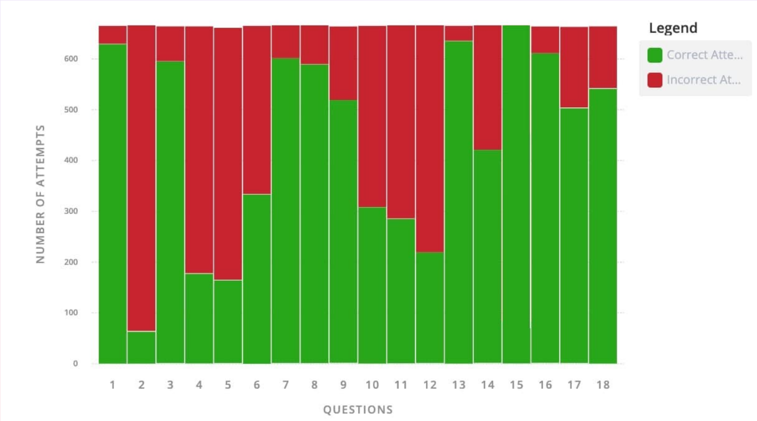
Some stock-taking

- Reading Assessment 1
- Unit 1 Survey

RA1: Results per student



RA1: Results per question



Reading Assessment 1: The hardest question

```
const x = 3;

function fun(x) {
    if (x % 2 === 0) {
        const z = 20;
    } else {
        const z = 30;
    }

    return x + y + z;
}

const y = 5;
const z = 10;
fun(x + y);
```

Unit 1 Survey

- Anonymous

Unit 1 Survey

- Anonymous
- 674 participants: 100%

Unit 1 Survey

- Anonymous
- 674 participants: 100%
- Candid feedback on module and staff

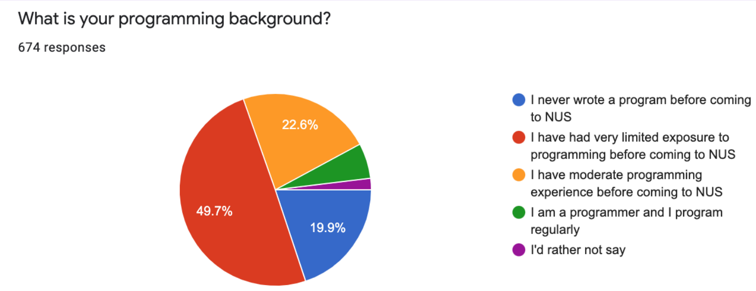
Unit 1 Survey

- Anonymous
- 674 participants: 100%
- Candid feedback on module and staff
- We hear you!

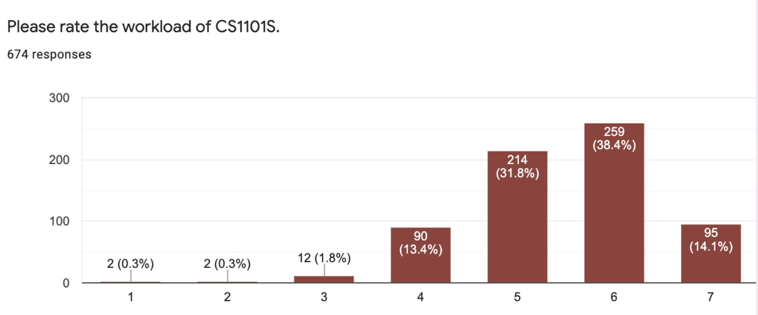
Unit 1 Survey

- Anonymous
- 674 participants: 100%
- Candid feedback on module and staff
- We hear you!
- Many thanks for participating!

