

## Reading

7. Wednesday 2 Feb: Davie<sup>1</sup>: Section 3.5; Gentle Intro: Sections 3.2.1, 3.3
8. Friday 4 Feb: Davie: Sections 2.3, 2.5, 2.7.1, 2.9, 3.2  
Gentle Intro: Sections 3.3, 3.5, 4.1, 4.2
9. Monday 7 Feb: Davie, Sections 3.4, 4.1–4.3, 4.8
10. Wednesday 9 Feb: Davie, Sections 2.8, 2.10, 2.11, 3.11  
Gentle Intro: Section 2.4.1, 2.4.2

**Note** on readings in Davie: Don't count on accuracy in every detail. He's not using Hugs, and there are differences.

1. His interpreter's command-line prompt is '>', which makes it easy to confuse his command-line expressions with Hugs98's literate scripts.
2. His interpreter allows multi-line commands, which Hugs98 does not.
3. Hugs98 does not support adding definitions via the command line (as described in Section 2.11).

## Exercises

**Due date:** Tuesday 8 February. Please leave your paper in the Taylor Hall homework box (it's located in the breezeway between 2.132 and 2.136) **by 4pm**.

1. [15] Show all possible sequences of steps in the evaluation of the following expression. Assume that all operators are left-associative, that (+) and (-) have precedence 1, and that (\*) and (/) have precedence 2.

$$10 + 18 * 3 - 81 / 9$$

2. [5] Show the steps in a stack evaluation of the following expression:

$$24 \ 16 + 10 - 18 \ 3 * 81 \ 9 / + +$$

3. [10] Give a sequence of trees corresponding to the innermost evaluation of the following expressions. In parts (b) and (c), assume that (| |) is defined nonstrictly (see Slide 64).

a.  $4 * 7 - 3 * 6$

b.  $2 < 5 \ || \ 4 \leq 7$

c.  $2 = 5 \ || \ 4 < 7$

4. [10] Haskell's **if** and **case** expressions can be translated into each other.

- a. Translate the expression

**if** e **then** x **else** y

into a **case**-expression.

- b. Translate the expression

```
case collegeCode of
  "E" -> "Nat Sci"
  "L" -> "Liberal Arts"
  "2" -> "Business"
  "4" -> "Engineering"
  "5" -> "Fine Arts"
  "6" -> "Grad"
  _   -> collegeCode
```

into an **if** expression.

5. [25] This exercise concerns the formula for computing the area of a triangle given the lengths of its sides  $a$ ,  $b$ , and  $c$ :

$$\sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)} \quad \text{where} \quad s = \frac{a + b + c}{2}$$

- a. Write a Haskell expression for the area of a particular triangle whose sides are 9, 4, and 11, and evaluate it using Hugs. Use a Haskell **let**-expression to define values for  $a$ ,  $b$ , and  $c$ , and to define  $s$  in terms of them. Also use the Prelude function `sqrt`.

<sup>1</sup> "Davie" refers to *Readings from "Intro to Functional Programming Systems Using Haskell"*.

