15-150 Fall 2014
Homework 05

Out: Tuesday 23rd September, 2014
Due: Tuesday 30th September, 2014 at 23:59 AST

1 Introduction

The purpose of this assignment is to practice working with more complex data structures and algorithms than in previous assignments. We will be operating entirely in the world of SML programs.

1.1 Getting the Homework Assignment

The starter files for the homework assignment have been distributed through Autolab at

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

Select the page for the course, click on "hw05", and then "Download handout". Uncompressing the downloaded file will create the directory `hw05-handout` containing the starter files for the assignment. The directory where you will be doing your work is called `hw/05` (see below): copy the starter files there.[1]

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 will need to contain the file `hw05.pdf` with your written solutions and the files `isort.sml`, `delimiters.sml` and `delimiters2.sml` with your code. The starter code for this assignment contains incomplete versions of these files. You will need to complete them with the solution to the various programming tasks in the homework.

To submit your solutions, run

[1]The download and working directory have different names so that you don't accidentally overwrite your work were the starter files to be updated. **Do not do your work in the download directory!**

```
make
```

from the hw/05 directory on any Unix machine (this include Linux and Mac). This will produce a file hw05-handin.tgz, containing the files to be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your hw05-handin.tgz file via the "Handin your work" link. If you are working on AFS, you can alternatively run

```
make submit
```

from the hw/05 directory. That will create hw05-handin.tgz, containing the files to be handed in for this homework assignment and directly submit this to Autolab.

All submission will go through Autolab's "autograder". The autograder simply runs a series of tests against the reference solution. Each module has an associated number of points equal to the number of functions you need to complete. For each such function, you get 1.0 points if your code passes all tests, and 0.0 if it fails at least one test. Click on the cumulative number for a module for details. Obtaining the maximum for a module does not guarantee full credit in a task, and neither does a 0.0 translate into no points for it. In fact, the course staff will be running additional tests, reading all code, and taking into account other aspects of the submitted code such as structured comments and tests (see below), style and elegance.

To promote good programming habits, your are limited to a maximum of 5 submissions for this homework. Use them judiciously! In particular, make sure your code compiles cleanly before submitting it. Also, make sure your own test suite is sufficiently broad.

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

The SML files isort.sml, delimiters.sml and delimiters2.sml 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 30th September, 2014 at 23:59 AST. Remember that there are no late days for this course and you will get 2 bonus points for every 12 hours that you submit early.

1.4 Proof Structure

Remember that every proof by induction is structured as follows:

1. The specific *technique* being employed and on what.

2. The *structure* of the proof (number of cases and what they are).

3. For each base case:

   - The statement specialized to this case ("*To show*").
   - The proof of this case.

4. For each inductive case:

   - The statement specialized to this case ("*To show*").
   - The induction hypothesis or hypotheses (*IH*).
   - The proof of this case.

Following this methodology, students have historically submitted proofs that contained fewer errors and were more likely to be correct than otherwise.

1.5 Code Structure

On this and future assignments, we will be grading your programs on more than just their input-output behavior. It's not enough to have programs that happen to work: they need to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

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

1. In the first line of comments, write a call template 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 (include type annotations for the arguments and result of the function)

5. Provide test cases, generally in the format

   ```
   val <return value> = <function> <argument value>.
   ```

For example, for the factorial function presented in lecture:

```
(*  factorial (n) ==> res
 *  REQUIRES:  n >= 0
 *  ENSURES: res is  n!
 *)
fun factorial (0: int): int = 1
  | factorial n = n * factorial (n-1)

(* Tests: *)

val 1 = factorial 0
val 720 = factorial 6
```

1.6 Style

Programs are written for people to read — it's convenient that they can be executed by machines, but their high level text is a way for one person to explain an idea to another person. Your code should reflect this, and we will grade your code on how easy it is to understand.

