15-150 Fall 2013
Lecture 19

Stephen Brookes *

Thursday, 31 October

1 Topics

- Functional programming and parallelism
- Cost semantics
- Sequences

*Building on notes by Dan Licata.

2 Introduction

The big picture

Before we start, let's think about the big picture of parallelism. Parallelism is relevant to situations where many things can be done at once—e.g. using the multiple cores in multi-processor machine, or the many machines in a cluster. Overall, the goal of parallel programming is to describe computation in a way that allows us to exploit the potential for doing work on multiple processors simultaneously. At the lowest level, this means deciding, at each step, what to do on each processor. These decisions are constrained by the data dependencies in a problem or a program. For example, evaluating `(1 + 2) + (3 + 4)` takes three units of work, one for each addition, but you cannot do the outer addition until you have done the inner two. So even with three processors, you cannot perform the calculation in fewer than two time-steps. That is, the expression has work 3 but span 2.

Now, one way to do parallel programming is to say explicitly what to do on each processor at each time-step — by giving what is called a *schedule*. There are languages that let you write out a schedule explicitly, but there are disadvantages to this approach. For example, when you buy a new machine, you may need to adapt your program that was written for (say) 4 processors to the new machine's larger number of processors, maybe 16, or 64, or a million. Moreover, it's tedious and boring to think about assigning work to processors, when what you really want think about is the problem you're trying to solve. After all, we might reasonably expect the scheduling details to be something a smart compiler can best figure out, but it's clearly the programmer's job to design an algorithm that best solves the problem.

Our approach

The approach to parallelism that we advocate in this class (and is further developed in 15-210) is based on raising the level of abstraction at which you can think, by *separating algorithm specification (and work/span analysis) from scheduling*. You, the programmer, worry about specifying what work there is to do, and how much potential for parallelism there is (that's what the span is concerned with); the compiler should take care of scheduling the work onto processors.

Three things are necessary to make this *separation of concerns* work:

1. The code you write to implement an algorithm must not "bake in" a schedule; design your code to avoid unnecessary data dependencies.

2. You must be able to reason about the evaluation *behavior* of your code independently of the schedule.

3. You must be able to reason about the *time complexity* of your code independently of the schedule.

Our central tool for avoiding schedule-baking is *functional programming*. Let's explain why functional programming is suitable.

First, taking advantage of higher-order functions, we can focus on *bulk operations* on (potentially large) *collections* of data. In many applications we want to combine the data in a collection (such as a list, or a tree) using a binary operation (such as `+`) which is *associative*, so it doesn't matter in what order we pairwise combine the data. If we can represent the collection as an abstract type with efficient bulk operations for this kind of combination, it's easy to write code that avoids being overly specific about evaluation order. Lists are *not* very well suited for this task, because the built-in combination operations on lists (`foldl` and `foldr`) bake in a sequential evaluation order. For trees we can define "tree folding" operations as higher-order functions in which the left and right subtrees get combined independently, and as we saw earlier the tree fold operations have better span than the list fold operations. But we can do even better than trees.

Today and in the next lectures we will talk about *sequences*. We will use the (math) notation `<x1,...,xn>` for a sequence of length `n` containing the items `x1` through `xn`. Our implementation of sequences includes a bunch of operations, including a `map` operation, a higher-order function similar in spirit to `List.map`, with the following (equational) specification [1]:

```
map f <x1, x2, . . ., xn> = <f x1, f x2, . . ., f xn>.
```

In terms of *evaluation*,

```
map f <x1, x2, . . ., xn> =>* <v1,. . ., vn>
if  f x1 =>* v1, . . ., f xn =>* vn.
```

[1] We extend the notion of extensional equality to sequences in the obvious way: two sequences are equal if they have the same length, and contain equal values at all positions.

