15-150 Fall 2013
Homework 05

Out: Wednesday, 25 September 2013
Due: Tuesday, 1 October 2013 at 23:59 EST

1 Introduction

This homework will focus on applications of higher order functions, polymorphism, and user-defined datatypes.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/05` directory should contain a file named exactly `hw05.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/05` directory (that contains a `code` folder and a file `hw05.pdf`). This should produce a file `hw05.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw05.tar` file via the “Handin your work” link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the “check” section of your latest handin on the “Handin History” page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

Your `hw05.sml` file must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 1 October 2013 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

2 Types and Polymorphism

In class we discussed typing rules. In particular:

- A function expression `fn x => e` has type `t -> t'` if and only if, by assuming that `x` has type `t`, we can show that `e` has type `t'`.
- An application e_1 e_2 has type `t'` if and only if there is a type `t` such that e_1 has type `t -> t'` and e_2 has type `t`.
- An expression can be used at any instance of its most general type.

Task 2.1 (4%). Consider the following ML function declaration:

```
fun all (your, base) =
    case your of
        0 => base
      | _ => "are belong to us" :: all(your - 1, base)
```

What is the most general type of `all`?

$\boxed{\textbf{Solution 2.1}}$ The most general type of `all` is `int * string list -> string list`. We perform integer checks on `your` so `your` must be of type `int`. `base` must be a list of some sort as we cons an element on to it, and it must be a `string list` because we're consing a string.

Task 2.2 (2%). Consider a different ML function:

```
fun funny (f, []) = 0
  | funny (f, x::xs) = f(x, funny(f, xs))
```

What is the most general type of `funny`?

$\boxed{\textbf{Solution 2.2}}$ The most general type of `funny` is `('a * int -> int) * 'a list -> int`.
Most generally, we know `f : 'a * 'b -> 'c` and `x::xs : 'd list`. As we see in the first case, `funny` returns an `int`, so the overall return type of `funny` must be `int`. Then, we narrow down the type of `f`. It accepts as its first argument an `x`, so it must accept the same type as the type of elements in the list which we'll now call `'a`, so `x::xs : 'a list` and `f : 'a * 'b -> 'c`. Now, we know `f` must have the same return type as `funny`, so `f : 'a * 'b -> int`. Lastly, the second argument to `f` is the return value of `funny`, so that too much be an `int`. Hence, `funny : ('a * int -> int) * 'a list -> int`.

Task 2.3 (2%). Now consider the following function expression:

```
fn x => (fn y => x)
```

What is its most general type?

Solution 2.3 The most general type of `fn x => (fn y => x)` is `'a -> 'b -> 'a`.
We will start with the most general type that this function can have. `x` has type `'a` and `y` has type `'b`. There is nothing in the function that forces x or y to have a nonpolymorphic type. Thus, `fn x => (fn y => x)` has type `'a -> 'b -> 'a`, since the function takes an input of `'a` and gives back a function that takes an input of `'b` and evaluates to `x` (defined to be of type `'a`).

Task 2.4 (2%). Now look at this slightly different expression:

```
(fn x => (fn y => x)) "Hello, World!"
```

What is the most general type of *this* expression?

Solution 2.4 The most general type is `'a -> string`. We know from the previous problem that the type of the two anonymous functions is `'a -> 'b -> 'a`. We then apply a value of type `string` to that function, fixing `'a` to be `string`, so now we have a new expression of type `'a -> string`.

3 Bases

The digits we use to represent a number depend on the base we use. We usually write numbers in base ten, or using digits from 0 to 9. More generally, numbers in a particular base b can use digits between 0 and $b-1$, inclusive. We can notate this base with a subscript, so the number fifty-four is 54_{10}. In computer science we also commonly see base 2, or binary numbers, which count with 1s and 0s. For example, $10_2 = 2_{10}$, $11_2 = 3_{10}$, and so on.

