Chapter 1

Introduction

parallel

The topic of this course might best be described as **problem solving with computers**. Suppose that you have a problem to solve and unlike in the good old days, you ^ also have a (parallel) computer to help you. In the beginning your primary concern is to solve the problem correctly. But it is also important to solve the problem quickly, because otherwise the problem becomes irrelevant.

For example, when you are trying to use your recently improved AHI (Artificial Homework Intelligence) application to do your homeworks for you, both correctness and timeliness are important, because if you turn in incorrect answers or turn in after the deadline, you might as well have partied your time away on the beach downtown. Less radically, if you are building a mobile application to help you understand your parents, the application server which runs on the cloud better respond to you on time before you have to talk to your parents soon. OK, well more realistically, you don't want to wait for your quicksort implementation to run on your computer for an hour, because you cannot play your favorite game while doing that.

As such, this course is about several aspects of such problem solving with computers. It is about defining the problem you want to solve. It is about learning the different techniques that can be used to solve such a problem, and about designing algorithms using these techniques. It is about designing abstract data types that can be used in these algorithms, and data structures that implement these types. And, it is about analyzing the cost of those algorithms and comparing them based on their cost. Unlike most courses on algorithms and data structures, which just consider sequential algorithms, in this course we will focus on how to design and analyze parallel algorithms and data structures.

In the rest of this chapter we will consider why it is important to study parallelism, why it is important to separate interfaces from implementations, and outline some algorithm-design techniques.

1.1 Parallelism

Parallel computation is the ability to run multiple sub-computations (tasks) at the same time.

Question 1.1. *Why should we care about parallelism?*

There are many reason for why parallelism is important. Fundamentally, parallelism is simply more powerful than sequential computation, where only one sub-computation can be run at a time. Allowing sub-computations to run at the same time enables solving bigger and more interesting problems in a shorter amount of time.

Another important reason is efficiency in terms of energy usage.

Question 1.2. *Do you know how much energy it takes to run a computation twice as fast using a sequential computer (one line of computation)?*

As it turns out, performing a computation twice as fast sequentially requires eight times as much energy. Precisely speaking, energy consumption is a cubic function of clock frequency (speed). With parallelism we don't need more energy to speed up a computation, at least in principle. For example, to perform a computation in half the time, we need two lines of computation (instead of one) that run for half the time as the sequential computation, thus consuming the same amount of energy. In reality, there are some overheads and we will need more energy, usually only a constant fraction more. These two factors—time and energy—have gained importance in the last decade catapulting parallelism to the forefront of computing.

Remark 1.3. *As is historically popular in explaining algorithms, we can establish an analogy between parallel algorithms and cooking. As in a kitchen with multiple chefs, in parallel algorithms you can do things in parallel for faster turnaround time. For example, if you want to cook 3 sophisticated dishes with a team of top chefs you can do so by asking each of 3 chefs to cook one. Doing so will often be faster that using one chef. But there are some overheads, you have to, for example, explain to each chef what you expect. Obviously, you also need more resources, at the very least you have to provide each chef a stove and probably some pots and pans.*

Parallel hardware. Today, it is nearly impossible to avoid parallelism. For example, when you do a simple web search, you are engaging a data center in some part of the world (likely near your geographic location) that houses thousands of computers.

Question 1.4. *Do you know how many computers are engaged in answering a typical web search?*

Many of these computers (perhaps as many as hundreds, if not thousands) take up your query and sift through data to give you an accurate response as quickly as possible. This form of parallelism may be viewed as large-scale parallelism, as it involves a large number of computers.[1]

Another form of parallelism involves smaller numbers of processors. For example, portable computers today have chips that have 4, 8, or more processor cores. Such chips, sometimes called *multicore chips*, are predicted to spread and provide increasing amount of parallelism over the years. For example, using current chip technology, it is not difficult to put together several multicore chips in a desktop machine to include 64 cores. While multicore chips were initially used only in laptops and desktops, they are now used in almost all smaller mobile devices such as phones (many mobile phones today have 4- or 8- core chips.)