This description also implicitly specifies the data dependencies (none!): to calculate `map f <x1, . . ., xn>`, you need to calculate `f x1`, ..., `f xn` and there are no data dependencies. And we didn't specify any particular schedule or evaluation strategy! Our description makes it clear that there is plenty of room for parallel evaluation, but also that other evaluation strategies may be available. You *could* implement the evaluation of `map f <x1, . . ., xn>` sequentially, by evaluating `f x1` first, then `f x2`, and so on; that would produce a result consistent with the above description. And if you had enough parallel processing power you *could* implement the evaluation by using a separate processor for each `f xi`, and again you would end up with the same final result. Our algorithmic description based on `map` was agnostic as to scheduling. If we had written the code for evaluating this expression as an iterative (sequential) loop, on an inherently sequential data structure such as a list, we would have been gratuitously throwing away opportunities for exploiting parallelism. But using `map` like this – a bulk operation on a collection data structure – we avoided this problem.

Second, as noted in the above example, functional programming focuses on pure, mathematical functions, and evaluation has no side effects. This limits the dependence of one chunk of work on another to what it is obvious from the data-flow in the program; all that matters from a prior evaluation is the value it returned, and if there's no data dependency we can do work in parallel and be sure to get the same results. For example, when you `map` a function `f` across a sequence, evaluating `f` on the first element has no influence on the value of `f` on the second element, etc. This is not the case for imperative programming, where one call to `f` might influence another via memory updates (or side effects).

So functional programming is well suited for algorithm design without commitment to a schedule.

Our central tool for reasoning about *behavior*, independently of schedule, is *deterministic parallelism*: the (extensional) behavior of your program is *the same* for any schedule! This is true because functional programs have a well-defined mathematical meaning independent of any implementation: the values produced by expression evaluation don't depend on what scheduling decisions occurred during evaluation. For example, assuming that `f` is a total function and the `xi` are suitably typed, we say that

```
map f <x1,...,xn> = <f x1, ... , f xn>
```

because the left-hand and right-hand expressions evaluate to equal sequence

values. We don't need to say here exactly how the left-hand expression or the right-hand expression take steps, and indeed there are many possible stepping strategies that are consistent with this equation. In all cases the left- and right- expressions evaluate to *equal* final values, and that's what the equation says.

So functional programming nicely supports reasoning about behavioral correctness, independently of scheduling.

Our central tool for reasoning about time complexity, independently of the schedule, is a *cost semantics*. This involves the asymptotic work/span analyses that we have been doing all semester. The cost semantics lets you reason abstractly, yet actually implies some concrete constraints on how well any potential scheduler might be able to divide work among processors. For example, the cost semantics for `map` will say that when `s` is an integer sequence of length n, `map (fn x => x + 1) s` takes $O(n)$ work but $O(1)$ span—each function application can be done in parallel. As we will, see work and span analysis can be phrased in terms of *cost graphs*, which can be used to reason abstractly about the running time of your program for any schedule, and are also a useful data structure for explaining the ideas behind scheduling.

Caveat programmer

However, there is an important *caveat*: even with today's technology, this methodology based on separation of concerns — you design the algorithm, the compiler schedules it — is not always going to deliver good performance. It's hard to get parallel programs to run quickly in practice, and many smart researchers are actively working on this problem. Some of the issues include: *overhead* (it takes time to distribute tasks to processors, notice when they've completed, etc.); *spatial locality* (we want to ensure that needed data can be accessed quickly); and *schedule-dependence* (the choice of schedule can sometimes make an asymptotic difference in time or space usage). So don't get the impression that writing programs our way will magically guarantee good performance.

There are some implementations of functional languages that address these issues, to varying degrees of success: Manticore, MultiMLton, and PolyML are implementations of Standard ML that have a multi-threaded run-time, so you *can* run the sequence code that we will write in parallel. But it's tricky to get actual speedups. NESL is a research language by Guy Blelloch (who designed 15-210), and that's where a lot of the ideas that

we will discuss today originated; there is a real implementation of NESL and some benchmarks with actual speedups on multiple processors. GHC, a Haskell compiler, implements many of the same ideas, and you can get some real speedups there too. Scala is a hybrid functional/object-oriented language, and one of the TAs implemented sequences in Scala and has gotten some performance gains on the 210 assignments.