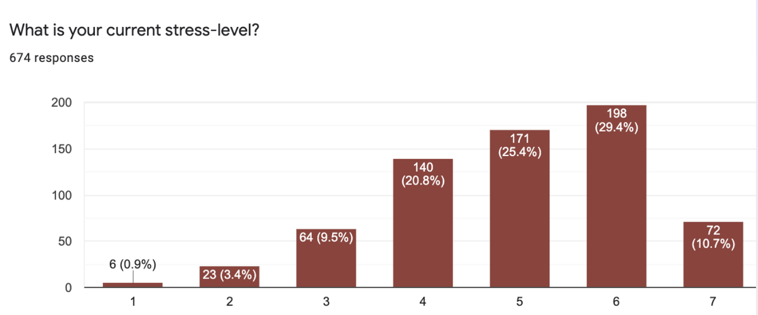
Unit 1 Survey: Programming Background



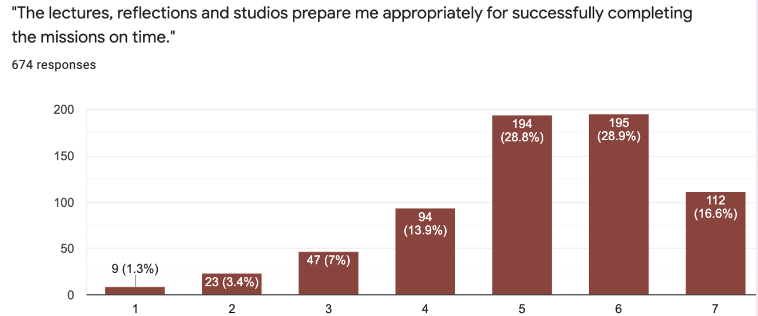
Unit 1 Survey: Workload



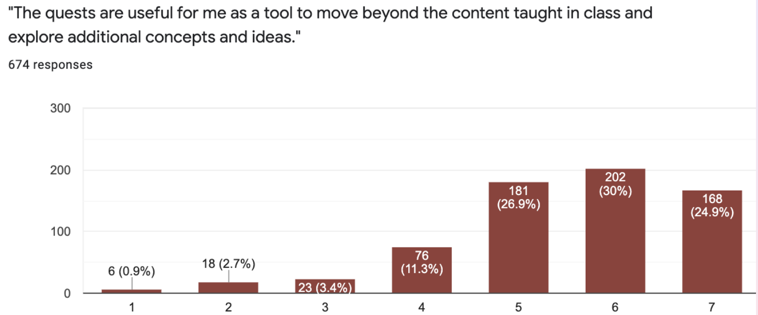
Unit 1 Survey: Stress Level



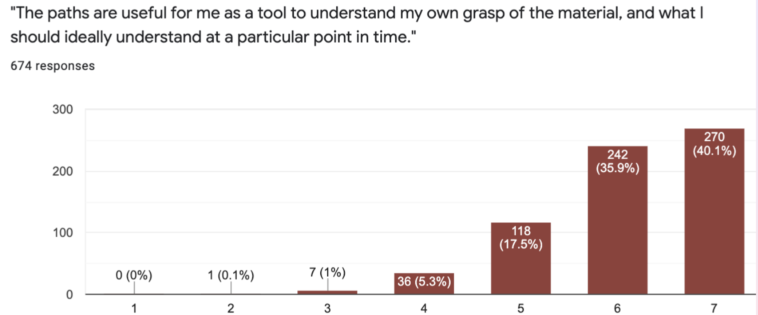
Unit 1 Survey: Preparation for Missions?



Unit 1 Survey: Quests



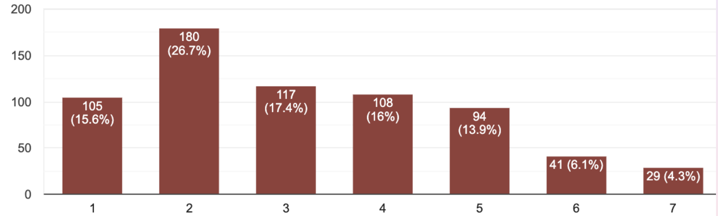
Unit 1 Survey: Paths



Unit 1 Survey: COVID

In your current view, how much are your studies in CS1101S affected by the COVID measures? (online-only lectures, reflections, social distancing etc)

674 responses



Some remarks

Some remarks

- No "curve"! Your individual learning counts!

Some remarks

- No “curve”! Your individual learning counts!
- It’s a marathon, not a sprint!

Some remarks

- No “curve”! Your individual learning counts!
- It’s a marathon, not a sprint!
- Make use of our resources!