In addition to the aforementioned parallel systems, there has been much interest in developing *domain-specific hardware* for specific applications. For example, graphics processing units (GPUs) can fit as many as 1000 small cores (processors) onto a single chip.

Question 1.5. *Can you think of consequences of these developments in hardware?*

These developments in hardware make the specification, the design, and the implementation of parallel algorithms an important topic.

Question 1.6. *What is the advantage of using a parallel algorithm instead of a sequential one?*

Parallel software. The most important advantage of using a parallel instead of a sequential algorithm is the ability to perform sophisticated computations quickly enough to make them practical or relevant. For example, without parallelism, computations such as Internet searches, realistic graphics, climate simulations would be prohibitively slow. One way to quantify such an advantage is to measure the performance gains that can be achieved with parallelism.

Example 1.7 illustrates the sort of performance gains that can achieved today. These times are on a 32 core commodity server machine. In the table, the sequential timings use sequential algorithms while the parallel timings use parallel algorithms. Notice that the speedup for the parallel 32 core version relative to the sequential algorithm ranges from approximately 12 (minimum spanning tree) to approximately 32 (sorting).

Question 1.8. *But why after all, do we have to do anything differently to take advantage of parallelism?*

[1]Of course, terms such as "large" are relative by definition. What we call large-scale today may be consider small scale in the future.

Example 1.7. *Example timings for some algorithms on 1 and on 32 cores.*

	Sequential	***Parallel***	
		1-core	*32-core*
Sorting 10 million strings	*2.9*	*2.9*	*.095*
Remove duplicates for 10 million strings	*.66*	*1.0*	*.038*
Minimum spanning tree for 10 million edges	*1.6*	*2.5*	*.14*
Breadth first search for 10 million edges	*.82*	*1.2*	*.046*

Running time in seconds.

Challenges of parallel software. It would be convenient to use sequential algorithms on parallel computers, but this does not work because parallel computing requires a somewhat different way of organizing the algorithm than sequential computing. The fundamental difference is that two computations can be performed in parallel only if they do not depend on each other. Thus when designing a parallel algorithm, we have to carefully identify the underlying dependencies between different computations to be performed.

Example 1.9. *Suppose that you want to run many searches on a database of student records. To improve your search time, you decide to sort the records by the student name so that you can use a fast binary search algorithm to find each student. Since the binary search has to wait for the sort to complete, you cannot sort and search in parallel. You can however perform all the searches in parallel. You can also sort in parallel by using a parallel sorting algorithm.*

We call this first challenge the algorithm-design challenge. The second challenge is somewhat more practical. The many forms of parallelism, ranging from large to small scale, and from general to special purpose, currently requires many different programming languages, libraries, and implementation techniques. For example, it is unlikely one can obtain the speedups that we discussed in Example 1.7 from unoptimized software. This diversity of hardware and software makes it difficult to learn both the high-level ideas of developing parallel algorithms and at the same time how to develop highly optimized parallel code for a particular machine. For example, we can easily spend weeks or even months talking about how we might optimize a parallel sorting algorithm on a GPU. Furthermore developing highly optimized code can obscure the key high-level ideas about parallelism.

Parallel Thinking. Maximizing speedup by highly tuning an implementation is not the goal of this book. Here, we aim to cover general design principles for parallel algorithms that can be applied in essentially all parallel systems, from the data center to the multicore chips on mobile phones. We will learn to think about parallelism at a high-level, learning general techniques for designing parallel algorithms and data structures, and learning how to approximately analyze their costs. The focus is on understanding when things can run in parallel, and when not due to dependences. There is much more to learn about parallelism, and we hope you will take further courses on the topic.

Work and Span. In this course we analyze the cost of algorithms in terms of two measures: *work* and *span*. Together these measures capture both the sequential time and the parallelism available in an algorithm. We typically analyze both of these asymptotically, using for example the big-O notation, which will be described in more detail in Chapter 4.