The published style guide is your primary resource here. Strive to write clear, concise, and elegant code. If you have any questions about style, or just have a feeling that a piece of code could be written more simply, don't hesitate to ask!

2 SML in Action

In this section, you will test your understanding of the way the SML interpreter works.

2.1 Type Inference

For each of the following functions, give a step-by-step description of the process of inferring its type. If the function does not have a type, explain where and why the type inference procedure stops.

Some of these functions refer to the following type declaration:

```
datatype 'a tree = empty
                 | node of 'a tree * 'a * 'a tree
```

Task 2.1 (2 points) Carry out type inference on the following function:

```
fun meow ([], kitty) = kitty
  | meow (sooo::FLUFFY, kitty) = "meow " ^ meow(FLUFFY, kitty)
```

Task 2.2 (2 points) Carry out type inference on the following function:

```
fun f (empty, n) = nil
  | f (node(L,x,R),n) = node(f(L,n+1), (x,n), f(R,n+1))
```

Task 2.3 (2 points) Carry out type inference on the following function:

```
fun f x = f (f (x+1))
```

2.2 Scope

In the following tasks, you will determine the type and value of various expressions within nested scope. You may need to look up the type of library functions.

Task 2.4 (6 points) Consider the following code fragment (the line numbers are for reference only — they are not part of the code itself):

```
 (1)  val x = 2
 (2)  val tmp = Real.fromInt (x+1)
 (3)  fun assemble (x: int, y: real): int =
 (4)    let
 (5)      val g: real =
 (6)        let
 (7)          val x = 4
 (8)          val m = tmp * Real.fromInt x
 (9)          val x = 13
(10)          val y = m * y
(11)        in
(12)          y - m + Real.fromInt x
(13)        end
(14)    in
(15)      x + Real.trunc g
(16)    end
(17)  val z = assemble (x, 5.0)
```

1. What is the value of the variable `x` on line (8)? Explain why. What is its type?
2. What is the value of the variable `m` on line (10)? Explain why. What is its type?
3. What is the value of the variable `x` on line (15)? Explain why. What is its type?
4. What is the value of the expression `assemble (x, 5.0)` on line (17)? Explain why. What is its type?

Task 2.5 (7 points) Consider the following code fragment (the line numbers are for reference only — they are not part of the code itself):

```
 (1)  fun f x =
 (2)    let
 (3)      val y = x + 3
 (4)      fun g z = z + y
 (5)      val y = 7
 (6)      fun h 0 = 0
 (7)        | h n = y - x + h (n-1)
 (8)    in
 (9)      g x + h x
(10)    end
(11)  val result = f 1
```

During the evaluation of `f 1` on line (11),

1. what is the value of the variable `y` on line (4)? Explain why.
2. what is the value of the variable `y` on line (7)? Explain why.
3. what is the value of the expression `g x` on line (9)? Explain why. What is its type?
4. what is the value of the expression `h x` on line (9)? Explain why. What is its type?
5. what is the value of the variable `result` on line (11)? Explain why.

3 Insertion Sort

In this section, we will be adapting insertion sort to work on trees.

3.1 Insertion Sort on Lists

Insertion sort is another classical sorting algorithm on lists. It works by inserting elements in a sorted list in the "right" position. To sort a given list, insertion sort does this operation for each element in it starting from the empty (sorted) list. The SML implementation of insertion sort is very simple:

```
fun insert (x: int, []: int list): int list = [x]
  | insert (x, y::l) = if x <= y
                         then x::y::l
                         else y :: insert (x, l)
fun isort ([]: int list): int list = []
  | isort (x::l) = insert (x, isort l)
```

(The starter file `isort.sml` contains this code with its specs for reference.) It is easy to check that the work and span of `isort` are quadratic in the length of the input list. It is not a great sorting algorithm.

3.2 Sorting Trees by Insertion

Could we use some of the ideas behind insertion sort on lists to sort trees? To explore this, we will consider trees with data in their inner nodes:

```
datatype 'a tree = empty
                 | node of 'a tree * 'a * 'a tree
```

All the trees in this exercise will contain integers data, and therefore have type `int tree`.

