PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Section 7

HW3 Recap, Streams, Macros

Spring 2020

HW 3 Recap

- Unnecessary function wrapping

E.g. Problem 12

```
(fn x => String.size(x)) vs String.size
```

HW 3 Recap

- Unnecessary argument to helper function

E.g. Problem 9

```
fun all_answers f l =
    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: (**`car (s))`**
- Second: **`(car ((cdr (s))))`**
- Third: **`(car ((cdr ((cdr (s))))))`**

(Usually bind **`(cdr (s))`** to a variable or pass to a recursive function)

Streams: Example

```
(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

```
(define nats2
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (+ x 1)))))])
    (f 1)))
```

```
(define nat3
  (letrec ([f (lambda (x)
                (cons x (f (+ x 1))))])
    (lambda () (f 1))))
```

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`**

- Happens to be written with a tail-recursive helper function

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
                (let ([pr (stream)])
                   (if (tester (car pr))
                       ans
                       (f (cdr pr) (+ ans 1)))))])
     (f stream 1)))
```

- **`(stream)`** generates the pair
- So recursively pass **`(cdr pr)`**, the thunk for the rest of the infinite sequence

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
  - "Replace any syntax of the form **`e1 andalso e2`** with **`if e1 then e2 else false`**"
- 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*
  - Before a program is run (or even compiled)

Example Racket macro definitions

Two simple macros

```
(define-syntax my-if                 ; macro name
  (syntax-rules (then else)          ; other keywords
    [(my-if e1 then e2 else e3)      ; macro use
     (if e1 e2 e3)]))               ; form of expansion
```

```
(define-syntax comment-out           ; macro name
  (syntax-rules ()                   ; other keywords
    [(comment-out ignore instead)    ; macro use
     instead]))                     ; form of expansion
```

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

- In these examples, list of possible use forms has length 1
- Else syntax error

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

```
(my-if x then y else z)   ; (if x y z)
(my-if x then y then z)   ; syntax error

(comment-out (car null) #f)
```

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))

```
(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)]))
```