We'll be representing numbers in different bases as an `int list` of digits with the least significant digit as the first element of the list, so 1100_2 is `[0, 0, 1, 1]`.

3.1 Converting to an `int`

Formally, given a base b and a string of n digits $d_n d_{n-1} \ldots d_1$, the numerical value of the string is

$$\sum_{i=1}^{n} b^{i-1} d_i$$

For example, 34_{10} is $3 * 10^1 + 4 * 10^0 = 3 * 10 + 4 * 1 = 34$.
We look at the same example in base 2: $100010_2 = 1 * 2^5 + 1 * 2^1 = 1 * 32 + 1 * 2 = 34$. Given this formula, we can take any string of digits in a base and convert the digits to an `int`.

Task 3.1 (5%). Define the higher-order function

```
toInt : int -> int list -> int
```

such that for all `b` > 1 and all `L : int list`, if `L` is a list of base `b` digits, then `toInt b L` is the corresponding integer value n. Note that `toInt` is higher-order, so `toInt b` should return a function from `int list` to `int`. For example,

```
val base2ToInt = toInt 2
val 2 = base2ToInt [0, 1]
```

Solution 3.1 See solution in `hw05.sml`.

3.2 Converting from an `int`

For any natural number n and a base b, we can convert n into base b with the following algorithm:

1. If $n = 0$, then stop. You're done.
2. Find the remainder of n divided by b. Prepend this to our current list of digits.
3. Do an integer division of n by b.
4. Go back to step 1 using $n = n$ `div` b.

For example, this is the process of converting 42_{10} to base 5.

$$\begin{aligned} 42 \text{ mod } 5 &= 2 & [42 \text{ div } 5 = 8] \\ 8 \text{ mod } 5 &= 3 & [8 \text{ div } 5 = 1] \\ 1 \text{ mod } 5 &= 1 & [1 \text{ div } 5 = 0] \end{aligned}$$

Reading from the bottom up, we get $42_{10} = 132_5$, which we can confirm using the formula for converting to base 10. $1 * 5^2 + 3 * 5 + 2 = 25 + 15 + 2 = 42$.

Task 3.2 (5%). Define the higher order function

```
toBase : int -> int -> int list
```

such that for all `b` > 1, `n` ≥ 0, `toBase b n` returns the representation of `n` in base `b`. Again, `toBase` is higher-order, so `toBase b` should return a function. For example,

```
val toBase3 = toBase 3
val [2, 1] = toBase3 5
```

Solution 3.2 See solution in `hw05.sml`.

Task 3.3 (5%). Define the higher order function

```
convert : int * int -> int list -> int list
```

such that for all `b1, b2` > 1 and for all `L : int list` such that `L` is a list of base `b1` digits, `convert (b1, b2) L` changes the representation of the input number from base `b1` to base `b2`. We should have `toInt b2 (convert(b1, b2) L) = toInt b1 L` hold for `convert`. You may use `toInt` and `toBase` in your solution.

Solution 3.3 See solution in `hw05.sml`.

4 Higher-Order Functions

Recently we introduced several new language features: polymorphism, option types, and higher-order functions. In the next problems, you will write some simple functions using these new tools.

You'll start by writing functions on vectors. Vectors of length n are essentially lists of numbers which indicate a direction and a magnitude in $\mathbb{R}^n$. For example, vectors in $\mathbb{R}^2$ are line segments from the origin to the (x, y) point they contain. We will represent vectors in SML as `real lists`. Mathematically, then, a vector $\langle a_1, a_2, ..., a_n \rangle$ translates to $[a_1, a_2, ..., a_n]$.

