CSC324: Principles of Programming Languages

Lecture 12

8 July 2020

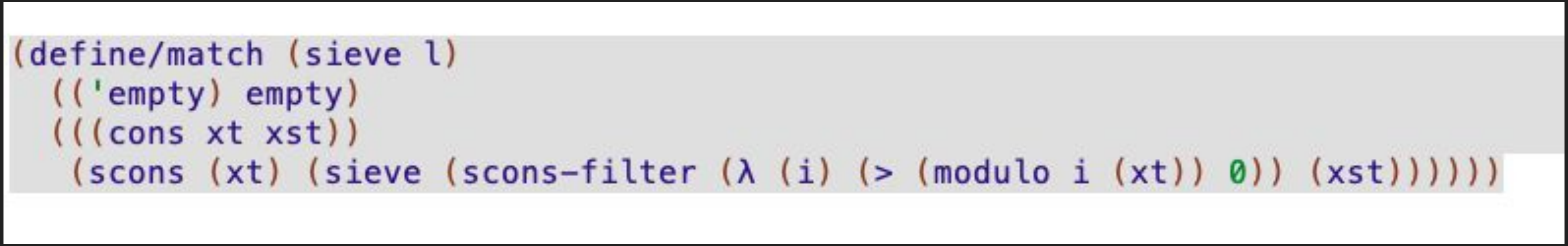
Hope you all had a nice break!

Previously, we saw how

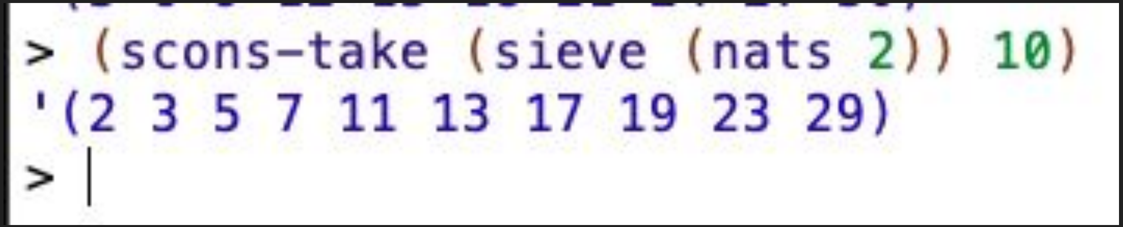
- evaluation of subexpressions could be deferred to be done **lazily**
- streams extend lazy evaluation of expressions into the realm of designing infinite data structures (wow #wow)

Shameless plug: if you are feeling rusty on this stuff after the break, I recommend working through the bonus tutorial videos I link to on the schedule page.

Our prime sieve from before the break….



```
(define/match (sieve l)
  (('empty) empty)
  (((cons xt xst))
   (scons (xt) (sieve (scons-filter (λ (i) (> (modulo i (xt)) 0)) (xst))))))
```



```
> (scons-take (sieve (nats 2)) 10)
'(2 3 5 7 11 13 17 19 23 29)
> 
```

Hope you all had a nice break!

Previously, we saw how

- evaluation of subexpressions could be deferred to be done **lazily**
- streams extend lazy evaluation into the realm of data structure design

- **How operating over lazy lists is useful for "generate and filter" applications: we generated all numbers and filtered out the ones that aren't prime**

Hope you all had a nice break!

Previously, we saw how

- evaluation of subexpressions could be deferred to be done **lazily**
- streams extend lazy evaluation into the realm of data structure design

- **How operating over lazy lists is useful for "generate and filter" applications: we generated all numbers and filtered out the ones that aren't prime**

Today: Redexes, Continuations, and non-deterministic programming

Some more expression terminology

```
(+ (+ 1 2) (* 3 1))
```

What is the first step of evaluating this expression?

Some more expression terminology

```
(+ (+ 1 2) (* 3 1))
```

Some more expression terminology

```
(+ (+ 1 2) (* 3 1))
```

Definition: In an expression that can be further reduced, the part that changes in a single step is said to be the **redex ("reducible expression")**

A property of evaluating the **redex** is that it has no idea of its place in a larger expression.

Some more expression terminology

```
(+ (+ 1 2) (* 3 1))
```

Definition: In an expression that can be further reduced, the part that does not change in a single step is said to be the **continuation** of the expression.

A property of the **continuation** is that it has no idea through how its inner expressions were ultimately derived.

Some more expression terminology

```
(+ 3 (* 3 1))
```

So let's say we evaluate the current redex...

Some more expression terminology

```
(+ 3 (* 3 1))
```

Note that when we start evaluating the next redex, the next continuation contains the subexpression that we just evaluated.

Continuations and function calls

```
(* 2 (slow-add 3 4))
```

In our previous example, the continuation was always an "outer" expression and the redex was an "inner" subexpression. What is the continuation of a function call?

Continuations and function calls

```
(* 2 (if (= m 0) n
         (slow-add (- m 1)
                   (+ n 1)))))
```

It's the body of the function! (and any surrounding expressions, of course)

A "hole" in the continuation

```
(+ (+ 1 2) (* 3 1))
```

A "hole" in the continuation

```
(+ ... (* 3 1))
```

Note the similarity...

```
(+ ... (* 3 1))
```

```
(λ (redex) (+ redex (* 3 1)))
```

Note the similarity...

```
(+ ... (* 3 1))
```

```
((λ (redex) (+ redex (* 3 1))) ...)
```

Note the similarity...

```
(+ (+ 1 2) (* 3 1))
```

Evaluating the redex inside the continuation is akin to calling a function.

```
((λ (redex) (+ redex (* 3 1))) (+ 1 2))
```

Reified continuations

```
((lambda (x) (+ x (* 3 1))) (+ 1 2))
```

A redex's continuation is said to be **reified** if it is transformed into an expression that the redex has access to.