Some remarks

- No "curve"! Your individual learning counts!
- It's a marathon, not a sprint!
- Make use of our resources!
- Learning attitude

Mastery Checks

Mastery Checks

- Oral tests

Mastery Checks

- Oral tests
- Conducted in 3-way meetings: 2 students, 1 Avenger, mostly of the same Studio

Mastery Checks

- Oral tests
- Conducted in 3-way meetings: 2 students, 1 Avenger, mostly of the same Studio
- Any time during the semester

Mastery Checks

- Oral tests
- Conducted in 3-way meetings: 2 students, 1 Avenger, mostly of the same Studio
- Any time during the semester
- Mastery check 1 (3 % of total grade)
  - scope,
  - higher-order function,
  - substitution model and iterative/recursive processes

Mastery Checks

- Oral tests
- Conducted in 3-way meetings: 2 students, 1 Avenger, mostly of the same Studio
- Any time during the semester
- Mastery check 1 (3 % of total grade)
  - scope,
  - higher-order function,
  - substitution model and iterative/recursive processes
- pass/fail

Mastery Checks

- Oral tests
- Conducted in 3-way meetings: 2 students, 1 Avenger, mostly of the same Studio
- Any time during the semester
- Mastery check 1 (3 % of total grade)
  - scope,
  - higher-order function,
  - substitution model and iterative/recursive processes
- pass/fail
- repetition allowed, also per topic

Mastery Checks: The Process

Mastery Checks: The Process

- Form pairs

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.
- Use proper English; illustrate your argument with examples.

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.
- Use proper English; illustrate your argument with examples.
- Be ready for using examples given by the Avenger.

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.
- Use proper English; illustrate your argument with examples.
- Be ready for using examples given by the Avenger.
- Be ready to answer questions on the topic posed by Avenger.

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.
- Use proper English; illustrate your argument with examples.
- Be ready for using examples given by the Avenger.
- Be ready to answer questions on the topic posed by Avenger.
- Avenger will pass you on a topic only if they are convinced you master it.

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.
- Use proper English; illustrate your argument with examples.
- Be ready for using examples given by the Avenger.
- Be ready to answer questions on the topic posed by Avenger.
- Avenger will pass you on a topic only if they are convinced you master it.
- The pair can have a repeat attempt(s) with the Avenger.

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.
- Use proper English; illustrate your argument with examples.
- Be ready for using examples given by the Avenger.
- Be ready to answer questions on the topic posed by Avenger.
- Avenger will pass you on a topic only if they are convinced you master it.
- The pair can have a repeat attempt(s) with the Avenger.
- Other staff may check as well for repeated attempts.

Mastery Checks: The Process

- Form pairs
- (Your Avenger will help you find a partner in a different Studio if you can't find one in your Studio.)
- Make appointment with Avenger
- For each topic, you have 10 minutes to convince the Avenger that both of you are mastering the topic.
- Use proper English; illustrate your argument with examples.
- Be ready for using examples given by the Avenger.
- Be ready to answer questions on the topic posed by Avenger.
- Avenger will pass you on a topic only if they are convinced you master it.
- The pair can have a repeat attempt(s) with the Avenger.
- Other staff may check as well for repeated attempts.
- Try finish MC1 b4 Recess Week. (Great prep for Midterm!)

Contest: Beautiful Runes

1. List Processing (2.2.1)
   - Review: The `length` of a list
   - `append`
   - `reverse`
2. Continuation-Passing Style
3. Higher-order List Processing (2.2.1–2.2.3)
4. Trees and Tree Processing (2.2.2)

Computing the length of a list (2.2.1)

Computing the length of a list (2.2.1)

The length of...

Computing the length of a list (2.2.1)

The length of...

...the empty list is 0

Computing the length of a list (2.2.1)

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**.

Computing the length of a list (2.2.1)

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

We can do this with an iterative process!

We can do this with an iterative process!

