1 Introduction

This assignment is designed to give you some practice with brute force and divide-and-conquer algorithms. You will implement two solutions to the *maximum parenthesis distance* problem and perform some analysis of your solutions. Note that this lab is conceptually difficult, so get started early!

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf parenlab-handout.tgz
```

from a terminal window on a unix/linux system. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones that will be handed in terms of code

1. * `MkBruteForcePD.sml`
2. * `MkDivideAndConquerPD.sml`
3. `Sandbox.sml`
4. `Tests.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment. Your written answers should be composed in a suitable word processor such as LaTeX, Word, OpenOffice, etc. *Handwritten (and scanned in) answers will NOT be accepted.* **Make sure your name is also in the `written.pdf` file.**

3 Submission

To submit your assignment to Autolab in Diderot, generate a handin archive `handin.tgz` with the command

```
make package
```

You can then submit your `handin.tgz` file Diderot. You should submit the answers to the written questions in *written.pdf* to Gradescope at `https://www.gradescope.com/courses/225303`.

Note that your code submitted to Autolab will not be graded until after the due date. It is your responsibility to test your code thoroughly before submitting.

3.1 Important points

1. Before submission, always, *always* verify that the files you will submit are indeed the versions that you intended to submit. Be very careful to NOT submit the wrong lab's `written.pdf`, blank files, or old versions of the files. **We will not entertain "*Oh no, I submitted the wrong file!*" emails or Diderot posts after the due date**. We will adopt a *zero-tolerance* approach to this, and you will receive the grade of your most recent submission (which may be subject to late submission penalties!) *You have been warned*.

2. If the homework has a written component, the answers to each written question in *written.pdf* should be on one page and each answer should fit on one page. If you decide not to answer one of the questions, still have the empty page in the file. This will make everyone's life easier in Gradescope.

3. If the written part has a question that asks you to show that your code has a certain complexity bound, **you should include a copy of your code in your answer and annotate it as suitable with relevant complexity bounds using comments and then argue or compute the resulting bounds. Answers to such question without a copy of the code you submitted will not be accepted.**

4. The questions below ask you to organize your solutions in a number of modules written almost from scratch. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

5. You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

6. This assignment requires the use of library code spread over several files. The compilation of this is orchestrated by the compilation manager through several `.cm` files. Instructions on how to use CM can be found in the file `cm.pdf` under Resources on Diderot.

7. Style grading for this assignment will be evaluated on a 5 point scale: -2 through +2. Style points will be added to your final grade from the rest of your homework. **You should review the full style guide available on the Diderot under Resources.**

8. **You should submit your solutions on or before the deadline on the top of the first page. Failure to do so will cause substantial deductions of your grade. Late submissions of up to a day will scale your homework grade by 70% (so if you were to get 100% as your grade, you will now get 70%). Late submissions of up to 2 days will scale your homework grade by 50%. Beyond 2 days, you will receive 0 as your homework grade. This scaling will be handled outside Autolab.**

4 Parenthesis Distance

We define a *matched* sequence of parentheses inductively as

$$p ::= \langle\rangle \mid p\ p \mid \texttt{(}\ p\ \texttt{)}$$

In other words, a matched sequence is one of (a) the empty sequence, (b) the concatenation of two matched sequences, or (c) a pair of parentheses surrounding a matched sequence. In the third case, we'll say that the *parenthesis distance* of the pair of parentheses is the length of the sequence they are surrounding. The *maximum* parenthesis distance (MPD) is the largest of these distances within a particular sequence. For example, `()` has an MPD of 0, `(())` has an MPD of 2, and `()(((())()))(())` has an MPD of 6. **Note that MPD is defined only for matched sequences which are non-empty.**

4.1 Logistics

- **Representation:** Take a look at `support/Paren.sml`. This contains the definition of the type of parentheses as well as some utility functions. Just to be clear: `Paren.L` and `Paren.R` are the "left" and "right" parentheses, respectively. The input `()` would be given by the sequence ⟨`Paren.L`, `Paren.R`⟩.
- **Indicating Parallelism:** The following code is an example of using `Primitives.par`, a function which runs its two arguments in parallel and returns their results.