4.1 Dot product

Recall : To calculate the dot product of vectors $\vec{a}$ and $\vec{b}$

$$\vec{a} \cdot \vec{b} = \langle a_1, a_2, ..., a_n \rangle \cdot \langle b_1, b_2, ..., b_n \rangle = (a_1 * b_1) + (a_2 * b_2) + ... + (a_n * b_n)$$

Task 4.1 (4%). Write the function

```
dotProduct : real list * real list -> real
```

that calculates the dot product of two vectors of the **same length**. You solution should be **non-recursive** and it should instead use higher order functions to accomplish its goal. **Hint:** You can use

```
zip: 'a list * 'b list -> ('a * 'b) list
```

Solution 4.1 See solution in `hw05.sml`.

4.2 Magnitude

Recall : The magnitude of a vector $\vec{a}$, denoted by $||\vec{a}||$, is

$$||\vec{a}|| = ||\langle a_1, a_2, ..., a_n \rangle|| = \sqrt{a_1^2 + a_2^2 + ... + a_n^2}$$

Task 4.2 (4%). Write the function

```
magnitudeOfVector : real list -> real
```

that calculates the magnitude of a given vector. You solution should be non-recursive and should again use higher order functions. Note that you can use

```
Math.sqrt : real -> real
```

to calculate the square root of a real number.

Solution 4.2 See solution in `hw05.sml`.

4.3 Angle

Now that we can calculate the dot product of two vectors and their magnitudes, we can also calculate the angle between them.

Recall : The angle between two vectors $\vec{a}$ and $\vec{b}$ is

$$\theta = \cos^{-1}\left(\frac{\vec{a}\cdot\vec{b}}{||\vec{a}|| * ||\vec{b}||}\right)$$

Task 4.3 (5%). Write the function

```
angleBetweenVectors : real list * real list -> real
```

that calculates the angle between given vectors. Please note that you can use

```
Math.acos : real -> real
```

to calculate the inverse cosine of a real number.

Solution 4.3 See solution in `hw05.sml`.

4.4 Extract

Task 4.4 (7%). Write a function

```
fun extract (p : 'a -> bool, l : 'a list) : ('a * 'a list) option =
```

such that

1. If there is some element `x` of `l` for which `p x` = `true`, then `extract(p,l)` evaluates to `SOME(x,l')`, where `l'` is `l` without that particular `x` but unchanged otherwise.
2. If for every element `x` of `l`, `p x` = `false` then `extract(p,l)` evaluates to `NONE`.

If there is more than one element satisfying the predicate in a particular argument list, it is your choice which to return.

For example:

```
extract(oddP , [2,3,4]) = SOME (3,[2,4])
extract(oddP , [2,4,6]) = NONE
extract(fn x => String.size x < 2 , ["aaa","b","bca"])
                          = SOME ("b", ["aaa", "bca"])
```

`extract` should be recursive. You should use this function when you implement Blocks World below.

Solution 4.4 See solution in `hw05.sml`.

5 Blocks World

In artificial intelligence, *planning* is the task of figuring what an agent (a robot, that paperclip in Microsoft Word, your roommate, etc.) should do. One way to solve planning problems is to simulate the circumstances of the agent, so that you can simulate plans, and then search through potential plans for good ones.

A simple planning problem, which is often used to illustrate this idea, is *blocks world*. The idea is that there are a bunch of blocks on a table:

```
---  ---  ---
|A|  |B|  |C|
---  ---  ---
-------------------------
```

and a robotic hand. You can pick one block up with the hand:

```
/|\
---
|C|
---
     ---  ---
     |A|  |B|
     ---  ---
-------------------------
```

and place it back on the table or on another block:

```
---
|C|
---
---  ---
|A|  |B|
---  ---
-------------------------
```

Of course, you can't put a block on one that already has something on it, so in the next two moves we can't pick up `B` and then put it on `A`. A planning problem would be something like "starting with the blocks on the table, make the tower `BCA`".

In this problem, you will represent blocks world in ML, so that you can simulate plans (we won't ask you to search for plans that achieve specific goals).

