L3: Higher-order Functions; Scope of Names

Structure and Interpretation of Computer Programs

Martin Henz

July 7, 2023

1. Announcements
2. Nested declarations
3. Higher-order functions
4. Scope of names

Announcements

- Missions: they are the main pieces of homework! Work in the assigned pairs.
- Quests: they are "extra homework", for practice! Quests are not graded.
- Contest "Beautiful Runes" opens today! Closes on Monday, 10/7. Contests are not graded.
- Exam: On Thursday, 20/7, 10am to 12noon.

Schedule

- Lectures, 10am–12noon: Saturday (8/7), Monday–Saturday (10/7–15/7)
- Labs, 2pm–6pm: Monday–Saturday (10/7–15/7)
- Project work: 17/7–12/7
- Project presentations: Tuesday 25/7

Some Words of Advice

- Read the textbook
- Use the substitution model (and stepper tool, if needed)
- Think, then program
- Less is more

1 Announcements

2 Nested declarations
- Nested function declarations
- Nested constant declarations

3 Higher-order functions

4 Scope of names

Names can refer to intermediate values

Example from SICP JS 1.3.2

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

Compute $f(2, 3)$

```
function f(x, y) {
    const a = 1 + x * y;
    const b = 1 - y;
    return x * square(a) + y * b + a * b;
}

f(2, 3);
```

Can we do this with functions, too?

```
function sum_of_squares(a, b) {
    return square(a) + square(b);
}
function hypotenuse(a, b) {
    return math_sqrt(sum_of_squares(a, b));
}
```

"Hiding"

Can we hide the function `sum_of_squares` so that only the body of `hypotenuse` can "see" it?

Yes, we can! (see also SICP JS 1.1.8)

```
function hypotenuse(a, b) {
    function sum_of_squares(a, b) {
        return square(a) + square(b);
    }
    return math_sqrt(sum_of_squares(a, b));
}
```

Function nesting

Function declarations can be *nested inside of* other function declarations, and only visible there.

sum_of_squares could use parameters of hypotenuse

```
function hypotenuse(a, b) {
    function sum_of_squares() {
        return square(a) + square(b);
    }
    return math_sqrt(sum_of_squares());
}
```

Even simpler version: Using nested const

```
function hypotenuse(a, b) {
    const sum_of_squares = square(a) + square(b);
    return math_sqrt(sum_of_squares);
}
```

Can we avoid tree recursion?

```
function fractal_1(rune, n) {
    return n === 1
        ? rune
        : beside(rune,
              stack(fractal_1(rune, n - 1),
                    fractal_1(rune, n - 1)));
}
```

Question

Can we implement this function with linear recursion?

Is this a good idea?

```
function fractal_2(rune, n) {
    const smaller_frac = fractal_2(rune, n - 1);
    return n === 1
        ? rune
        : beside(rune,
            stack(smaller_frac, smaller_frac));
}
```

Can we declare a const...

...**just for the alternative of the conditional**?

Conditional *statements* (see SICP JS 1.3.2)

```
function fractal_3(rune, n) {
    if (n === 1) {
        return rune;
    } else {
        const f = fractal_3(rune, n - 1);
        return beside(rune, stack(f, f));
    }
}
```

- Each branch of the conditional is a *block*.
- A block can have local names, only visible inside the block.
- Remember to return a result *in each branch*.
  (Otherwise `undefined` is returned.)

1. Announcements
2. Nested declarations
3. Higher-order functions
   - Functions as arguments (1.3.1)
   - Lambda expressions (1.3.2)
   - Functions as returned values (1.3.4)
   - Summary of constructs discussed today
4. Scope of names

Passing functions to functions

```
function f(g, x) {
    return g(x);
}

function g(y) {
    return y + 1;
}

f(g, 7);
```

Passing more functions to functions

```
function f(g, x) {
    return g(g(x));
}

function g(y) {
    return y + 1;
}

f(g, 7);
```

Abstraction: Recall `repeat_pattern`

Repeating the pattern *n* times

```
repeat_pattern(4, make_cross, rcross);
```

Passing a rune function

```
function transform_mosaic(p1, p2, p3, p4,
                          transform) {
    return transform(mosaic(p1, p2, p3, p4));
}
```

Look at this function (1.3.1)

```
function sum_integers(a, b) {
     return a > b
          ? 0
          : a + sum_integers(a + 1, b);
}
```

...and this one

```
function cube(x) {
    return x * x * x;
}

function sum_skip_cubes(a, b) {
    return a > b
        ? 0
        : cube(a) + sum_skip_cubes(a + 2, b);
}
```

Abstraction (1.3.1)

```
function sum(a, b) {
    return a > b ? 0
                 : ⟨compute value with a⟩
                   +
                   sum(⟨next value from a⟩, b);
}
```

in Source:

```
function sum(term, a, next, b) {
    return a > b ? 0
                 : term(a)
                   +
                   sum(term, next(a), next, b);
}
```

sum_integers using sum

```
function identity(x) {
    return x;
}
function plus_one(x) {
    return x + 1;
}
function sum_integers(a, b) {
    return sum(identity, a, plus_one, b);
}
```

sum_skip_cubes using sum

```
function cube(x) {
    return x * x * x;
}
function plus_two(x) {
    return x + 2;
}
function sum_skip_cubes(a, b) {
    return sum(cube, a, plus_two, b);
}
```

sum_skip_cubes using sum

```
function cube(x) {
    return x * x * x;
}
function plus_two(x) {
    return x + 2;
}
function sum_cubes(a, b) {
    return sum(cube, a, plus_two, b);
}
```

Visibility

Can we "hide" `cube` and `plus_two` inside of `sum_skip_cubes`?