Rationale

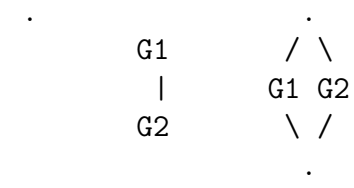
So why are we teaching you this style of parallel programming? There are two reasons: First, even if you have to get into more of the gritty details of scheduling to get your code to run fast today, it's good to be able to think about problems at a high level first, and then figure out the details. If you're writing code for an internship this summer using a low-level parallelism interface, it can be useful to first think about the abstract algorithm—what are the data dependencies, and what can be done in parallel?—and then figure out the details. You can use parallel functional programming to design algorithms without worrying about scheduling, and then translate down to whatever interface you need. Second, it's our thesis that eventually this kind of parallel programming will be practical and common: as language implementations improve, and computers get more and more cores, this kind of programming will become feasible, and even necessary in order to fully exploit the potential speed-up of using parallelism. You're going to be writing programs for a long time, and we're trying to teach you tools that will be useful years down the road.

The Plan

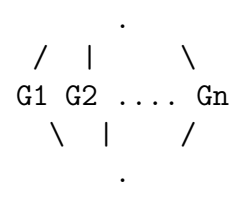
In the next two lectures, we will discuss three things: *cost semantics*, in more detail; *sequences*, an important data structure with good parallel complexity; and we use these tools to build an `n`-*body simulation*.

3 Cost Semantics

A cost graph is a form of *series-parallel graphs*. A series-parallel graph is a directed graph (we always draw a cost graph so that the edges point down the page) with a designated source node (no edges in) and sink node (no edges out), formed by two operations called sequential and parallel composition. The particular series-parallel graphs we need are of the following form:



The first is a graph with one node (both the source and the sink).
The second is the *sequential combination* of graphs `G1` and `G2`, formed by by putting an edge from the sink of `G1` to the source of `G2`; its source is the source of `G1` and its sink is the sink of `G2`. The third is the *parallel combination* of graphs `G1` and `G2`, formed by adding a new source and sink, and adding edges from the source to the sources of `G1` and `G2`, and from the sinks of `G1` and the sink of `G2` to the new sink. We also use n-ary versions of sequential and parallel composition for cost graphs. We draw an n-ary parallel combination of graphs `G1 ... Gn` as



Again, a new source and a new sink, with edges from the new source to the sources of the `Gi`, and edges from the sinks of the `Gi` to the new sink.

(Unfortunately, it's not easy to draw these pictures using `verbatim` mode.)

When drawing pictures of cost graphs, we sometimes put arrows on the arcs to emphasize the flow of control. In the diagrams above we relied on verticality to do the same job.

The *work* of a cost graph `G` is the number of nodes in `G`. The *span* of `G` is the length of the longest path from the source of `G` to the sink of `G`, which we may refer to as the *critical path*, or the *diameter* of the graph. We will

associate a cost graph with each closed program, and define the work/span of a program to be the work/span of its cost graph.

These graphs model *fork-join parallelism*: a computation forks into various subcomputations that are run in parallel, but these come back together at a well-defined joint point. These forks and joins are well-nested, in the sense that the join associated with a later fork precedes the join associated with an earlier fork.

For example, the expression

```
(1 + 2)
```

has cost graph

This says that the summands `1` and `2` are evaluated in parallel; because `1` and `2` are already values, the middle graphs have only one node. After the parallel combination, there is one step for doing the addition. The work of this graph is 5 and the span is 4. (Note: these numbers are an additive constant-factor higher than we may have said earlier in the course for the work and span of the *expression* `(1+2)`, because in the *graph* we count the fork and the join as separate nodes, and we count a step for evaluating a value to itself; asymptotically, this minor detail doesn't matter.)

We can link cost graphs with our earlier discussions of work and span for expressions, as follows. Recall that we add the work and add the span for code fragments that need to be executed in sequential order; just as the number of nodes in the sequential composition or the parallel composition of `G1` and `G2` is the sum of the numbers of nodes in the two subgraphs. And for independent code fragments we combine spans with *max* rather than +, just as the span of the parallel composition of two graphs corresponds to the max of the spans of the individual graphs.