At the end of the problem, you'll be able to interact with Blocks World as in the figure above. We've written all the input/output code for you, so you just need to do the interesting bits.

```
- playBlocks ();

Possible moves:
  pickup <block> from table
  put <block> on table
  pickup <block> from <block>
  put <block> on <block>
  quit

---   ---   ---
|A|   |B|   |C|
---   ---   ---
-------------------------
Next move: pickup C from table

/|\
---
|C|
---
      ---   ---
      |A|   |B|
      ---   ---
-------------------------
Next move: put C on A

---
|C|
---
---   ---
|A|   |B|
---   ---
-------------------------
```

Figure 1: Sample Blocks World Interaction

5.1 Rules

We will model Blocks World as follows:

- There are three blocks, A, B, C.
- We will represent the state of the world as a list of facts. There are five kinds of facts:
  - Block b is free (available to be picked up)
  - Block a is on block b
  - Block a is on the table
  - The hand is empty
  - The hand is holding block b
- At each step, there are four possible moves:

  ```
  pickup <b> from table
  put <b> on table
  pickup <a> from <b>
  put <a> on <b>
  ```

  These moves act as follows:

  - `pickup <a> from table`
    Before: a is free, and a is on the table, and the hand is empty.
    After: the hand holds a.
  - `put <b> on table`
    Before: the hand holds b.
    After: the hand is empty, and b is on the table, and b is free.
  - `pickup <a> from <b>`
    Before: a is free, and a is on b, and the hand is empty.
    After: b is free, and the hand is holding a.
  - `put <a> on <b>`
    Before: the hand holds a, and and b free.
    After: a is free, the hand is empty, and a is on b.

  In these descriptions, the "before" facts must hold about the world for the move to be executed; after executing the move, the "before" facts no longer hold (e.g. after picking up a block, the hand is no longer empty), and the "after" facts holds.

5.2 Tasks

Task 5.1 (5%). First, we will need a function to extract many elements from a list. Write a function

```
extractMany : ('a * 'a -> bool * 'a list * 'a list) -> ('a list) option
```

`extractMany` is polymorphic in the list's element type, but it needs to test whether two list elements are equal. For this reason, `extractMany` takes an argument function `eq:'a * 'a -> bool` that can be used to test whether two values of type `'a` are equal.

`extractMany (eq,toExtract,from)` "subtracts" the elements of `toExtract` from `from`, checking that all the elements of `toExtract` are present in `from`. More formally, if `toExtract` is a sub-multi-set (according to the definition given in the subset-sum problem on HW 3, but using `eq` to determine when an element "appears") of `from`, then `extractMany(eq,toExtract,from)` returns `SOME xs`, where `xs` is `from` with every element of `toExtract` removed. If `toExtract` is not a sub-multi-set of `from`, then `extractMany(eq,toExtract,from)` returns `NONE`.

This means that the number of times an element occurs matters, but order does not:

```
extractMany(inteq, [2,1,2], [1,2,3,3,2,4,2]) = SOME [3,3,4,2]
extractMany(inteq, [2,2], [2]) = NONE
```

You may define this recursively, and should use `extract`.

Solution 5.1 See solution in `hw05.sml`.

Task 5.2 (8%). Define datatypes representing blocks, moves, and facts, according to the above rules:

```
datatype block = ...
datatype move = ...
datatype fact = ...
```

Observe the convention that datatype constructors start with an upper-case letter (e.g. `Node` and `Empty`).

Solution 5.2 See solution in `hw05.sml`.

Task 5.3 (2%). Define a `state` of the world to be a list of facts:

```
type state = fact list
```

Fill in

```
val initial : state = ...
```

to represent the following state: the hand is empty, each of A,B,C is on the table, and each of A,B,C is free.

Solution 5.3 See solution in `hw05.sml`.