  ```
  val (x, y) = Primitives.par (fn () ⇒ foo 42, fn () ⇒ bar "hello")
  ```

  Note that all code is sequential by default, and therefore `par` is necessary to indicate what should run in parallel. It has the following type:

  ```
  val par : (unit → α) * (unit → β) → (α * β)
  ```

  You should only use `par` when it is absolutely necessary – i.e., when the use of `par` asymptotically improves the span of a function. We may take off points for unwarranted uses of `par`.
- **Modules:**
  - By default, we tend to keep all of our structures closed. This means that you will always need to preface identifiers with what structure they come from. For example, to map across a sequence in Task 4.1, you'll need to write "`Seq.map`".
  - For structures which implement an ADT, we use the convention of including a `type t` in the signature. For example, the type of parentheses in this assignment should be written `Paren.t`.

4.2 Implementation

You will now write two implementations of the function

```
val parenDist : Paren.t seq → int option
```

where (`parenDist` S) returns the maximum parenthesis distance of S if S is a matched, non-empty sequence. Otherwise, it returns `NONE`.

Task 4.1 (20 pts). In `MkBruteForcePD.sml`, implement `parenDist`. Your implementation must be *brute force*, meaning that it should first generate all possible solutions, then select the one which meets the requirements of the problem. You should strive for a significant amount of parallelism, since each possible solution can be considered independently of all others.

A reference that will be incredibly useful to you throughout the semester is the Library documentation that is available on the Diderot Home page under Books. In particular, you'll want information about the `SEQUENCE` signature outlining types and specs, and about the `ArraySequence` structure, implementing `SEQUENCE` with certain cost bounds. For some examples, look up the functions `map` or `tabulate` to see what their types and cost bounds are.

Task 4.2 (50 pts). In `MkDivideAndConquerPD.sml`, implement `parenDist`. Your implementation must be *divide-and-conquer*, meaning that it satisfies the following work and span recurrences where n is the length of the input. We assume here that `Seq` = `ArraySequence`.

$$W(n) = 2\,W\left(\frac{n}{2}\right) + O(1)$$
$$S(n) = S\left(\frac{n}{2}\right) + O(1)$$

Important: You're only allowed to use the sequence functions `splitMid` and `length` in this task. (You can also use `toString` for debugging.) This is enforced by the signature of the `Seq` parameter of `MkDivideAndConquerPD`. Check out `support/SEQUENCE_RESTRICTED.sig` for more details.

Don't forget to use `Primitives.par` to indicate parallelism.

5 Testing

There are two ways to test your code.

1. In `Sandbox.sml`, write whatever testing code you'd like. You can then access the sandbox at the REPL:

   ```
   - CM.make "sandbox.cm";
   ...
   - open Sandbox;
   ```

2. In `Tests.sml`, add test cases according to the instructions given. Then run the autograder:

   ```
   - CM.make "autograder.cm";
   ...
   - Autograder.run ();
   ```

As usual, it is very important that you thoroughly test your code before you submit. After the deadline, we will grade your code with a private set of test cases.

6 Analysis

Task 6.1 (15 pts). Analyze your brute force implementation of `parenDist`, giving tight Big-*O* bounds for its work and span. For cost bounds of sequence functions, you should assume the `ArraySequence` implementation. Your score for this problem will depend on the correctness of your code (i.e. you can't just ignore Task 4.1 and expect to get free points here for correctly stating that your code is $O(1)$).

For each of the following, solve the given work and span recurrences. Show your work, and give your answers as tight Big-*O* bounds in terms of n, the length of the input.

Task 6.2 (15 pts). These recurrences were given in Task 4.2. Solve them to determine the asymptotic runtime behavior of your code. You may use any method from class: repeated expansion, tree, brick, or substitution.

$$W(n) = 2\,W\left(\frac{n}{2}\right) + O(1)$$
$$S(n) = S\left(\frac{n}{2}\right) + O(1)$$

Task 6.3 (15 pts). In Task 4.2, we assumed `Seq` = `ArraySequence`. If we instead used `TreeSequence`, then your work and span might have looked something like the following.

Solve the work recurrence. You may use any of repeated expansion, tree, brick, or substitution methods for the span recurrence.

$$W(n) = 2\,W\left(\frac{n}{2}\right) + O(\log n)$$
$$S(n) = S\left(\frac{n}{2}\right) + O(\log n)$$

Task 6.4 (15 pts). Finally, if we used `ListSequence`, then your work and span might look like the following. Solve these recurrences using any method from class: repeated expansion, tree, brick, or substitution.

$$W(n) = 2\,W\left(\frac{n}{2}\right) + O(n)$$
$$S(n) = S\left(\frac{n}{2}\right) + O(n)$$

Task 6.5 (5 pts). Solve the following recurrence asymptotically using any of the methods:

$$W(n) = W(\frac{2n}{3}) + W(\frac{n}{3}) + kn$$