Homework 05

15-150 Fall 2024

Due: Mon 7th Oct, 2024 at 8:00PM

LaTeX

The written part of your assignments must be written in LaTeX. To make your job easier, a LaTeX template is provided for each assignment where you simply need to fill in your solutions.

If you want to know what is the LaTeX name for a symbol, you can use Detexify. Also, Mathcha can help you type math. For constructs particular to this course (inductive definitions, proofs, etc), we provide a LaTeX guide with templates.

Finally, if you are just lost on how to type something or why your file is not compiling, reach out to the course staff.

Your submission will be penalized if it is not written using LaTeX. The applicable penalties are described in the style guide.

Code Structure

Check out the *basic requirements* for coding tasks in the style guide.

Proof Structure

Check out the *basic structure* for proofs in the style guide.

1 SML in Action

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

1.1 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 1.1 (2 points) Carry out type inference on the following function:

```
fun woof ([], puppy) = puppy
  | woof (sooo::CUTE, puppy) = 4.0 + woof(CUTE, puppy)
```

Task 1.2 (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 1.3 (2 points) Carry out type inference on the following function:

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

1.2 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 1.4 (6 points) Consider the following code fragment:

```
val x : int = 5
val y : real = 2.0
val temp : real = real (x - 1)
fun generate (x : int, y : real, z : real) : int =
  let
    val g : real =
      let
        val x : int = 3
        val z : real = z * y
        val k : real = temp * (real x)
        val a : int = 38
        val y : real = k * y
      in
        z * y + (real a) + k
      end
  in
    x + trunc g
  end

val z = generate (x, y, temp)
```

1. What value gets substituted for the variable `x` on line (10)? Briefly explain why. What is its type?
2. What value gets substituted for the variable `k` on line (12)? Briefly explain why. What is its type?
3. What value gets substituted for the variable `x` on line (17)? Briefly explain why. What is its type?
4. What value does the expression `generate (x, y, temp)` evaluate to on line (20)?

Task 1.5 (7 points) Consider the following code fragment:

```
1  val x = ~1
2  fun f x =
3      let
4        val y = x
5        val x = 90
6        fun f x = y + 9
7        val x = 50
8      in
9        f x
10     end
11 val x = f x
```

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

1. What value gets substituted for the variable `y` on line (4)? Briefly explain why. What is its type?
2. What value gets substituted for the variable `x` on line (9)? Briefly explain why. What is its type?
3. To what is the identifier `f` bound on line (9)?
4. What value gets substituted for the rightmost occurence of the variable `x` on line (11)? Briefly explain why. What is its type?
5. What value gets substituted for the leftmost occurence of the variable `x` on line (11)? Briefly explain why. What is its type?

Task 1.6 (6 points) Consider the following code fragment:

```
1  val x = 13
2  val x =
3      let
4        val x = 14
5        val x = (case x mod 4
6                   of 1 => "x"
7                    | 2 => "o") ^
8                let
9                  val x = 9
10                 val x = "x"
11               in
12                 x
```

```
13              end
14      in
15        x ^ "o"
16      end
```

1. What value gets substituted for the leftmost occurrence of the variable `x` on line (5)? Briefly explain why. What is its type?
2. What value gets substituted for the variable `x` on line (12)? Briefly explain why. What is its type?
3. What value gets substituted for the variable `x` on line (15)? Briefly explain why. What is its type?
4. What would change if line 1 were changed to `val x = 12`? Why?
5. What would change if line 4 were changed to `val x = 12`? Why?

2 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.

2.1 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`.

You are allowed to use these functions for debugging and testing purposes *only*.

Coding Task 2.1 (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.

2.2 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?

Coding Task 2.2 (2 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.[1]

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.

[1] 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.

Coding Task 2.3 (11 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.

Coding Task 2.4 (1 points) Implement the SML function

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

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

2.3 Beyond Parentheses

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 par2 = 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 second part of starter file `delimiters.sml` defines these types as well as printing functions for the so-modified versions of `pList`, `pTree` and `stack` (called `pList2`, `pTree2` and `stack2`). It also contains the function `pList2_fromString` that produces a `pList2` 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 second part of file `delimiters.sml`.

Coding Task 2.5 (3 points) Implement the SML function

```
valid2: pList2 -> bool
```

so that the call `valid2 ps` returns `true` if `ps` represents a valid, i.e., properly nested, delimiter string.

Coding Task 2.6 (1 points) Implement the SML function

```
flattenPTree2: pTree2 -> pList2
```

that converts a `pTree2` to the corresponding `pList2`.

Coding Task 2.7 (5 points) Write the SML function

```
pp2: pList2 * stack2 -> pTree2
```

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

Coding Task 2.8 (1 points) Implement the SML function

```
parsePar2: pList2 -> pTree2
```

so that, given a valid delimiter string `ps`, the call `parsePar2 ps` returns the `pTree2` corresponding to `ps`.

3 Common Subsequences

A common subsequence of two sequences X and Y is a sequence Z such that it is a subsequence of X and a subsequence of Y at the same time. For example, if $X = \langle 1, 5, 1, 5, 0\rangle$ and $Y = \langle 1, 5, 2, 1, 0\rangle$, then $Z = \langle 5, 1, 0\rangle$ is a common subsequence of X and Y.

More formally, let $X = \langle x_1, x_2, ..., x_n\rangle$ and $Y = \langle y_1, y_2, ..., y_m\rangle$ be two sequences of elements. The sequence $Z = \langle z_1, z_2, ..., z_k\rangle$ is called a *common subsequence* of X and Y if it can be obtained by either sequence via the deletion of some elements. Equivalently, more mathematically, for every i, j, there exists a, b, c, d such that $z_i = x_a = y_c$ and $z_j = x_b = y_d$ and if $i < j$, then $a < b$ and $c < d$.

A popular algorithm for finding the *longest* common subsequence of two sequences is based on two properties. Suppose we are trying to find the longest common subsequence (LCS) of two sequences `X = x::lx` and `Y = y::ly`. Then:

1. If `x = y`, then the LCS of `X` and `Y` will contain this element.
2. Otherwise, it will be either the LCS between `x::lx` and `ly` or the one between `lx` and `y::ly` (whichever is the longest).

Task 3.1 (3 points) Write down a mathematical inductive definition for a function that computes the longest common subsequence of two sequences based on the properties above.

Coding Task 3.2 (2 points) Implement the SML function that computes the longest common subsequences for two lists with elements of an arbitrary type.

```
lcs: ''a list * ''a list -> ''a list
```

If we want to compute instead of the longest subsequence, the common subsequence with a specific length k, then we need to modify the previous algorithm to do some backtracking.

Coding Task 3.3 (5 points) Using two types of exceptions implement a function that will find a common subsequence of size k between two sequences if it exists, or raise `SubSeqTooShort` otherwise.

```
exception SubSeqTooLong;
exception SubSeqTooShort;

css: ''a list * ''a list * int -> ''a list
```