Passing the continuation to the redex

```
((lambda (k)
     (k (+ 1 2)))
   (lambda (x) (+ x (* 3 1))))
```

A redex's continuation is said to be **reified** if it is transformed into an expression that the redex has access to.

But, given an expression, how do we get the continuation (as a function) while evaling the redex (as an expression)?

```
(+ (+ 1 2) (* 3 1))
```

The answer: the **shift operation**

shift

Shift is a special form of two or more arguments:

```
(shift <id> <redex-expr> ...)
```

Here's an example of it in use: It's a subexpression in our larger computation now.

```
(+ (shift k (+ 1 2)) (* 3 1))
```

shift

```
(+ (shift k (+ 1 2)) (* 3 1))
```

k is bound to the continuation
of the expression

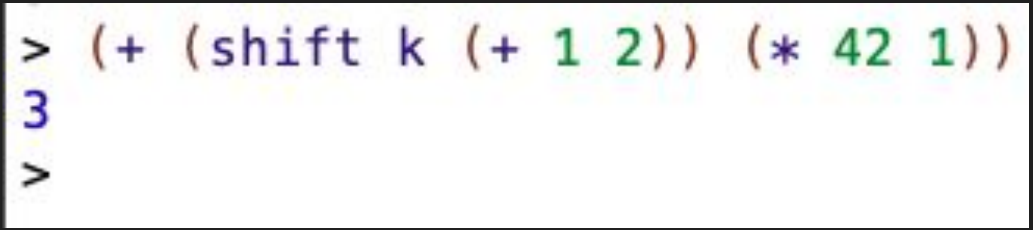
```
(lambda (x) (+ x (* 3 1))
```

shift

```
(+ (shift k (+ 1 2)) (* 3 1))
```

when the shift form is evaluated, the redex expression is evaled (with k bound in the environment…)

...but what's produced by this whole expression?

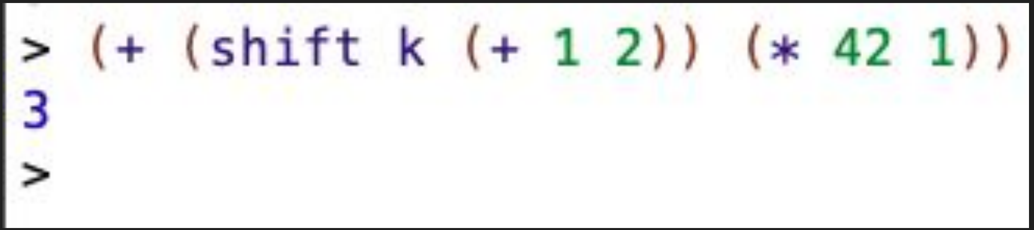


```
> (+ (shift k (+ 1 2)) (* 42 1))
3
>
```

What happened to the rest of the expression? It's not evaluated, but simply discarded.

Where else have we seen "return something early" in imperative languages?

- A return statement before the final statement in a function/method
- Raising an exception!



```
> (+ (shift k (+ 1 2)) (* 42 1))
3
>
```

What happened to the rest of the expression? It's not evaluated, but simply discarded.

In this example, how can we execute the rest of the expression, like before? We call the continuation function k!

reset: the dual of shift

Shift is a special form of one argument

```
(reset <expr-with-a-shift>)
```

Reset **limits where the continuation extends to** in the expression;

- Only expressions **inside** the reset are bound to the continuation
- Expressions **outside** the reset are evaluated like normal.

```
(* 2 (reset (+ 42 (shift k (+ 1 2)))))
```

reset: the dual of shift

When we evaluate the shift, we'll make the continuation `(λ (x) (+ 42 x))`

```
(* 2 (reset (+ 42 (shift k (+ 1 2)))))
```

reset: the dual of shift

We discarded k in this example, and the (* 2 …) was outside the reset, so it remains

```
(* 2 (reset (+ 42 (shift k (+ 1 2)))))
```

```
(* 2 (+ 1 2))
```

reset: the dual of shift

And then evaluation proceeds as normal.

```
(* 2 (reset (+ 42 (shift k (+ 1 2)))))

(* 2 (+ 1 2))

(* 2 3)
```

reset: the dual of shift

Tada!

```
(* 2 (reset (+ 42 (shift k (+ 1 2)))))

(* 2 (+ 1 2))

(* 2 3)

6
```

Capturing a continuation

What does this expression evaluate to?

```
(+ 1 (* 2 (shift k k)))
```

Capturing a continuation

What does this expression evaluate to?

```
(+ 1 (* 2 (shift k k))))
```

Any time you see a shift, the answer is "it depends on the surrounding context"!!!

Capturing a continuation

OK, how about *this* expression?

```
(reset (+ 1 (* 2 (shift k k))))
```

By adding the (reset …), we know that k will only capture the `(* 2 (+ 21 …)` continuation

Who cares?

We've seen how

- `shift` captures an expression's surrounding expression
- `reset` delimits how far `shift`'s capture goes

OK, Nathan, but who cares? What can we do with this?

Who cares?

We've seen how

- `shift` captures an expression's surrounding expression
- `reset` delimits how far `shift`'s capture goes

OK, Nathan, but who cares? What can we do with this?

- Because we can discard a continuation k by not calling it, this lets us abort the evaluation of an expression, a bit like raising an exception
- Because k is an ordinary function, we can bind k to an identifier and reuse the continuation later on!