Inserting a node in a sorted tree in such a way that the resulting tree is sorted is a fairly simple. It is achieved by the following function `Insert` (first letter capitalized) which does on trees what `insert` does on lists:

```
fun Insert (x: int, empty: int tree): int tree = node(empty, x, empty)
  | Insert (x, node(tL,y,tR)) =
     if x <= y
       then node (Insert (x, tL), y, tR)
       else node (tL, y, Insert (x, tR))
```

This function can be found in the starter file `isort.sml`, together with its specs.

Task 3.1 (6 points) Write the recurrence relation for the work of `Insert` in terms of the height of the input tree, deduce an upper-bound approximation in closed form, and determine a tight big-O class for it. Use this to identify the best and worst cases measured in terms of the number of nodes in the tree, and give their big-O classes. Do the same for its span.

One simple approach for sorting a tree is to read off its elements into a list (the familiar code for `inorder` is given in `isort.sml`) and then repeatedly use `Insert`. That pretty much follows the idea behind insertion sort on lists.

Task 3.2 (5 points) Implement the SML function

```
ILsort: int tree -> int tree
```

that returns a sorted tree with the exact same elements as its input. Your code should use `Insert` and `inorder`.

Task 3.3 (8 points) Show that your code produces a sorted tree by proving the following property:

Property 1. *For all* `t: int tree`*,*

`sorted (inorder (ILsort t))` $\cong$ `true`

If your code for `ILsort` makes use of auxiliary functions, you may need to state and prove auxiliary lemmas

As you carry out these proofs, you may find the following lemma useful (but you shall reference each use):

Lemma 1. *For all* `x: int` *and* `t: int tree`*,*

If `sorted (inorder t)` $\cong$ `true`,
then `sorted (inorder (Insert(x, t)))` $\cong$ `true`.

Task 3.4 (10 points) Write the recurrence relation for the work of `ILsort` in terms of the number of nodes in the tree. For both the best case and the worst case, deduce an upper-bound approximation in closed form, and determine a tight big-O class for it. Do the same for its span.

Task 3.5 (2 points) Chances are that your code does not prevent the worst case scenario from happening. How would you modify it so that the best case is always realized. Explain why. [You are not required to modify your code in this way, although you are welcome to.]

Task 3.6 (4 points) How does this compare with the work and span of merge sort on trees seen in class? If it is either better or worse, explain why.

3.3 Insertion Sort for Trees

The above approach to sorting a tree goes through a list, the inorder traversal of the input tree. Can we implement a more direct version of insertion sort on trees?

Task 3.7 (5 points) Implement the SML function

```
    Isort: int tree -> int tree
```

that returns a sorted tree with the exact same elements as its input. Your code should use `Insert`, but it is not allowed to use any list or operations on lists.

4 Delimiters

In programming languages, mathematics and even written English, the right and left parenthesis allow grouping what's between them for whatever purpose. For example, in the SML expression `(1+2)*3`, the parentheses tell us that `1+2` should be evaluated first. Such expressions lose their meaning when parentheses are not properly nested. For example, `(1+2*3` and `1+2)*3` are both incorrect. Other constructs behave like parentheses, like `{` and `}` for example. They are called *delimiters*

This section contains exercises about checking that delimiters are correctly nested and expressing this effectively.

4.1 Nested Parentheses

Let's start with a radical simplification: to determine whether a string contains properly nested parentheses, we will throw away anything that is not a parenthesis. That will leave us with strings of the following form:

- `"()"`
- `"(())()(())"`
- `"()()(())"`
- `")(())"`
- `"(()()"`
- `"(()))"`

A string of parentheses is *valid* (i.e., properly nested) if, when reading it from left to right, at every point there are no more right parentheses than left parentheses, but every left parenthesis is eventually matched by some right parenthesis. The example strings on the left are properly nested, but the strings on the right are not.