3.1 Brent's Principle

Cost graphs can be used to reason abstractly about the time complexity of your program, independently of the schedule. For example, below, we associate a cost graph with each operation on sequences. You can reason about the work and span of code that uses these operations to manipulate sequences, via these graphs, without worrying about any particular schedule.

But you may well ask: what do work and span predict about the *actual* running time of your code? The work predicts the running-time when evaluated sequentially, using a single processor; the span predicts the running time if you have "infinitely many" processors and the scheduler always uses parallel evaluation for independent computations. Of course in practice you are unlikely to have infinitely many processors (and it's even unlikely that you're always going to have "enough" processors). What can we say, based on work and span, about what happens when you evaluate ML code using the 2 processors in your laptop, or the 1000 in your cluster, relying on whatever scheduling strategy is implemented on these machines? The answer is that there is a provable bound on the best running time that can be achieved, asymptotically:

> **Brent's Theorem**
> An expression with work w and span s can be evaluated on a p-processor machine in time $O(max(w/p, s))$.

The intuition behind Brent's Theorem is that the best (most efficient) way to employ p processors is to try dividing the total work w up into chunks of size p (there are w/p such chunks) and do these chunks one after another; but the data dependencies may prevent this from being feasible, and you can never do better than the span s.

For example, if you have 10 units of work to do and span 5, you can achieve the span on 2 processors. If you have 15 units of work, it will take at least 8 steps on 2 processors. But if you increase the number of processors to 3, you can achieve the span 5. If you have 5 units of work to do, the fact that you have 2 processors doesn't help: you still need 5 steps.

Brent's Theorem can also tell us useful information about the potential for speed-up when multiple processors are available. If your code has work w and span s, calculate the smallest integer p such that $w/p \leq s$. Given this many processors, you can evaluate your code in the best possible time. Having even more processors yields no further improvement. (Of course these

are all *asymptotic* estimates, and in practice constant factors do matter! So you may not see real speedups consistent with these numbers.)

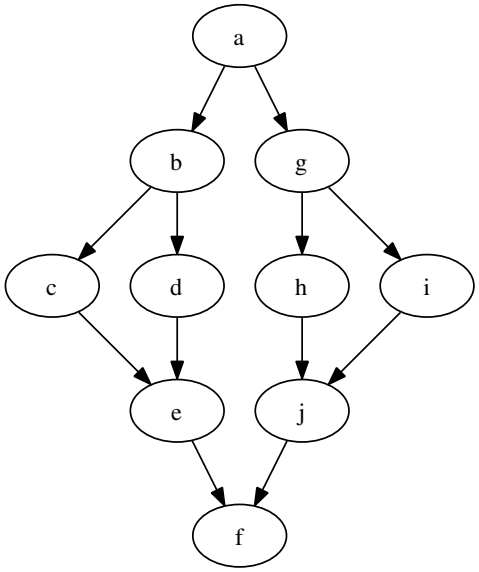
Brent's theorem should hold for any language you design; if not, you got the cost semantics wrong! Thus, we will sometimes refer to *Brent's Principle*: a language should be designed so that Brent's Principle is in fact a theorem.

3.2 Scheduling

Cost graphs are also a helpful data structure for scheduling work onto processors. Let's take a look at how a compiler might do this.

A schedule can be generated by *pebbling* a cost graph. To schedule a graph onto p processors, you play a pebble game with p pebbles. The rules of the game are: in each step, you can pick up a pebble from a node or from your hand and place it on a node all of whose predecessors have been visited, marking that node as visited. To generate a *schedule* that says what each pebble (processor) does in each timestep, we divide the pebbling up into steps; at each step you can play at most p pebbles. The nodes played on are the units of work completed in that timestep. The restriction to playing on nodes whose predecessors have been visited ensures that dependencies are respected. The nodes whose predecessors have been visited, but have not themselves been visited, form the *frontier*, the currently available work.