Yes, we can! (see also SICP JS 1.1.8)

```
function sum_skip_cubes(a, b) {
    function cube(x) {
        return x * x * x;
    }
    function plus_two(x) {
        return x + 2;
    }
    return sum(cube, a, plus_two, b);
}
```

Another look at such local functions

```
function sum_skip_cubes(a, b) {
    function cube(x) {
        return x * x * x;
    }
    function plus_two(x) {
        return x + 2;
    }
    return sum(cube, a, plus_two, b);
}
```

This is still quite verbose

Do we need all these words such as `function`, `return`?
Do we need to even give names to these functions?

No, we don't! Lambda expressions

```
function sum_skip_cubes(a, b) {
    function cube(x) {
        return x * x * x;
    }
    function plus_two(x) {
        return x + 2;
    }
    return sum(cube, a, plus_two, b);
}
// instead just write:

function sum_skip_cubes(a, b) {
    return sum(x => x * x * x, a, x => x + 2, b);
}
```

Lambda expressions (1.3.2)

New kinds of expressions

(*parameters*) => *expression*

If there is only one parameter, you can write

parameter => *expression*

Meaning

The expression evaluates to a function value.
Function has given *parameters* and `return` *expression* ; as body.

An alternative syntax for function declaration

```
function plus4(x) {
    return x + 4;
}
```

can be written as

```
const plus4 = x => x + 4;
```

Returning Functions from Functions (1.3.4)

```
function make_adder(x) {
    function adder(y) {
        return x + y;
    }
    return adder;
}

const adder_four = make_adder(4);
adder_four(6);
```

...or with the new lambda expressions

```
function make_adder(x) {
    return y => x + y;
}

const adder_four = make_adder(4);
adder_four(6);
```

Returning Functions from Functions

```
function make_adder(x) {
    return y => x + y;
}

( make_adder(4) )(6);

// you can also write:
//
// make_adder(4)(6);
```

Returning Functions from Functions

```
function make_adder(x) {
    return y => x + y;
}

const adder_1 = make_adder(1);
const adder_2 = make_adder(2);

adder_1(10); // returns 11

adder_2(20); // returns 22
```

Summary of constructs discussed today

- Nested constant and function declaration statements
- Conditional statements and blocks
- **Lambda expressions**

1. Announcements
2. Nested declarations
3. Higher-order functions
4. Scope of names
   - Examples
   - Overview of scoping rules
   - The details

Scope of names: an example

```
const z = 2;
function f(g) {
    const z = 4;
    return g(z);
}

f( y => y + z );
```

Questions about scope

What names are declared by this program?
Which declaration does each name occurrence refer to?

Scope of names: another example

```
const x = 10;
function square(x) {
    return x * x;
}
function addx(y) {
    return y + x;
}
square(x + 5) * addx(x + 20);
```

Questions about scope

Which declaration does each occurrence of `x` refer to?

Scope of names: yet another example

```
const pi = 3.141592653589793;
function circle_area_from_radius(r) {
    const pi = 22 / 7;
    return pi * square(r);
}
```

Questions about scope

Which declaration does the occurrence of `pi` refer to?

Scope of names: hypotenuse example

```
function square(x) {
    return x * x;
}
function hypotenuse(a, b) {
    function sum_of_squares() {
        return square(a) + square(b);
    }
    return math_sqrt(sum_of_squares());
}
```

Names can refer to declarations outside of the immediately surrounding function declaration.

Overview of scoping rules

Declarations mandatory

All names in Source must be declared.

Forms of declaration

1. Pre-declared names
2. Constant declarations
3. Parameters of function declarations and lambda expressions
4. Function name of function declarations

Scoping rule

A name occurrence refers to the *closest surrounding* declaration.

Forms of Declaration

(1) Pre-declared names

The Source §1 pages tell us what names are pre-declared, e.g. `math_floor`.
We can also `import` further pre-declared names from modules. For example, from the rune module:

```
import { heart, quarter_turn_right } from "rune";
```

Forms of Declaration

(2) Constant declarations

The scope of a constant declaration is the closest surrounding pair of `{...}`, or the whole program, if there is none.

Example

```
function f(x, y) {
    if (x > 0) {
        const z = x * y;
        return math_sqrt(z);
    } else {
        ...
    }
}
```

Forms of Declaration

(3) Parameters

The scope of the parameters of a lambda expression or function declaration is the body of the function.

```
function f(x, y, z) {
    ... x ... y ... z ...
}

(v, w, u) => ... v ... w ... u ...
```

Forms of Declaration

(4) Function name

The scope of the function name of a function declaration is as if the function was declared with `const`.

```
function f(x) {
    ...
}
```

as if we wrote

```
const f = ...;
```

Lexical scoping

Scoping rule

A name occurrence refers to the *closest surrounding* name declaration.

```
function f(x) {
    function g(w) {
        const x = ...;
        const y = x + 1;
        return x =>
            ...x...;
    }
    return g(x);
}
```

Important Ideas

- *Hiding* can be a useful abstraction technique
- *Recursion* is an elegant pattern of problem solving
- Functions can be *passed to* functions
- Functions can be *returned from* functions
- Higher-order functions are *useful* for building abstractions
- With nested functions and conditional statements, we need to understand the *scope* of names

Concluding Chapter 1

- Mental model: substitution model
- Big ideas: iterative/recursive processes, higher-order, scope
- Problem solving technique: recursion ("wishful thinking")
- Look out for Chapter 2: Building Abstractions with Data, starting with L5 tomorrow