Strings are tedious to work with. For this reason, we introduce the following type declarations

```
datatype par = LPAR | RPAR
type pList = par list
```

where `LPAR` represents the left parenthesis `"("` and `RPAR` stands for the right parenthesis `")"`. Then, a parentheses string is just a list of `par`, which we abbreviate as the type `pList`. For example, the string `"()()"` is expressed as the `pList` value `[LPAR, RPAR, LPAR, RPAR]`.

While representing parentheses string as values of type `pList` is convenient for writing code, it makes testing and debugging complicated. For this reason, we have provided you with the functions `pList_fromString` and `pList_toString` that, respectively, convert a parentheses string like `"()()"` into the corresponding value `[LPAR, RPAR, LPAR, RPAR]` of type `pList`, and vice versa. You can find them in the starter file `delimiters.sml`.

Task 4.1 (6 points) Implement the SML function

```
valid: pList -> bool
```

so that the call `valid ps` returns `true` if `ps` represents a valid, i.e., properly nested, parentheses string. It returns `false` otherwise. For example, it shall return `true` on the three earlier examples on the left, and `false` on the three examples on the right.

4.2 Parentheses Trees

Values of type `pList` can represent invalid parentheses strings, for example `[LPAR, RPAR, LPAR]` corresponds to `"()("` which is not valid. Can we engineer a representation so that only valid parentheses strings can be written? The idea is to observe that there are just three ways to construct properly nested parentheses string:

- The empty string is properly nested, trivially.
- If we have a properly nested parentheses string s, then putting a left parenthesis to its left and a right parenthesis to its right yields (s) which is is properly nested.
- If we have two properly nested parentheses string s_1 and s_2 neither of which is empty, then putting them side by side as in $s_1 s_2$ yields a properly nested parentheses string.

We can capture this idea by means of the following datatype

```
datatype pTree = empty                (* no parentheses *)
               | nested of pTree      (* nested parentheses *)
               | sbs of pTree * pTree (* side by side *)
```

Each constructor represents one of the above possibilities. For example, the (valid) parentheses string `"()"` becomes `nested empty` while `"()()"` becomes `sbs(nested empty, nested empty)`.

The starter file `delimiters.sml` provides the printing function `pTree_toString` that converts a `pTree` to the corresponding SML `string`. Feel free to use it.

Now, it should be possible to go back and forth between (valid) parentheses strings represented as `pList` and `pTree`, shouldn't it?

Task 4.2 (3 points) Implement the SML function

```
flattenPTree: pTree -> pList
```

that converts a `pTree` to the corresponding `pList`. In particular, `pList_toString (flattenPTree t)` and `pTree_toString t` should return the exact same SML string.

Going the other way around, from a parentheses string represented as a `pList` to the corresponding `pTree` is more complicated. Consider the parentheses string `"(())"`: upon seeing the first `"("`, we need to remember it so that when we find the matching `")"` we can build a proper nested tree out of what we have found in between (the tree `nested empty` corresponding to `"()"`). However, when processing the last closed parenthesis in `"((())())"`, we must combine the trees `nested (nested empty)` and `nested empty` corresponding to `"(())"` and `"()"` respectively into `sbs(nested (nested empty), nested empty)` before proceeding.[2]

The simplest way to do all this is to make use of a *stack* where we can store partially done work to be completed later. The following type declarations define stacks:

```
datatype stackItem = OPEN
                   | T of pTree
type stack = stackItem list
```

Our stack contains one of two kinds of items, either open parentheses that we have not closed yet (that's the constructor `OPEN`) or `pTree`'s that we have completely recognized (that's the constructor `T`). The starter code in file `delimiters.sml` provides the printing function `stack_toString` that returns a string representation of a stack.