Consider the following cost graph:



A 2-pebbling with pebbles X and O might start out like this:

Step	**X**	**O**
1	a	
2	b	g
3	c	

In the first step, we can only play a pebble on the source (all of its predecessors have been pebbled, trivially, because it has no predecessors), because no other node in the graph is available. In this step to a processor is idle because there is not enough available work that can be done. In the second step, we can play on both of the next two nodes. In the third step, we can play on any of the four nodes at the next level, but we can (and did) choose to play on only one of them. This is a step in which a processor was idle idle, even though there is work that could have been done. A *greedy schedule* assigns as many processors as possible work at each time step. (A non-greedy scheduler might be more efficient overall if not all units of work took the same time.) The schedule outlined above is not greedy!

For a fixed number of processors p we can identify two particularly natural scheduling algorithms, known as $pDFS$ (p depth-first search) and $pBFS$ (p breadth-first search). A DFS schedule prefers the left-most bottom-most available nodes, whereas a BFS schedule prefers higher nodes (but then tackles them left-to-right). At each step, you play as many pebbles (a maximum of p) as you can, subject to availability of work.

Here is a schedule for the cost graph above, using 2DFS (giving preference to processor X when only one unit of work is available):

Step	**X**	**O**
1	a	
2	b	g
3	c	d
4	e	h
5	i	
6	j	
7	f	

In step 4, we prefer node e to node i because e is lower (bottom-most). In the remaining steps there is only one node available to work on.

Here's a schedule for the same graph, using 2BFS:

Step	X	O
1	a	
2	b	g
3	c	d
4	h	i
5	e	j
6	f	

This time, in step 4 we prefer i to e because it is higher. Consequently, in step 6, two units of work are available to do, so we can finish in 6 steps instead of 7.

Thus: different scheduling algorithms can give different overall run-time. Additionally, they differ in how many times a processor stops working on one computation and switches to working on an unrelated computation. For example, in the 2-BFS schedule, step 4, both processors "jump" over to the other side of the graph. This can be bad for spatial reasons (cache performance may be worse) but the details are subtle. And actually in this kind of small example the differences aren't really likely to be noticeable. Of course, this may not be the case in more realistic examples.

4 Sequences

To introduce sequences, here is a signature containing some of the main operations that we'll begin with.

```
signature SEQ =
sig
   type 'a seq
   exception Range
   val tabulate : (int -> 'a) -> int -> 'a seq
   val length : 'a seq -> int
   val nth : int -> 'a seq -> 'a
   val map : ('a -> 'b) -> 'a seq -> 'b seq
   val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
   val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end
```

Let's assume we have an implementation of this signature, a structure `Seq:SEQ`. For any type `t`, the type `t Seq.seq` has values that represent sequences of values of type `t`. Sequences are *parallel collections*: ordered collections of values, with parallelism-friendly operations on them. Don't think of sequences as being implemented by lists or trees (though you could implement them as such); think of them as a new built-in abstract type with only the operations we're about to describe (the operations mentioned in the signature `SEQ`). The differences between sequences and lists or trees show up in the cost of these operations, which we specify below.

We write `Seq.seq`, `Seq.map`, etc. to refer to the `seq` type and the `map` function defined in our given structure named `Seq`. This use of qualified names will help us to avoid confusion with the built-in ML `map` function on lists (which we could also refer to using the qualified name `List.map`).

Intuitively, the sequence operations have names that suggest that they do the same things as the corresponding operations on lists that you are familiar with. However, they have different work and span than the corresponding list functions. Firstly, sequences admit *constant-time* access to items — `nth i s` takes constant time. Secondly, sequences have better parallel complexity—many operations, such as `map`, act on each element of a sequence in parallel.

For each sequence operation, we (1) describe its behavior abstractly and (2) give a cost graph, which specifies the work and span.

Before we start, recall that we use the math notation

```
<v1, . . . , vn>
```

for a sequence of length `n` whose entries are `v1` through `vn`. Actually, since the `nth` function uses zero-based indexing, it is sometimes more convenient to mimic this and write

```
<v0, . . ., v(n-1)>
```

tabulate

The operation for building a sequence is *tabulate*, which constructs a sequence from a function that gives you the items at each position, the positions being indexed from 0 up to a specified bound. So `tabulate f n` builds a sequence of length `n` with 0'th item `f 0`, 1st item `f 1`, and so on. So we can characterize this operation equationally:

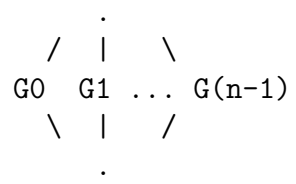
```
tabulate f n = <f 0, . . ., f (n-1)>
```

and its evaluation properties are summarized by:

```
tabulate f n =>* <v0 , ... , v_(n-1)>
   if f 0 =>* v0, f 1 =>* v1, . . ., f (n-1) =>* v_(n-1).
```

(Again, this characterization does *not* imply any specific order of evaluation.)