Task 5.4 (3%). Define a short helper function

```
consumeAndAdd : (state * fact list * fact list) -> state option
```

`consumeAndAdd(s,before,after)` subtracts `before` from `s` and adds `after` to the result, checking that every fact in `before` occurs. More formally, if `before` is a sub-multi-set of `s`, then `consumeAndAdd(s, before, after)` returns `SOME s'`, where `s'` is `s` with `before` removed and `after` added. If `before` is not a sub-multi-set, `consumeAndAdd(s, before, after)` returns `NONE`.

You will need to use the provided function `extractManyFacts`, which instantiates your `extractMany` with an equality operation derived from the `fact` datatype.

`consumeAndAdd` should not be recursive.

Solution 5.4 See solution in `hw05.sml`.

Task 5.5 (7%). Implement a function

```
step : (move * state) -> state option
```

If the "before" facts of `m` hold in `s`, then `step(m,s)` must return `SOME s'`, where `s'` is the collection of facts resulting from performing the move `m`. It should return `NONE` if the move cannot be applied in that state. This function should not be recursive.

Solution 5.5 See solution in `hw05.sml`.

Task 5.6 Optional: In the file `blocks_world.sml`, fill in your datatype constructors at the spots indicated. You will then be able to play Blocks World interactively as follows:

```
- use "hw05.sml";
- use "blocks_world.sml";
- playBlocks();
```

This task is optional; do not hand in `blocks_world.sml`.

Task 5.7 EXTREMELY OPTIONAL CHALLENGE TASK:

If you're really really bored, here's something fun you can try.

The text-based interface we made for blocks world works but is kind of bland. Download a graphics library for SML and use it to implement a fancier interface for blocks world. You'll almost certainly have to make a custom .cm file, so don't modify the one for this assignment. Make a new one and when you're done submit it along with the rest of the homework.

If you want to do 2D graphics you can learn about SDL::ML at `http://www.hardcoreprocessing.com/Freeware/SDLML.html` and if you want to do 3D graphics you can learn about SML3D at `http://sml3d.cs.uchicago.edu/`.

6 Conflatten

In this question you will prove the correctness for some simple functions on lists. First, consider the declaration for the `size` function.

```
fun size [] = 0
  | size ([]::R) = 1 + size R
  | size ((x::L)::R) = 1 + size (L::R)
```

Task 6.1 (2%). Describe in a sentence or two what `size` does. Give the most general type for `size`.

> **Solution 6.1** Given a list of lists `L`, `size` returns the number of lists in `L` plus the sum of the size of each list in `L`. The type of `size` is `size : 'a list list -> int`.

Now, consider the following functions:

```
fun flatten [] = []
  | flatten (L::R) = L @ flatten R
```

```
fun concat [] = []
  | concat ([]::R) = concat R
  | concat ((x::L)::R) = x :: concat(L::R)
```

Both of these functions achieve the same end. They take a list of lists and put all the values from each sub-list into a single main list. For example,

```
val [1, 2, 3] = flatten [[1], [], [2, 3]]
val [1, 2, 3] = concat [[1], [], [2, 3]]
val ["a", "b", "c"] = flatten [["a", "b"], [], [], ["c"], []]
```

Task 6.2 (12%). Prove Theorem 1 by induction. Think carefully about what variable you induct over, as now you're inducting over lists of lists instead of just lists. Be sure to cite any lemmas you use in your proof.[1]

Theorem 1. *For all types* `t` *and for all* `L : t list list`*,*

$$\texttt{flatten}\,\texttt{L} = \texttt{concat}\,\texttt{L}$$

Lemma 1. `size` *is a total function. (ie. for all types* `t`*, for all values* `L` *of type* `t list list` *then* `size L` *evaluates to a value.)*

Lemma 2. *You may assume that*

$$\texttt{size}\ [] = 0$$

Lemma 3. *For all correctly-typed values* `R`*,*

$$\texttt{size}\ ([] :: \texttt{R}) > \texttt{size}\ \texttt{R}$$

Lemma 4. *For all correctly-typed values* `x` *and* `R`*,*

$$\texttt{size}((\texttt{x} :: \texttt{L}) :: \texttt{R}) > \texttt{size}(\texttt{L} :: \texttt{R})$$

