CSC324 Principles of Programming Languages

Lecture 6

October 23, 2019

The story so far...

- We learned how to solve problems using **functional programming**, using features like tail recursion, currying, etc.
- We built **interpreters** for languages based on the Lambda Calculus (calculator, Dubdub), and explored the difference between...
  - Eager vs lazy evaluation
  - Lexical vs dynamic scoping
- We use **macros** to write code that manipulates code, to ...
  - Create new syntax for object-oriented programming
  - Create streams

Next few weeks

- We will build up to **logic programming** using macros
- We will introduce the idea of **continuations**, and use continuations to solve problems
- We will program in Haskell, and explore Haskell's **type system** and more

Today

- Review streams
- Self-modifying streams
- Haskell (for Exercise 6)

Haskell

Exercise 6

- Exercise 6 will begin introducing Haskell

Haskell Pattern Matching

Recall from week 2:

```
add1_list []     = []
add1_list (x:xs) = (1 + x):(add1_list xs)
```

Pattern Matching in Your Exercise

You can (and should!) use pattern matching in exercise 6

```
numEvens :: [Int] -> Int
numEvens [] = undefined                  -- base case
numEvens (first:rest) = undefined  -- recursive case
```

Haskell is *strongly typed*. We'll explore what that means in week 9.

Haskell Type Signatures

Values and functions can have type signatures:

```
celsiusToFahrenheit :: Float -> Int
celsiusToFahrenheit temp = undefined -- todo
```

Haskell Data Type

Example: creating our own **List** type

```
data List = Empty
          | Cons Integer List
          deriving (Show, Eq)
```

- **List** is the name of the data type
- **Empty** is a value representing an empty List,
- **Cons** is a *constructor* function that takes two arguments, an integer and a List, and returns a new List

Ignore the `deriving (Show, Eq)` for now

Examples of List

```
data List = Empty
          | Cons Integer List
          deriving (Show, Eq)
```

- Empty
- (Cons 3 (Cons 4 (Cons 5 Empty)))
- (Cons -23 Empty)

Haskell Data Type: Calculator

```
data Expr = Number Float
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          deriving (Show, Eq)
```

- **Expr** is the name of the data type
- **Number**, **Add**, **Sub**, ... are *constructor* functions that constructor data of type Expr

Pattern Matching on Data Constructors

```
calculate :: Expr -> Float
calculate (Number n)          = undefined --todo
calculate (Add expr1 expr2) = undefined --todo
...
```

Streams

Why Streams?

Streams are lists that are lazily evaluated.

Streams allows us to work with infinite lists.

Example in Haskell:

```
Prelude> lst = [4..]
Prelude> take 20 lst
[4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23]
```

Stream Definition

```
(define s-null 's-null)
(define (s-null? stream) (equal? stream s-null))

(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))

(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))

(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))
```

Worksheet

Write a function `s-take` that takes the first `n` elements of a `stream` and produces a list with those `n` elements.

Worksheet

Write a function `repeat` that takes an integer `n` and returns the infinite stream where each element is `n`.

```
Prelude> myRepeat x = x : myRepeat x
Prelude> take 3 (myRepeat 6)
[6,6,6]
Prelude> take 10 (myRepeat 6)
[6,6,6,6,6,6,6,6,6,6]
```

Worksheet

Write a function `s-append` that append two streams `s` and `t`

This is analogous to the `append` function for lists.

Python Generator

```
>>> vals = (x for x in [1, 2, 3]) # create generator
>>> vals
<generator object <genexpr> at 0x000001BDB5E6DF68>
>>> next(vals) # access one element at a time
1
>>> next(vals)
2
>>> next(vals)
3
>>> next(vals) # no more elements
StopIteration
```

Python Generator Function

```
def gen():
    for i in [3, 2, 1, 0]:
        yield 12 / i

>>> xs = gen()
>>> next(xs)
4
>>> next(xs)
6
>>> next(xs)
12
>>> next(xs)
ZeroDivisionError
```

Infinite Iterator in Python

```
define repeat(n):
    while True:
        yield n

>>> g = repeat(3)
>>> next(g)
3
>>> next(g)
d3
```

Desired Self-Updating Stream in Racket

```
> (define s (make-stream 1 2 3))
> (next! s)
1
> (next! s)
2
> (next! s)
3
> (next! s)
'DONE
> s
's-null
```

Mutation!

The exclamation mark `!` at the end of `next!` is pronounced "bang"

The most basic mutation function in Racket is `set!`

```
>>> (define x 4)
>>> (set! x 2)
>>> x
2
```

Behaviour of `next!`

The syntax `next!` takes a stream, and if the stream is non-empty:

- Update the stream to `s-rest` of the stream
- Return the `s-first` of the stream

If the stream is empty, return 'DONE.

Behaviour of `next!` on non-empty stream

```
(let* ([tmp s])
  (begin
    (set! s (s-rest s))
    (s-first tmp)))
```

Exercise: Implement `next!` as a macro

```
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)

    ]))
```

The macro next!

```
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)
     (if (s-null? <g>)
         'DONE
         (let* ([tmp <g>])
           (begin
             (set! <g> (s-rest <g>))
             (s-first tmp))))]))
```

Next class

In the next class, we'll create an operator `-<` (pronounced "amb") that behaves like this:

```
> (define g (-< 1 2 (+ 3 4)))
> (next! g)
1
> (next! g)
2
> (next! g)
7
> (next! g)
'DONE
```

But isn't `-<` just `make-stream`?

```
> (define g (make-stream 1 2 (+ 3 4)))
> (next! g)
1
> (next! g)
2
> (next! g)
7
> (next! g)
'DONE
```

The Ambiguous Choice Operator -<

The -< operator will also *capture the surrounding computation* so that an expression like:

```
(+ 10 (-< 1 2 (+ 3 4)))
```

. . . would create a stream with the values:

- (+ 10 1)
- (+ 10 2)
- (+ 10 (+ 3 4))