The cost graph for `tabulate f n` is



where each `Gi` is the cost graph for `f i`. So when `f i` is constant time, the work for `tabulate f n` is $O(n)$ and the span is $O(1)$.

nth

We define the behavior of `nth` as

```
nth i <x0 , ... , xn-1>  = xi             if 0 <= i < n
                         = raise Range    if i>=n
```

For a sequence value `s` and integer value `i`, the cost graph for `nth i s` is trivial:

As a consequence of the structure of this cost graph, it is obvious that `nth i s` has $O(1)$ work and $O(1)$ span. Thus, as promised, sequences provide constant-time access to elements.

length

The behavior of `length` is given by:

```
length <x1 , ... , xn> = n
```

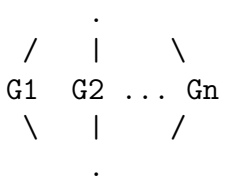
The cost graph for `length s` when `s` is a sequence value is also trivial:

And again as a consequence, `length s` has $O(1)$ work and span.

map
The behavior of `map f <x1,...xn>` is given by

```
map f <x1 , ... , xn> = <f x1,...,f xn>
```

Each function application may be evaluated in parallel. This is shown by the cost graph:



where we let `G1` be the cost graph for `f x1`, etc. (This cost graph resembles the cost graph for tabulate.)

As a consequence, if `f x` takes constant time, and `s` is a sequence value with length n, then `map f s` has $O(n)$ work and $O(1)$ span.[2]

reduce
`reduce` is intended to be used with an *associative* binary function `g` of type `t * t -> t`, for some type `t`, a value `z:t`, and a sequence of items of type `t`. The function `g` is associative iff for all values `a,b,c` of type `t`,

```
g(a, g(b, c)) = g(g(a, b), c).
```

Associativity implies that if we pairwise combine a sequence of values using `g`, in any order that respects the original enumeration order, we'll get the same result. For example,

```
g(g(x1, x2), g(x3, x4)) = g(x1, g(x2, g(x3, x4))).
```

When `g` is associative, let's write

```
x1 g x2 g . . . g xn
```

[2]Unfortunately, there is no known way to express the time complexity of the `map` function itself, abstractly — we lack a theory of asymptotic analysis for higher-order functions.

for the result of (any such) combination. (This math notation is chosen because it deliberately abstracts away from parenthesization, and resembles the ML notation when `g` is actually an *infix* function.)

Assuming that `g` is associative, `reduce g z <x1, . . ., xn>` has the following equational characterization:

$$\texttt{reduce g z}\ \langle x_1, \ldots x_n \rangle = x_1\ \texttt{g}\ x_2\ \texttt{g}\ \ldots\ \texttt{g}\ x_n\ \texttt{g z}$$