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

function length_iter(xs) {  // iterative
  function len(xs, counted_so_far) {
    return is_null(xs)
        ? counted_so_far
        : len(tail(xs),
              counted_so_far + 1);
  }
  return len(xs, 0);
}
```

Appending two lists

Append `list(1, 3, 5)` and `list(2, 4)` results in

`list(1, 3, 5, 2, 4)`

Assumption: Both arguments are lists

Strategy for `append(xs, ys)`

If `xs` is empty, return `ys`.

Strategy for `append(xs, ys)`

If `xs` is empty, return `ys`.

Otherwise, wishful thinking!

Strategy for `append(xs, ys)`

If `xs` is empty, return `ys`.

Otherwise, wishful thinking!

Append the tail of `xs` to `ys`

Strategy for `append(xs, ys)`

If `xs` is empty, return `ys`.

Otherwise, wishful thinking!

Append the tail of `xs` to `ys`

Form a pair of the head of `xs` and the result.

The same in Source

```
function append(xs, ys) {
  return is_null(xs)
      ? ys
      : pair(head(xs),
             append(tail(xs), ys));
}
```

The same in Source

```
function append(xs, ys) {
  return is_null(xs)
      ? ys
      : pair(head(xs),
                append(tail(xs), ys));
}
```

Time complexity? Space complexity?

Reversing a list: First attempt

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

Reversing a list: First attempt

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

What's wrong?

Reversing a list: First attempt

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

What's wrong?

Result not a list.

Reversing a list: First attempt

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

What's wrong?
Result not a list.
We simply *reverse*
roles of head and tail.

Reversing a list: Correct but naive

```
function reverse(xs) {
    return is_null(xs)
        ? null
        : append(reverse(tail(xs)),
                 list(head(xs)));
}
```

Reversing a list: Correct but naive

```
function reverse(xs) {
    return is_null(xs)
        ? null
        : append(reverse(tail(xs)),
                 list(head(xs)));
}
```

Correct, but what about runtime?

Reversing a list efficiently

```
function reverse(xs) {
  function rev(original, reversed) {
    return is_null(original)
           ? reversed
           : rev(tail(original),
                 pair(head(original),
                      reversed));
  }
  return rev(xs,null);
}
```

Reversing a list efficiently

```
function reverse(xs) {
  function rev(original, reversed) {
    return is_null(original)
           ? reversed
           : rev(tail(original),
                 pair(head(original),
                      reversed));
  }
  return rev(xs,null);
}
```

Order of growth? Time? Space?

1. List Processing (2.2.1)
2. Continuation-Passing Style
   - Closer Look at `append`
   - Iterative process with `reverse`
   - Iterative process with Continuation Passing
3. Higher-order List Processing (2.2.1–2.2.3)
4. Trees and Tree Processing (2.2.2)

A closer look in Source at append

```
function append(xs, ys) {
    return is_null(xs)
           ? ys
           : pair(head(xs),
                  append(tail(xs), ys));
}
```

A closer look in Source at append

```
function append(xs, ys) {
    return is_null(xs)
           ? ys
           : pair(head(xs),
                  append(tail(xs), ys));
}
```

Can we do this using an iterative process?

Iterative process, first attempt

```
function append_iter(xs, ys) {
  return is_null(xs)
      ? ys
      : append_iter(tail(xs),
                    pair(head(xs), ys));
}
```

Iterative process using reverse

```
function app_rev_iter(xs, ys) {
  return is_null(xs)
    ? ys
    : app_rev_iter(tail(xs), pair(head(xs), ys));
}
function append(xs, ys) {
  return append_rev_iter(
           reverse(xs), ys);
}
```

Iterative process using reverse

```
function app_rev_iter(xs, ys) {
  return is_null(xs)
    ? ys
    : app_rev_iter(tail(xs), pair(head(xs), ys));
}
function append(xs, ys) {
  return append_rev_iter(
           reverse(xs), ys);
}
```

Can we do this without reverse?

An iterative version of append

```
function append(xs, ys) {       // recursive proc
  return is_null(xs)
         ? ys
         : pair(head(xs),
                append(tail(xs), ys));
}
function app(current_xs, ys, c) {  // iter. proc
  return is_null(current_xs)
         ? c(ys)
         : app(tail(current_xs), ys,
               x => c(pair(head(current_xs), x)));
}
function append_iter(xs, ys) {
    return app(xs, ys, x => x);
}
```

Continuation-Passing Style