The ***work*** of an algorithm corresponds to the total number of primitive operations performed by an algorithm. If running on a sequential machine, it corresponds to the sequential time. However, when running in parallel it might be possible to share the work among multiple processors, therefore reducing the time. The interesting question is to what extent can the work be shared. Ideally we would like the work to be evenly shared. If we had W work and P processors to work on it in parallel, then even sharing would imply each processor does $\frac{W}{P}$ work, and hence the total time is $\frac{W}{P}$. An algorithm that achieves such ideal sharing is said to have ***perfect speedup***. Perfect speeedup, however, is not always possible.

Question 1.10. *Can you come up with an example where perfect speedup is not possible?*

If our algorithm is fully sequential (each operation depends on prior operations, leaving no room for parallelism), for example, we can only take advantage of one processor, and the time would not be improved at all by adding more. There is no sharing—-at least in parallel.

The purpose of the second measure, *span*, is to analyze to what extent the work of an algorithm can be shared among processors. The ***span*** of an algorithm basically corresponds to the longest sequence of dependences in the computation. It can be thought of the time an algorithm would take if we had an unlimited number of processors on an ideal machine. However, we do not have an unlimited number of processors, but instead probably care how well the algorithm performs on some fixed number of processors. When designing and analyzing algorithms in terms of work and span we do not directly control how the computations are mapped onto these processors. Instead we assume there is a runtime scheduler that dynamically schedules the various parts of the computation onto processors as the algorithm runs.

What is particularly attractive about analyzing work and span is that by using just these two measures we can roughly predict the time of an algorithm on any number of processors by just assuming a “reasonable” runtime scheduler. Furthermore we can predict the largest number of processors for which we can get close to perfect speedup. This is called the *parallelism of the algorithm*. But what is a reasonable scheduler? Consider a scheduler that runs on P processors and that has the following greedy property: no processor will sit idle (not work on some part of the computation) if there is work ready to do. Such a scheduler is called a ***greedy scheduler***. It can be shown that when using a greedy schedule on a computation with W work and S span, the runtime will be at most:

$$T \leq \frac{W}{P} + S \; .$$

If the first term dominates, then we are getting close to perfect speedup (within a factor of two). We therefore define the ***parallelism*** $\mathcal{P}$ of an algorithm to be the number of processors at which the two terms are equal, which gives:

$$\mathcal{P} = \frac{W}{S} \; .$$

For $P = \mathcal{P}$ we are getting half of perfect speedup. For any $P < \mathcal{P}$ the speedup is better, and for any $P > \mathcal{P}$ it is worse. Therefore $\mathcal{P}$ gives us a rough upper bound on the number of processors we can effectively use.

Example 1.11. *As an example, consider the mergeSort algorithm for sorting a sequence of length n. The work is the same as the sequential time, which you might know is*

$$W(n) = O(n \log n) \; .$$

We will show that the span for mergeSort is

$$S(n) = O(\log^2 n) \; .$$

The parallelism is therefore

$$\mathcal{P} = O\left(\frac{n \log n}{\log^2 n}\right) = O\left(\frac{n}{\log n}\right) \; .$$

This means that for sorting a million keys, we can effectively make use of quite a few processors: $10^6/(\log_2 10^6) \approx 50,000$. In practice we might not be able to make good use of this many because of overheads, and constants hidden in the big-O, but we are still likely to make good use of over 1000 processors.

Question 1.12. *How do we calculate the work and span of an algorithm?*

In this book we will calculate the work and span of algorithms in a very simple way that just involves composing costs across subcomputations. Basically we assume that sub-computations are either composed sequentially (one is done after the other) or in parallel (they can be done at the same time). Informally, the rules are simply the following:

	work (W)	**span** (S)
Sequential composition	$1 + W_1 + W_2$	$1 + S_1 + S_2$
Parallel composition	$1 + W_1 + W_2$	$1 + \max(S_1, S_2)$

where W_1 and S_1 are the work and span of the first subcomputation and W_2 and S_2 of the second. The 1 that is added to each rule is the cost of composing the computations.

These rules are hopefully intuitive since whether we do things sequentially or in parallel the total work always adds. However, if we are doing things in parallel, then the work will still add, but for the span we have to wait for the longer of the two sub-computations to finish, and hence we have to take the maximum of their spans.