Lemma 5. *For all types* `t` *and all values* `L : t list`*,*
either `L = []` *or* `L = x :: L'` *for some* `x : t` *and* `L' : t list`*.*

Lemma 6. *For all types* `t` *and all values* `L : t list`*,*

$$[]@\texttt{L} = \texttt{L}$$

Lemma 7. *For all types* `t` *and all values* `L : t list`*,* `R : t list`*, and* `x : t`*,*

$$\texttt{x} :: (\texttt{L}@\texttt{R}) = (\texttt{x} :: \texttt{L})@\texttt{R}$$

Solution 6.2

Proof. **WTS:** for values `L : 'a list list` that `concat L` $=$ `flatten L`.

Proceed by induction on `size l`.

Base Case: `size L = 0`

Claim that `size L = 0` $\iff$ `l = []`.

[1] It's interesting to note that we could have stated Theorem 1 a more concisely as

$$\texttt{flatten} = \texttt{concat}$$

which is a direct transcription of the intuition of the problem into a formal statement. The statement given is an immediate expansion of this, so we don't lose anything by being a little bit more verbose.

For the base case `L = []`, then `size L = 0` by first case.

Assume that for `L = x::L'` that `size L` ≥ 0.

By lemma 5, either `x = []` or `x = x'::L''`.

In the first case, `size ([]::L')` $=$ `1 + size L'` ≥ 1.

In the second case, `size ((x'::L'')::L')` $=$ `1 + size(L''::L')` ≥ 1.

Hence, for all `L` $\neq$ `[]`, we have `size L` ≥ 1, so `size L = 0` $\iff$ `L = []`.

Hence, for the BC of the main proof, we have `L = []`. WTS: `concat [] = flatten []`

By clause 1 of `concat`, `concat [] = []`.

By clause 1 of `flatten`, `concat [] = []`.

Hence, `concat L` $=$ `flatten L`.

Inductive Case: Let `L` be a value such that for $n =$ `size L`, $n > 0$.

Assume for all $0 \leq k <$ `size L` that the theorem holds for all values `L` such that `size L` $= k$.

By lemma 1, `size` is total so `size L` will always evaluate to a value we can use.

WTS: `concat L` $=$ `flatten L`. As shown, `size L` $> 0 \implies$ `size L` $\neq$ `[]`, so by lemma 5, `L = y::R` for some values `y : 'a list` and `R : 'a list list`.

Also by lemma 5, `y` is either `[]` or `x :: L'`. We case on `y`:

Subcase 1: `L` matches `[] :: R`.

`flatten L`	$=$ `[] @ flatten R`	clause 2 of `flatten`
	$=$ `flatten R`	lemma 6
`concat L`	$=$ `concat R`	clause 2 of `concat`
	$=$ `flatten R`	IH

Note that by lemma 3, `size ([] :: R)` $>$ `size R`, so by the IH: `concat R = flatten R`.

Hence, `concat L = flatten L`.

Subcase 2: `L` matches `(x::L')::R`.

`concat L`	$=$ `x :: concat(L'::R)`	clause 3 of `concat`
	$=$ `x :: flatten(L'::R)`	IH
	$=$ `x :: (L' @ flatten R)`	clause 2 of `flatten`
	$=$ `(x :: L') @ flatten R`	lemma 7
	$=$ `flatten((x::L') @ R)`	clause 2 of `flatten`
	$=$ `flatten L`	`L = (x::L')::R`

Note that by lemma 4, `size((x::L')::R)` $>$ `size(L'::R)`, so by the IH we know

`concat (L'::R) = flatten (L'::R)`. Hence, `concat L = flatten L`.