```
function app(current_xs, ys, c) {  // iter. proc
  return is_null(current_xs)
         ? c(ys)
         : app(tail(current_xs), ys,
               x => c(pair(head(current_xs), x)));
}
function append_iter(xs, ys) {
    return app(xs, ys, x => x);
}
```

Programming Pattern: CPS

Passing the deferred operation as a function in an extra argument is called "Continuation-Passing Style" (CPS).

Continuation-Passing Style

```
function app(current_xs, ys, c) {  // iter. proc
  return is_null(current_xs)
         ? c(ys)
         : app(tail(current_xs), ys,
               x => c(pair(head(current_xs), x)));
}
function append_iter(xs, ys) {
    return app(xs, ys, x => x);
}
```

Programming Pattern: CPS

Passing the deferred operation as a function in an extra argument is called "Continuation-Passing Style" (CPS).
We can convert *any* recursive function this way!

Recall L3-Excursion 2: a "divine" solution

```
function fractal_5(rune, n) {
    return n === 1
           ? rune
           : beside(rune,fractal_5(stack(rune,rune),
                                   n - 1));
}
```

Recall L3-Excursion 2: a "divine" solution with CPS

```
function fractal_5(rune, n) {
    return n === 1
        ? rune
        : beside(rune,fractal_5(stack(rune,rune),
                                  n - 1));
}
function frac(rune, n, c) {
    return n === 1
       ? c(rune)
       : frac(stack(rune, rune), n - 1,
              res => c(beside(rune, res)));
}
function fractal_5_iter(rune, n) {
    return frac(rune, n, rune => rune);
}
```

1. List Processing (2.2.1)
2. Continuation-Passing Style
3. Higher-order List Processing (2.2.1–2.2.3)
   - `map`
   - `accumulate`
   - `filter`
4. Trees and Tree Processing (2.2.2)

Map, reduce, filter

Advantage of functional programming

Results only depend on arguments.

Map, reduce, filter

Advantage of functional programming

Results only depend on arguments. FP is *scalable*.

Map, reduce, filter

Advantage of functional programming

Results only depend on arguments. FP is *scalable*.

Functional programming in industry

FP has achieved widespread use in software industry.

Map, reduce, filter

Advantage of functional programming

Results only depend on arguments. FP is *scalable*.

Functional programming in industry

FP has achieved widespread use in software industry.

Map/reduce/filter

Set of abstractions that forms the core of many big data processing engines, such as Apache Hadoop.

Map, reduce, filter

Advantage of functional programming

Results only depend on arguments. FP is *scalable*.

Functional programming in industry

FP has achieved widespread use in software industry.

Map/reduce/filter

Set of abstractions that forms the core of many big data processing engines, such as Apache Hadoop.

Terminology

Following SICP, we say *accumulate* instead of *reduce*.

Scaling a list (2.2.1)

Let us scale all elements of a list of numbers by a factor `f`.

```
function scale_list(xs, factor) {
  return is_null(xs)
      ? null
      : pair(factor * head(xs),
             scale_list(tail(xs),
                        factor));
}
```

Squaring a list

Let us *square* all elements of a list of numbers.

```
function square_list(xs) {
    const square = x => x * x;
    return is_null(xs)
        ? null
        : pair(square(head(xs)),
                 square_list(tail(xs)));
}
```

Abstraction: `map` (2.2.1)

Mapping means applying a given function `f` element-wise to a given list `xs`.

Abstraction: `map` (2.2.1)

Mapping means applying a given function `f` element-wise to a given list `xs`.

The result is a list consisting of the results of applying `f` to each element of `xs`.

Abstraction: map (2.2.1)

Mapping means applying a given function f element-wise to a given list xs.

The result is a list consisting of the results of applying f to each element of xs.

```
function scale_list(xs, factor) {
  return map(x => factor * x, xs);
}
```

Abstraction: map (2.2.1)

Mapping means applying a given function f element-wise to a given list xs.

The result is a list consisting of the results of applying f to each element of xs.

