HW 3 Recap

- Unnecessary argument to helper function
  E.g. Problem 9
  ```
  fun all_answers f =
    let fun helper (f, xs, acc) = …
  f already accessible in helper, and doesn’t change recursively!
  ```

Streams

- A stream is an infinite sequence of values
  - So cannot make a stream by making all the values
  - Key idea: Use a thunk to delay creating most of the sequence
  - Just a programming idiom
- A powerful concept for division of labor:
  - Stream producer knows how to create any number of values
  - Stream consumer decides how many values to ask for

Using Streams

We will represent streams using pairs and thunks

Let a stream be a thunk that when called returns a pair:

```
'(next-answer . next-thunk)
```

So given a stream \(s\), the client can get any number of elements

- First: \((\text{car} \ (s))\)
- Second: \((\text{car} \ ((\text{cdr} \ (s))))\)
- Third: \((\text{car} \ ((\text{cdr} \ ((\text{cdr} \ (s)))))))\)

(Usually bind \((\text{cdr} \ (s))\) to a variable or pass to a recursive function)

Streams: Example

```scheme
(define nats
  (letrec ([f (lambda (x)
               (cons x (lambda () (f (+ x 1)))))]
           (lambda () (f 1)))))
```

Q: How would you get the second number in this stream and save it as a variable \(x\)?
Streams: Non-example

Q: Why are each of these wrong?

Example using streams

This function returns how many stream elements it takes to find one for which tester does not return #f

Q: (define nats2)

(letrec ([f (lambda (x) (cons x (lambda () (f (+ x 1)))))])

(f 1)))

Q: (define nat3)

(letrec ([f (lambda (x) (cons x (f (+ x 1)))))

(lambd (f 1)))])

Practice with Streams

Worksheet questions Q1, Q2, Q3

What is a macro

• A macro definition describes how to transform some new syntax into different syntax in the source language

• A macro is one way to implement syntactic sugar

• A macro system is a language (or part of a larger language) for defining macros

• Macro expansion is the process of rewriting the syntax for each macro use

Example Racket macro definitions

Two simple macros

(define-syntax my-if
  (syntax-rules (then else)
    [(my-if e1 then e2 else e3) (if e1 e2 e3)])
)

(define-syntax comment-out
  (syntax-rules ()
    [(comment-out ignore instead) instead]
)

If the form of the use matches, do the corresponding expansion

• In these examples, list of possible use forms has length 1

Example uses

It is like we added keywords to our language

• Other keywords only keywords in uses of that macro

• Syntax error if keywords misused

• Rewriting ("expansion") happens before execution

Winter 2019 CSE341: Programming Languages
Practice with Macros

Define a macro `my-and` and `my-or` that take two expressions and do the equivalent things. (Do not use `and/or`, use `my-if`) (e.g. `(my-and e1 e2) == (and e1 e2))

```scheme
(define-syntax my-and
  (syntax-rules ()
    [(my-and e1 e2)
      (my-if e1 then e2 else #f)]))

(define-syntax my-or
  (syntax-rules ()
    [(my-or e1 e2)
      (my-if e1 then #t else e2)]))
```