B12: The Power of Simplicity

CS1101S: Programming Methodology

Martin Henz

November 3, 2021

1. Recursive and iterative processes
   - The good news
   - Simplest case
   - Complications
   - Cookbook
   - Exercise: “Iterativization”

2. Debugging

repeat_pattern_1, our recursive process

```
// repeat pattern n times
function repeat_pattern_1(n, pat, pic) {
    return n === 0
        ? pic
        : pat(repeat_pattern_1(n - 1, pat, pic));
}
```

repeat_pattern_1, our recursive process

```
// repeat pattern n times
function repeat_pattern_1(n, pat, pic) {
    return n === 0
        ? pic
        : pat(repeat_pattern_1(n - 1, pat, pic));
}

repeat_pattern_1(3, q_t_r, heart)
```

repeat_pattern_1, our recursive process

```
// repeat pattern n times
function repeat_pattern_1(n, pat, pic) {
    return n === 0
        ? pic
        : pat(repeat_pattern_1(n - 1, pat, pic));
}

repeat_pattern_1(3, q_t_r, heart)
```

Recursive process

The applications of pat *accumulate* as a result of the recursive calls. They are deferred operations.

repeat_pattern_2, an iterative process

```
function repeat_pattern_2(n, pat, pic) {
    return n === 0
        ? pic
        : repeat_pattern_2(n - 1, pat, pat(pic));
}
```

repeat_pattern_2, an iterative process

```
function repeat_pattern_2(n, pat, pic) {
    return n === 0
        ? pic
        : repeat_pattern_2(n - 1, pat, pat(pic));
}

repeat_pattern_2(3, q_t_r, heart)
```

repeat_pattern_2, an iterative process

```
function repeat_pattern_2(n, pat, pic) {
    return n === 0
        ? pic
        : repeat_pattern_2(n - 1, pat, pat(pic));
}

repeat_pattern_2(3, q_t_r, heart)
```

Iterative process

With applicative order reduction, the pat function is applied before the recursive call. There is no deferred operation.

The good news

The good news is that...

...it is possible to turn every program into a program the gives rise to an iterative process.

The good news

The good news is that...

...it is possible to turn every program into a program the gives rise to an iterative process.

A programming method for this is "Continuation Passing Style"

Review CPS, as presented in Lecture L5.

The good news

The good news is that...

...it is possible to turn every program into a program the gives rise to an iterative process.

A programming method for this is "Continuation Passing Style"

Review CPS, as presented in Lecture L5.

Special cases—a cookbook

...but there are few special cases that may be useful *in the assessments*!

A bigger example

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}
factorial(5);
```

Why is it so easy for repeat_pattern?

```
function repeat_pattern_1(n, pat, pic) {
    return n === 0
        ? pic
        : pat(repeat_pattern_1(n - 1, pat, pic));
}

function repeat_pattern_2(n, pat, pic) {
    return n === 0
        ? pic
        : repeat_pattern_2(n - 1, pat, pat(pic));
}
```

Why is it so easy for `repeat_pattern`?

The reason

The operation is the same each time!

It does not matter...

...whether we apply the pattern before or after the recursive call!

Operation changes

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}
```

Observation: Order still does not matter

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}
```

Observation: Order still does not matter

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}

function fact_iter(n) {
    function helper(i, acc) {
        return i === 1 ? acc
               : helper(i - 1, acc * i);
    }
    return helper(n, 1);
}
```

What if the order matters?

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

What if the order matters?

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

function map_iter(f, xs) { // is this right?
  function helper(ys, acc) {
    return is_null(ys)
      ? acc
      : helper(tail(ys), pair(f(head(ys)), acc));
  }
  return helper(xs, null); }
```

Solution: Reverse the list! (in this case)

```
function map_iter(f, xs) {
  function helper(ys, acc) {
    return is_null(ys)
      ? acc
      : helper(tail(ys), pair(f(head(ys)), acc));
  }
  return helper(reverse(xs), null);
}
```

CPS: A simple example

```
function g(y) {
    return 1 + f(y);
}
function f(x) {
    return x * x;
}
g(7);
```

CPS: A simple example

```
function g(y) {
    return 1 + f(y);
}
function f(x) {
    return x * x;
}
g(7);
```

Deferred operation

After `f` returns, we still need to carry out `1 + f(y)`.

CPS: A simple example

```
function g(y) {
    return 1 + f(y);
}
function f(x) {
    return x * x;
}
g(7);
```

Deferred operation

After `f` returns, we still need to carry out `1 + f(y)`.

What if...

CPS: A simple example

```
function g(y) {
    return 1 + f(y);
}
function f(x) {
    return x * x;
}
g(7);
```

Deferred operation

After `f` returns, we still need to carry out `1 + f(y)`.

What if...

...we pass the this obligation to add 1 along to `f` as a *continuation*?

CPS: A simple example, concluded

```
function cps_g(y) {
    return cps_f(y, a => a + 1);
}
function cps_f(x, cont) {
    return cont(x * x);
}
```

Functions get one extra argument

CPS: A simple example, concluded

```
function cps_g(y) {
    return cps_f(y, a => a + 1);
}
function cps_f(x, cont) {
    return cont(x * x);
}
```

Functions get one extra argument

A *continuation* that is applied to whatever result they produce.

CPS: A simple example, concluded

```
function cps_g(y) {
    return cps_f(y, a => a + 1);
}
function cps_f(x, cont) {
    return cont(x * x);
}
```

Functions get one extra argument

A *continuation* that is applied to whatever result they produce.

Initially we pass a default continuation

```
cps_g(7, x => x);
```

The cookbook

- Is the operation the same? If yes, just do it before the call
- Does the order of operations matter? If no, use helper function
- If yes: adjust things for reversing order.
- If that’s very difficult: Use idea from CPS (review Lecture L5)

Exercise: Iterativization

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

Exercise: Write a version of the function `filter`...

...that gives rise to an iterative process.

Good habits: Debugging using `display`

You can trace the execution of your program...

...by inserting `display` calls in the right places.

Good habits: Debugging using `display`

You can trace the execution of your program...

...by inserting `display` calls in the right places.

Simple `display`

`display(my_value)` displays whatever `my_value` refers to

Good habits: Debugging using `display`

You can trace the execution of your program...

...by inserting `display` calls in the right places.

Simple `display`

`display(my_value)` displays whatever `my_value` refers to

`display` with extra argument

`display(my_value, "value of my_value: ")` displays
`value of my_value: ...`

Good habits: Debugging using `display`

You can trace the execution of your program...

...by inserting `display` calls in the right places.

Simple `display`

`display(my_value)` displays whatever `my_value` refers to

`display` with extra argument

`display(my_value, "value of my_value: ")` displays
`value of my_value: ...`

Fringe benefit: `display` returns its first argument

see here for an example

Good habits: Invest in testing

Make sure you have a way to tell...

...whether your solution is right.

Good habits: Invest in testing

Make sure you have a way to tell...

...whether your solution is right.

It pays to invest...

...in a little function that displays the result, if needed.