```
function scale_list(xs, factor) {
  return map(x => factor * x, xs);
}

function square_list(xs) {
  return map(x => x * x, xs);
}
```

Definition of map

```
function map(fun, xs) {
  return is_null(xs)
         ? null
         : pair(fun(head(xs)),
                map(fun, tail(xs)));
}
```

Definition of map

```
function map(fun, xs) {
  return is_null(xs)
         ? null
         : pair(fun(head(xs)),
                map(fun, tail(xs)));
}

function scale_list(xs, factor) {
  return map(x => factor * x, xs);
}

function square_list(xs) {
  return map(x => x * x, xs);
}
```

Example: summing the elements of a list (2.2.3)

Problem

Compute the sum of all elements of a given list of numbers

Example: summing the elements of a list (2.2.3)

Problem

Compute the sum of all elements of a given list of numbers

```
function list_sum(xs) {
    return is_null(xs)
        ? 0
        : head(xs) + list_sum(tail(xs));
}
```

Example: summing the elements of a list (2.2.3)

Problem

Compute the sum of all elements of a given list of numbers

```
function list_sum(xs) {
    return is_null(xs)
           ? 0
           : head(xs) + list_sum(tail(xs));
}
```

Right-to-left folding

`list_sum(list(1,2,3,4))` computes
`1 + (2 + (3 + (4 + 0)))`.

accumulate: an abstraction for right-to-left folding

```
function list_sum(xs) { // programmed "by hand"
    return is_null(xs)
        ? 0
        : head(xs) + list_sum(tail(xs));
}
```

accumulate: an abstraction for right-to-left folding

```
function list_sum(xs) { // programmed "by hand"
    return is_null(xs)
        ? 0
        : head(xs) + list_sum(tail(xs));
}

function accumulate(f, initial, xs) {
  return is_null(xs)
    ? initial
    : f(head(xs),accumulate(f,initial,tail(xs)));
}
function list_sum(xs) {  // using accumulate
    return accumulate( (x, y) => x + y, 0, xs);
}
```

filter (2.2.3)

Problem: take only even elements of list of numbers

filter (2.2.3)

Problem: take only even elements of list of numbers

```
filter(x => x % 2 === 0, list(1, 2, 3, 4, 5, 6));
```

filter (2.2.3)

Problem: take only even elements of list of numbers

```
filter(x => x % 2 === 0, list(1, 2, 3, 4, 5, 6));

function filter(pred, xs) {
  return is_null(xs)
       ? xs
       : pred(head(xs))
       ? pair(head(xs),
              filter(pred, tail(xs)))
       : filter(pred, tail(xs));
}
```

1. List Processing (2.2.1)
2. Continuation-Passing Style
3. Higher-order List Processing (2.2.1–2.2.3)
4. Trees and Tree Processing (2.2.2)
   - What are trees?
   - Higher-order Tree Processing
   - Counting Data Items
   - Syntax Trees

Trees (2.2.2)

Trees (2.2.2)

A list *of a certain type*

...is either `null` or a pair whose head is of that type and whose tail is a list of that type

Trees (2.2.2)

A list *of a certain type*

...is either `null` or a pair whose head is of that type and whose tail is a list of that type

A tree *of a certain type*

... is a list whose elements are of that type, or trees of that type.

Trees (2.2.2)

A list *of a certain type*

...is either `null` or a pair whose head is of that type and whose tail is a list of that type

A tree *of a certain type*

... is a list whose elements are of that type, or trees of that type.

Consider:

```
const tree = list(0, list(1,2), list(3,4), 5);
```

Trees (2.2.2)

A list *of a certain type*

...is either `null` or a pair whose head is of that type and whose tail is a list of that type

A tree *of a certain type*

... is a list whose elements are of that type, or trees of that type.

Consider:

```
const tree = list(0, list(1,2), list(3,4), 5);
```

Caveat

Cannot consider `null` or pair as “certain type” for trees.

Scaling trees (2.2.2)

Example: scale each data item by a factor 10

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

```
scale_tree(my_tree, 10);
```

should have the same result as:

```
list(10,list(20,list(30,40),50),list(60,70));
```

Scaling trees: Idea

Recall: A *tree* is a list whose elements are data items, or trees.

Scaling trees: Idea

Recall: A *tree* is a list whose elements are data items, or trees.

Idea: Map over the list. If element is a data item, scale element.
If not: *scale tree*

.

Scaling trees: Implementation using list map

```
function scale_tree(tree, factor) {
    return map(sub_tree =>
                   ! is_list(sub_tree)
                   ? factor * sub_tree
                   : scale_tree(sub_tree, factor),
               tree);
}
```

Abstraction: Mapping over trees

```
function map_tree(f, tree) {
    return map(sub_tree =>
                   ! is_list(sub_tree)
                   ? f(sub_tree)
                   : map_tree(f, sub_tree),
               tree);
}
```

Scaling trees: Implementation using map_tree

```
function scale_tree(tree, factor) {
    return map_tree(data_item =>
                        data_item * factor,
                    tree);
}
```

.

Counting Data Items (2.2.2)

Task: Compute the number of data items in a given tree?