With the help of a stack, we can convert a parentheses string into a `pTree` as follows:

- If we see a left parenthesis, we push it onto the stack — we will need to find the matching right parenthesis, but we first need to build the `pTree` for what's in between.
- If we see a right parenthesis, we'd better have a `pTree` on the stack and a left parenthesis below it. Then, we replace them with a larger `pTree`.
- If we have two `pTree`'s on the stack, then we can combine them into a single larger `pTree`.
- If we have reached the end of the parentheses string given as input, the stack should not have any unprocessed right parentheses. In fact, we should be able to return the `pTree` corresponding to our original input.

Task 4.3 (14 points) Write the SML function

```
pp: pList * stack -> pTree
```

that implements the above strategy. Specifically, the call `pp (ps, S)` returns the value `t` of type `pTree` whenever processing `ps` starting from stack `S` produces `t`. You may assume that, were you to prefix `ps` with as many `LPAR` as there are `OPEN` in `S`, the resulting `pList` would be valid.

Task 4.4 (2 points) Implement the SML function

[2]Going from a string-like representation to a representation that brings out its structure is called *parsing*. The parsing algorithm we are pursuing is a very simple instance of what is known as *LARL parsing*, which is employed in nearly all compilers and interpreters.

```
parsePar: pList -> pTree
```

so that, given a valid parentheses string `ps`, the call `parsePar ps` returns the `pTree` corresponding to `ps`.

4.3 Beyond Parentheses — Bonus Tasks

Parentheses are not the only delimiters in common use. For example, many programming languages also use `"{"` and `"}"` and other brackets, HTML has tags such as `<h1>` and `</h1>`, and there is LaTeX's environments such as `\begin{center}` and `\end{center}`. In fact, we almost always use multiple delimiters in the programs we write.

In this section, we will adapt the ideas we developed for parentheses to handle generic delimiters, possibly several kinds at once. Valid nesting acquires a new dimension as we do not want different delimiters to cross. For example, `"({)}"` is not properly nested even though each opening delimiter is followed by a closing delimiter of the same kind. Instead `"({})"` is properly nested.

Delimiters such as `"<h1>"` and `"</h1>"` have a *root*, `h1`, and a *left* and *right form*, `<h1>` and `</h1>` respectively. The same is true of `\begin{center}` and `\end{center}`. We modify the definition of the type `par` as follows to include the root of a delimiter:

```
datatype par = L of string | R of string
```

Therefore, `<h1>` and `\end{center}` are represented as `L "h1"` and `R "center"`, respectively. The other types used to represent parentheses strings earlier are modified similarly. We do not directly consider delimiters without a root, such as `"{"` and `"}"`, although they can easily be encoded.

The starter file for this section, `delimiters2.sml`, defines these types as well as printing functions for the so-modified versions of `pList`, `pTree` and `stack`. It also contains a variant of the function `pList_fromString` that produces a `pList` on the basis of a string representation. In this representation, the right and left delimiters with root `h1` are written `"(h1"` and `")h1"`, respectively, and similarly for every other delimiter. These functions are meant to make testing and debugging more convenient.

With these preliminaries in place, you are asked to re-implement the functions you wrote for parentheses string to work with generic delimiters. Do so in the file `delimiters2.sml`.

Task 4.5 (5 bonus points) Implement the SML function

```
valid: pList -> bool
```

so that the call `valid ps` returns `true` if `ps` represents a valid, i.e., properly nested, delimiter string.

Task 4.6 (2 bonus points) Implement the SML function

```
flattenPTree: pTree -> pList
```

that converts a `pTree` to the corresponding `pList`.

Task 4.7 (7 bonus points) Write the SML function

```
pp: pList * stack -> pTree
```

that implements the same strategy seen for parentheses strings, but for delimiter strings.

Task 4.8 (1 bonus points) Implement the SML function

```
parsePar: pList -> pTree
```

so that, given a valid delimiter string `ps`, the call `parsePar ps` returns the `pTree` corresponding to `ps`.