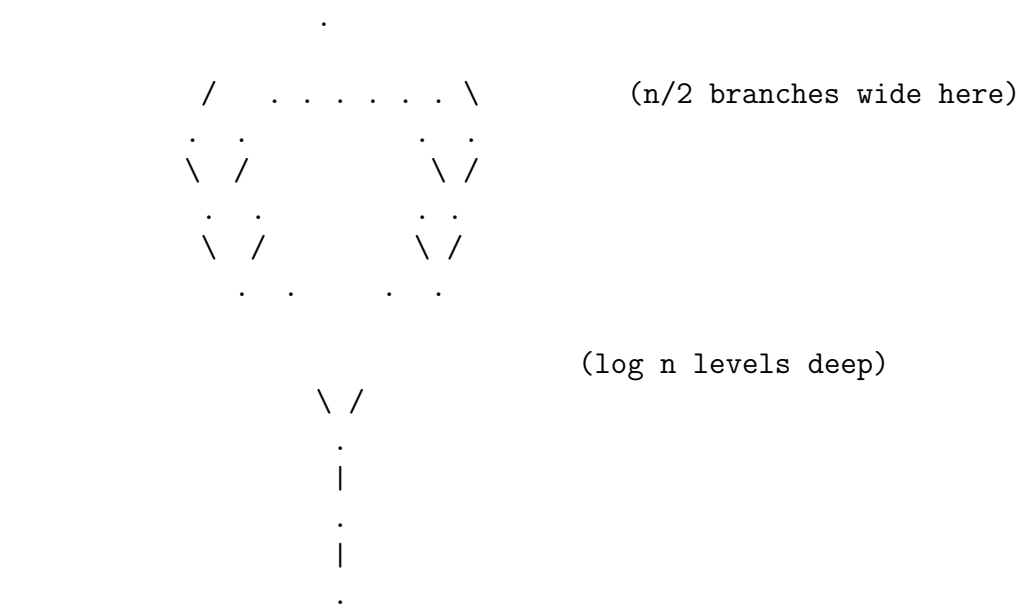
The implementation of `reduce` is assumed to use a balanced parenthesization format for the pairwise combinations, so for example the cost graph of

```
reduce g z <x1, x2, x3>
```

is the same as the cost graph for

```
g(g(x1, x2), g(x3, z)).
```

If `g(a,b)` is constant time, for all values `a` and `b`, it follows that the cost graph for `reduce g z` $\langle x_1, \ldots x_n \rangle$ looks something like



Consequently, the graph has size (work) $O(n)$ and critical path length (span) $O(\log n)$. This computation does not have *constant* span, because later uses of `g` need the results of earlier combinations: there are data dependencies, because we can only combine two items at a time.

Contrast this with lists and `foldl` or `foldr`. When `L` is a list of length n, `foldr g z L` has work $O(n)$ and also span $O(n)$. Similarly for `foldl`.

mapreduce
`mapreduce` is intended to be used with a function `f` of type `t1 -> t2` for some types `t1` and `t2`, an *associative* binary function `g` of type `t2 * t2 -> t2`, a value `z:t2`, and a sequence of items of type `t1`.

The behavior of `mapreduce f z g <x1, . . ., xn>` (assuming that `g` is associative) is given by:

$$\texttt{mapreduce f z g}\,\langle x_1, \ldots x_n\rangle = (\texttt{f}\ x_1)\ \texttt{g}\ (\texttt{f}\ x_2)\ \texttt{g}\ \ldots\ \texttt{g}\ (\texttt{f}\ x_n)\ \texttt{g z}$$

The implementation of `mapreduce` uses a balanced parenthesization format for pairwise combinations, as with `reduce`. Its work and span are as for `reduce`, when `f` and `g` are constant time.

5 Examples

In these examples, let's assume we have `open`-ed the structure `Seq:SEQ`, so we no longer need to use qualified names.

- Building an empty sequence: the function

  ```
  empty : unit -> 'a seq
  ```

  is defined by

  ```
  fun empty ( ) = tabulate (fn _ => raise Range) 0
  ```

  It's easy to see (using the specs for `tabulate`) that `empty( )` defined like this does return a sequence of length zero.

- You can implement a `cons` operation for sequences, with tabulate. This function has type `cons : 'a -> 'a seq -> 'a seq`, and we specify that `cons x xs` returns a sequence with length one more than the length of `xs`, whose first (0'th) element is `x`, and whose remaining items are the items of `xs` shifted by one:

  ```
  fun cons (x : 'a) (xs : 'a seq) : 'a seq =
    tabulate (fn 0 => x | i => nth (i-1) xs) (length xs + 1)
  ```

- A function `first : int -> int seq` that constructs the sequence consisting of the first `n` integers:

  ```
  fun first n = tabulate (fn x:int => x) n
  ```

  For $n > 0$, `first n` returns the sequence value

  ```
  <0, 1, . . ., n-1>
  ```

  For $n \leq 0$ what happens?

