Recitation 2

Parenthesis Matching

2.1 Announcements

- *ParenLab* has been released, and is due **Friday afternoon**. It's worth 100 points. This lab is conceptually difficult – if you haven't started yet, do so tonight!
- *SkylineLab* will be released on Friday.

2.2 Parentheses and Matched Sequences

Suppose you are given a sequence of parentheses. You want to determine if it is *matched*, meaning "properly nested". Let's begin by defining this more carefully.

Definition 2.1. *A* matched *sequence of parentheses* p *is defined inductively as*

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

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.

To be consistent with ParenLab, we'll implement parentheses as a custom datatype given in a structure `Paren`.

```
structure Paren =
struct
  datatype t = L | R
  ...
end
```

Our goal is to implement a function

```
val parenMatch : Paren.t Seq.t → bool
```

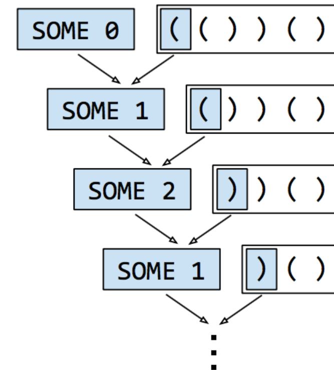
where (`parenMatch` S) determines whether or not S is a matched sequence.

Note that you will need to familiarize yourself with the 210 library. Documentation can be found on the course website at http://www.cs.cmu.edu/~15210/docs/. In particular, you should look closely at the `SEQUENCE` interface and the `ArraySequence` implementation.

2.3 From Left to Right

Task 2.2. *Implement* `parenMatch` *using the sequence function* `iterate`.

This algorithm is fairly simple: we just iterate a counter across the sequence. Starting from 0, we increment it at each sighting of a left-parenthesis, and decrement it at each right-parenthesis. If the counter never goes negative and its final value is 0, then the sequence is matched.



In terms of implementation, we'll actually use the type `int option` for the counter. Instead of letting it go negative, we'll set it to `NONE`, and then carry the `NONE` through to the end. (Alternatively, we could raise an exception and then handle it outside the `iterate`, but using an `option` is a bit cleaner.)

Algorithm 2.3. *Iterative parenthesis matching.*

```
1  fun parenMatch S =
2    let
3      fun adjustCounter (x, p) =
4        case x of
5          NONE => NONE
6        | SOME c =>
7            case p of
8              Paren.L => SOME (c+1)
9            | Paren.R => if c = 0 then NONE else SOME (c-1)
10   in
11     Seq.iterate adjustCounter (SOME 0) S = SOME 0
12   end
```

Remark 2.4. *The sequence function* `iterate` *is nearly identical to the list function* `foldl`*. The only differences are that it operates on sequences, and its function argument expects a pair in swapped order:*

```
val iterate : (β * α → β) → β → α seq → β
val foldl : (α * β → β) → β → α list → β
```

We type `iterate` *in this way to emphasize that it operates from left to right.*

2.4 Divide and Conquer

Task 2.5. *Implement* `parenMatch` *with a divide-and-conquer approach. Your implementation should satisfy the following work and span recurrences where* n *is the length of the input.*

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

Also briefly justify that your implementation meets the cost bounds shown. You should assume `Seq` = `ArraySequence` *for cost bounds.*

Our goal is to split the sequence roughly in half, recursively solve the smaller instances, then combine their results. But what should the recursive calls return? Our first thought might be to just return whether or not the smaller sequences are matched. However, this won't work. A sequence such as `((()))` would be split into `(((` and `)))`, neither of which are matched. We can't possibly determine that the concatenation of two unmatched sequences forms a matched one without more information. We need to *strengthen the problem*.

Remark 2.6. *"Strengthening the problem" is akin to strengthening the inductive hypothesis in an inductive proof. We prove a stronger statement, then conclude the original statement as a corollary.*

Consider this: take a matched sequence, and find an instance of the immediate pair `()`. Remove this pair. Is the sequence still matched? Yes it is! How about if the original sequence was unmatched – is it still unmatched? Once again, yes!

Observation 2.7. *If a sequence* S *contains the immediate pair* `()`*, then* S *is matched if and only if it is still matched after removing the pair.*

Now consider repeatedly removing all immediate pairs. Eventually, you will be left with a sequence of the form $)^i\,(^j$ – that is, a sequence of some number of right-parentheses followed by some number of left-parentheses. If the original was matched, then you'll have the empty sequence, which can be written as $)^0\,(^0$.

To make use of this in our divide-and-conquer algorithm, we'll have our recursive calls return a pair (i, j) indicating that the given sequence has the form $)^i\,(^j$ after conceptually removing all immediate pairs. The rules for combining two of these are simple. Given two sequences of the form $)^i\,(^j$ and $)^k\,(^\ell$:

- If $j \leq k$, then their concatenation has the form $)^{i+k-j}\,(^\ell$.

- If $j > k$, then their concatenation has the form $)^i\,(^{\ell+j-k}$.

In terms of implementation, we need to be able to split a sequence in half. We could do this with `take` and `drop`, but it's much cleaner to use `splitMid`. We also need `Primitives.par` for parallelism – the code `Primitives.par (fn () ⇒` e_1`, fn () ⇒` e_2`)` indicates the parallel pair $(e_1 \parallel e_2)$.

Algorithm 2.8. *Divide-and-conquer parenthesis matching.*

```
1  fun parenMatch S =
2    let
3      fun parenMatch′ S =
4        case Seq.splitMid S of
5          Seq.EMPTY ⇒ (0,0)
6        | Seq.ONE Paren.L ⇒ (0,1)
7        | Seq.ONE Paren.R ⇒ (1,0)
8        | Seq.PAIR (A,B) ⇒
9            let val ((i,j),(k,ℓ)) =
10                 Primitives.par (fn () ⇒ parenMatch′ A,
11                                 fn () ⇒ parenMatch′ B)
12           in if j ≤ k then (i+k−j, ℓ) else (i, ℓ+j−k)
13           end
14   in
15     parenMatch′ S = (0,0)
16   end
```

Let's now analyze cost bounds. On input of size n, we split the problem into two subproblems of size $n/2$ and solve them in parallel, then perform a little bit of arithmetic. Assuming the `ArraySequence` implementation, splitting requires $O(1)$ work and span. We can clearly do the arithmetic in $O(1)$ work and span. Each of the subproblems has $W(n/2)$ work and $S(n/2)$ span. Recall that we *add* the work of parallel subcomputations, while taking the *max* of their spans, resulting in $2W(n/2)$ work and $S(n/2)$ total span for computing the subproblems.

2.5 Additional Exercises

Exercise 2.9. *As implied by the name, the* `ArraySequence` *implementation of sequences lays out its elements in an array. Describe how to implement* `splitMid` *(and in general,* `subseq`*) in* $O(1)$ *work and span.*

Exercise 2.10. *Carefully prove Observation 2.7.*

.