```
const tree = pair(list(1, 2), list(3, 4));
```

What is `count_data_items(tree)`?

.

Counting Data Items: Idea

Every tree is a list.

Counting Data Items: Idea

Every tree is a list.

That list can be empty, in which case we return 0.

Counting Data Items: Idea

Every tree is a list.

That list can be empty, in which case we return 0.

If the list is not empty, we add the *number of data items* of the head to the *number of data items* of the tail.

Counting Data Items: Idea

Every tree is a list.

That list can be empty, in which case we return 0.

If the list is not empty, we add the *number of data items* of the head to the *number of data items* of the tail.

The head can be a tree, in which case we need to count *its* data items. If it's not a tree, it's a data item and we count 1.

Counting Data Items

```
function count_data_items(tree) {
  return is_null(tree)
         ? 0
         : ( is_list(head(tree))
             ? count_data_items(head(tree))
             : 1 )
           +
           count_data_items(tail(tree));
}
```

Counting Data Items

```
function count_data_items(tree) {
  return is_null(tree)
         ? 0
         : ( is_list(head(tree))
             ? count_data_items(head(tree))
             : 1 )
           +
           count_data_items(tail(tree));
}
```

Exercise

Design an abstraction that lets you count the elements using `(x, y) => x + y` and that lets you compute a list of elements using append

How to Implement Programming Language Tools?

How to Implement Programming Language Tools?

- Goal: implement language processing tools

How to Implement Programming Language Tools?

- Goal: implement language processing tools
- Programmers write text

How to Implement Programming Language Tools?

- Goal: implement language processing tools
- Programmers write text
- Text is “unstructured”

How to Implement Programming Language Tools?

- Goal: implement language processing tools
- Programmers write text
- Text is “unstructured”; we need program *structure*
- Solution: *parser* translating from string to tree

How to Implement Programming Language Tools?

- Goal: implement language processing tools
- Programmers write text
- Text is "unstructured"; we need program *structure*
- Solution: *parser* translating from string to tree
- Implement language processors
  as data processors

How to Implement Programming Language Tools?

- Goal: implement language processing tools
- Programmers write text
- Text is "unstructured"; we need program *structure*
- Solution: *parser* translating from string to tree
- Implement language processors as data processors

Teaser for Lecture L11

The evaluator, which determines the meaning of statements in a programming language, is just another program.

Summary

Summary

- `length`, `reverse`, `append`

Summary

- `length`, `reverse`, `append`
- Continuation-Passing Style for iterative processes

Summary

- `length`, `reverse`, `append`
- Continuation-Passing Style for iterative processes
- Higher-order list processing with `map`, `accumulate`, `filter`

Summary

- `length`, `reverse`, `append`
- Continuation-Passing Style for iterative processes
- Higher-order list processing with `map`, `accumulate`, `filter`
- Trees

Summary

- `length`, `reverse`, `append`
- Continuation-Passing Style for iterative processes
- Higher-order list processing with `map`, `accumulate`, `filter`
- Trees and higher-order tree processing

Summary

- `length`, `reverse`, `append`
- Continuation-Passing Style for iterative processes
- Higher-order list processing with `map`, `accumulate`, `filter`
- Trees and higher-order tree processing
- Syntax trees allow us to represent the structure of programs

Summary

- `length`, `reverse`, `append`
- Continuation-Passing Style for iterative processes
- Higher-order list processing with `map`, `accumulate`, `filter`
- Trees and higher-order tree processing
- Syntax trees allow us to represent the structure of programs
- Outlook: Brief B5: Overview of programming language processing