- A function `squares : int -> int seq` that constructs the sequence consisting of the first `n` square numbers:

  ```
  fun squares n = map (fn x => x*x) (first n)
  ```

- Note that we could have defined `squares` with a single tabulate, rather than a tabulate followed by a map, as in:

  ```
  fun squares n = tabulate (fn i => i*i) n
  ```

  There is a general "fusion" law that expresses this kind of result. When `f : int -> t1` and `g : t1 -> t2` are total functions, it follows that `g o f : int -> t2` is also total, and for all `n:int`,

  ```
  map g (tabulate f n) = tabulate (g o f) n
  ```

  The `squares` example is a special case (let `f` be `fn x:int => x`, and `g` be `fn x:int => x*x`).

- Similarly, when `f:t1 -> t2` and `g : t2 => t3` are total functions, so is `g o f : t1 => t3`, and

  ```
  map (g o f) = (map g) o (map f)
  ```

  As an example, consider

  ```
  fun square x = x*x;
  fun quads s = map square (map square s)
  ```

Here `quads` is extensionally equal to `(map square) o (map square)`. And `quads` is extensionally equal to `map (square o square)`. Since `square o square` is extensionally equal to `fn x:int => x*x*x*x`, we have

```
quads = map (fn x:int => x*x*x*x)
```

- Counting, revisited.
  In the first lecture, we defined some (recursive) ML functions that operate on lists of integers, combining the integers together by adding them. Later we saw how to use `foldl` and `foldr` to do the same job. Now we can revisit these ideas using sequences.

  Here is a function `sum : int seq -> int` for adding the integers in a sequence:

  ```
  fun sum (s : int seq) : int = reduce (op +) 0 s
  ```

  (Note that `(op +)` is associative!)

  And a function `count : int seq seq -> int` for combining the integers in a sequence of integer sequences:

  ```
  fun count (s : int seq seq) : int = sum (map sum s)
  ```

  `sum` takes an integer sequence and adds up all the numbers in it using `reduce`, just like we did with folds for lists and trees. `count` sums up all the numbers in a sequence of sequences, by (1) summing each individual sequence and then (2) summing the sequence that results.

  (Exercise: rewrite `count` with `mapreduce`, so it takes only one pass).

  We can now see that `count` on a sequence of n sequences, each of length n, requires $O(n^2)$ work: `sum s` is implemented using `reduce` with constant-time arguments, and thus has $O(n)$ work and $O(\log n)$ span, where n is the length of s. Each call to `sum` inside the `map` is on a sequence of length n, and thus takes $O(n)$ work. This function is mapped across n rows, yielding $O(n^2)$ work for the `map`. The row sums also form a sequence of length n, so the final `sum` contributes $O(n)$ more work, which is subsumed by the $O(n^2)$. So the total work here is $O(n^2)$,

as promised. However, the span is $O(\log n)$: the `map` doesn't contribute anything, and both the inner and outer `sum`s are on sequences of length n, and therefore have $O(\log n)$ span. The total span is the sum of the inner span and the outer span, because of the data dependency: the outer additions happen after the inner sums have been computed. The sum of $\log n$ and $\log n$ is still $O(\log n)$, so the total span is $O(\log n)$.

(In the lecture slides this example is explained using cost graphs, and you should compare the two work/span derivations to make sure you understand how cost graphs can be used.)

- Reversing a sequence, a nice use of re-indexing:

```
fun reverse (s : 'a seq) : 'a seq =
    tabulate (fn i => nth (length s - i - 1) s) (length s)
```

This implementation of reverse has linear work and constant span! But index calculations tend to be hard to read and get right, so try to avoid them when there's an obvious alternative (or prove that you got them right!).

Note that with `nth` and `tabulate` you can write very index-y array-like code. Use this style sparingly: it's hard to read! For example, never write something like

```
tabulate (fn i => ... nth i s ...) (length s)
```

if the function doesn't otherwise mention `i`: you'd be reimplementing `map` in a hard-to-read way! And you would probably be able to express your algorithmic intentions more elegantly as something like

```
map (fn x => ... x ...) s
```