In both possible cases, then, `concat L = flatten L`. Thus, by induction over `size L`, the theorem holds. □

7 Higher-Order Shrubs

For the next few problems, we're going to introduce a different tree-like data structure called a shrub. Instead of containing data at the nodes, shrubs only have data at the leaves. You'll be writing and analyzing higher order functions on polymorphic shrubs. Here's the type definition of `shrub`:

```
datatype 'a shrub = Leaf of 'a
                  | Branch of 'a shrub * 'a shrub
```

Task 7.1 (5%). Define a higher order function

```
shrubMap : ('a -> 'b) -> 'a shrub -> 'b shrub
```

such that for any shrub `s : 'a shrub` and any total function `f : 'a -> 'b`, `shrubMap f s` returns a shrub with `f` applied to every leaf. For example, to multiply all the leaves by 3 for `someShrub : int shrub`, we could use `shrubMap`:

```
val multThree = shrubMap (fn(n) => n * 3)
val newShrub = multThree someShrub
```

Solution 7.1 See solution in `hw05.sml`.

Task 7.2 (2%). Write a recurrence for $W_{shrubMap}(n)$ where n is the size of the input shrub. For this and the following problems, assume that the function `f` given to shrubMap has $O(1)$ work and span, and assume that the input shrub is balanced.

Solution 7.2

$$W_{\texttt{shrubMap}}(0) = k_1$$
$$W_{\texttt{shrubMap}}(n) = k_2 + 2W_{\texttt{shrubMap}}(n \texttt{ div } 2)$$

Task 7.3 (1%). Derive an estimate for the big-O of $W_{shrubMap}(n)$.

Solution 7.3 Intuitively, `shrubMap` applies a constant time function to every element in the shrub, so it has $O(n)$ work as the shrub has n elements.

More formally, we know the work tree for $W(n) = 2W(n/2) + k$ has $\log_2 n$ levels and at the i-th level has 2^i nodes doing k work each, so the total work is

$$\begin{aligned} k \sum_{i=1}^{\log_2 n} 2^i &= k(2^{\log_2 n+1} - 1) \\ &= 2kn - k \\ &\in O(n) \end{aligned}$$

Task 7.4 (2%). Write a recurrence for $S_{shrubMap}(n)$.

Solution 7.4

$$S_{\texttt{shrubMap}}(0) = k_1$$
$$S_{\texttt{shrubMap}}(n) = k_2 + S_{\texttt{shrubMap}}(n \texttt{ div } 2)$$

Task 7.5 (1%). Derive an estimate for the big-O of $S_{shrubMap}(n)$.

Solution 7.5 `shrubMap` does k work for $\log n$ levels, so its span is $O(\log n)$.

Task 7.6 (5%). Define a higher order function

```
shrubCombine : ('a * 'a -> 'a) -> 'a -> 'a shrub -> 'a
```

such that for any shrub `s : 'a shrub` and any total, associative function `f : 'a * 'a -> 'a` and its corresponding identity `i : 'a`, `shrubCombine f i s` returns the result of recursively combining the shrub with `f`. You can think of `shrubCombine` like `foldl` for shrubs. For example, to sum all the leaves in a shrub, we could use `shrubCombine`:

```
val sumShrub = shrubCombine (op +) 0
val someSum = sumShrub someShrub
```

At a leaf, `shrubCombine` should combine the identity with the value at the leaf (in that order). At a branch, `shrubCombine` should combine sub-branches in left-to-right order.

For `f` to be associative, for all well-typed values `a, b, c` we have: `f(a, f(b, c)) = f(f(a, b), c)`. For example, addition is associative as $(1+2)+3 = 1+(2+3)$, but subtraction is not associative as $(1-2)-3 \neq 1-(2-3)$. The identity `i` of `f` is a value such that `f(i, a) = f(a, i) = a` for all well-typed values `a`.

And now that you've defined the `shrub`bery, you must cut down the mightiest `tree` in the forest with... a herring!

Solution 7.6 See solution in `hw05.sml`.