- b.** Generalize the expression you wrote in part (a) to make a lambda expression that can be applied to the lengths of a triangle's three sides to compute its area. Demonstrate it on the triangle of part (a), and on two other triangles of your choosing.
- c.** Give the function you defined in Part (b) a name; you'll need to put the name's definition in a script. Load the script, and demonstrate the named function on the examples of Part (b).
- d.** Redefine the function of Part (c) using the standard Haskell function-definition form (slide 57), and demonstrate that it gives the same results as the  $\lambda$ -expression version.
- e.** Redefine the function of Part (d) using **where** instead of **let**, and demonstrate that it gives the same results as the **let** version.
- 6.** [15] Show the steps in the evaluation of the expression
- ```
let f x y = if x > 0 then x else x * y in f (3 + 7) (f 8 4)
```
- For each evaluation policy, count the number of addition operations and the number of comparisons. In each step, underline the subexpression evaluated in the next step (as in Slide 53).
- a.** Use the leftmost-*innermost* policy.
- b.** Use the *outermost* policy.
- c.** Use the *lazy* policy.
- 7.** [10] Show the steps in the evaluation of the expression
- ```
let f x y = if x > 0 then x else x * y
    loop = 2*loop
in f 9 loop
```
- a.** Use the leftmost-*innermost* evaluation policy.
- b.** Use the leftmost-*outermost* evaluation policy.
- 8.** [10] Perform an experiment to determine whether, and in which arguments, the Haskell operators (&&) and (||) are strict. The answers can be obtained by evaluating Hugs98 command-line expressions.
- 9.** [15] Redefine each of the following functions using only conditional (i.e., **if...then...else**) expressions.
- a.** fa x y
- ```
| x > y    = x + 2*y
| x < y    = fa (2*x) (y `div` 2)
| otherwise = 7
```
- b.** fb 0 = 1  
fb 1 = 1  
fb n = n + fb (n-1)
- c.** fc 0 = 1  
fc 1 = 1  
fc n | n>0 = n + fc (n-1)
- 10.** [15] Redefine the following functions
- ```
even, odd : Int -> Bool
even n = if n==0 then True else odd (n-1)
odd n = if n==0 then False else even (n-1)
```
- a.** using guards only.
- b.** using patterns as much as possible, and guards where patterns can't be used.
- c.** using **case**-expressions.

- 11.** [45] This exercise illustrates how changing a given function's implementation can affect its time and space performance.

Our example is the exponentiation function, which raises a number to a positive integer power. Here is a very simple implementation:

```
> power :: Integer -> Integer -> Integer
> power a 0 = 1
> power a b = a * power a (b-1)
```

- a.** [10] Show the steps in the evaluation of the expression `power 2 7`. How do the time and maximum space required by the evaluation of `power n k` depend on `n` and `k`?
- b.** [5] Experiment with Hugs to determine for approximately ( $\pm 100$ ) how large a value of `k` the expression `power 2 k` can be evaluated. The symptoms of failure vary somewhat depending on the platform on which you're running Hugs— Hugs may report a stack overflow, or it may crash.
- c.** [10] The cause of the limitation is the accumulation of pending multiplications, which is inherent in the way `power` is defined— each multiplication has to wait until its right argument has been evaluated.

One way to eliminate the multiplication delay is to convert the function's definition to a tail-recursive implementation:

```
> trPower :: Integer -> Integer -> Integer
> trPower a b = trp b 1
>   where
>     trp 0 p = p
>     trp n p = trp (n-1) (a*p)
```

Show the steps in the evaluation of the expression `trPower 2 7`, making sure not to reduce any subexpression prematurely.

Does the space problem appear to be solved?

- d.** [5] Repeat the experiment of part (b) for `trPower`.
- e.** [10] Modify `trPower` to force the accumulator argument to be evaluated prematurely, using the operator `($!)` as described in Slide 65.
- Repeat the calculation of part (c) using your modified version of `trPower`.
- f.** [5] Repeat the experiment of part (b) using your modified version of `trPower`, demonstrating that Hugs no longer crashes.
- 12.** [20] In Exercise 11 we improved `power`'s space performance, but all three versions take time linear in the exponent argument.
- a.** [5] Confirm this by running `power`, `trPower`, and `trPowerS` on the largest exponent for which you've discovered that all three succeed. Have Hugs count the number of reduction steps by turning on the statistics flag:

```
CS345HW2> :s +s
```

- b.** [5] Here is an algorithm which improves both time and space performance:

```
> turboPower a 0 = 1
> turboPower a b
>   | even b   = turboPower (a*a) (b `div` 2)
>   | otherwise = a * turboPower a (b-1)
```

Instead of merely decrementing its first argument, this algorithm halves it whenever it is even, thereby reaching termination much more quickly. Demonstrate it on the same arguments you used in part (a).

- c.** [10] Repeat the calculation of Exercise 11c using `turboPower`. You do not need to show the evaluation of the guard expression `even b`. The recursive applications of `turboPower` give rise to multiple instances of the parameter `a`; handle this by adding digits or prime characters (`'`) to distinguish between them.

# CS 345 Richards

## Assignment 2

Authors:

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

4. \_\_\_\_\_