If algorithm A has less work than algorithm B, but has greater span then which algorithm is better? In analyzing sequential algorithms there is only one measure so it is clear when one algorithm is asymptotically better than another, but now we have two measures. In general the work is more important than the span. This is because the work reflects the total cost of the computation (the processor-time product). Therefore typically the goal is to first reduce the work and then reduce the span. However sometimes it is worth giving up a little in work to gain a large improvement in span.

We say that a parallel algorithm is ***work efficient*** if the work is asymptotically the same as the time for an optimal sequential algorithm. For example, the parallel mergeSort described in Example 1.11 is work efficient since it does $O(n \log n)$ work, which optimal time for comparison based sorting. In this course we will try to develop work-efficient or close to work-efficient algorithms.

In summary there are several advantages of analyzing algorithms in terms of work and span.

1. The work tells us the sequential time of an algorithm.
2. The span tells us the least time required to complete the algorithm on an unbounded number of processors.
3. The work and span give us a rough sense of how many processors we can effectively make use of (i.e. get close to perfect speedup).
4. We do not have to worry about scheduling the computations onto processors.
5. It is relatively easy to calculate work and span by composition.

Example 1.13. *Going back to our culinary example, suppose that we have 30 eggs to cook using 3 top chefs. Whether we ask all 3 chefs to do the cooking or just one, the total work remains unchanged: 30 eggs need to be cooked. On the other hand, the span represents the longest sequence of dependences. Each of the 3 chefs can cook 10 eggs at the same time, giving us a span of 10 eggs plus some time to divide up the eggs. (If we break one, we might have to run to the grocery store, which may blow up our span.) More formal rules are given in Chapter 4.*

Remark 1.14. Truth in advertising. *Work and span only give a rough estimate of cost and parallelism. There are many costs that are not included in the analysis, such as the overhead for a scheduler, and the cost for communicating among processors. These are topics of a more advanced course.*

1.2 Specification and Implementation

In this course we try to carefully distinguish between specifications and implementation. We will do this both for individual functions (e.g. a sorting function), and for sets of functions that act on a common data type (e.g. a set of operations on a priority queue). A ***specification*** (sometimes called ***interface***) defines precisely what we want of a function or of the data. An ***implementation*** describes how to meet the specification. In other words specifications and implementations refer to the *what* and the *how*: what we want a function or data type to achieve and how to do that. The goal of the specification is to abstract away from the specific implementation and capture just what is needed to use a function. There may be many ways to implement an abstract specificantion—i.e. many hows for a given what.

What do we need to specify? Certainly we need to specify what a function does—its functionality. We might specify that a function squares an integer, or sorts a sequence of numbers. In this book, however, we also care about the cost of functions, for example, how much work and span is needed to sort n integers (asymptotically). We might be tempted to say that the cost is associated with a particular implementation. However as a user of a sort function, one often cares about the cost without caring about the implementation. As long as the sort takes $O(n \log n)$ work and $O(\log^2 n)$ span, for example, we do not care if it is a mergesort or a upside-down inside-out reversion sort? Indeed the implementation we use might be some complicated hybrid sort. To avoid needing to know how the implementation works, and allow us to change implementations over time, we would like a cost specification for the function that is independent of the implementation.

To sum up, we have considered three levels of abstraction—the functional specification, the associated cost specifications, and implementations that match it. There might be multiple cost specifications for a given functional specification, and then multiple implementations that match each cost specification. As mentioned before, these levels can either be used for individual functions, or for sets of functions on common data types. For an individual function we refer to the functional specification as an ***algorithmic problem*** (or simply ***problem***), a cost specification as the ***function cost***, and an implementaiton as an ***algorithm***. For a set of functions we refer to the functional specification as a ***data type***, the cost specification as the ***operation costs***, and an implementaiton as an ***data structure***. This is summarized with the following table:

	Specification		**Implementation**
	Functionality	**Cost**	
Functions	Problem	Function Cost	Algorithm
Data	Data Type	Operation Costs	Data Structure

We now consider some examples to clarify these ideas. The first is sorting, and we can define comparison-based sorting as follows.

Problem 1.15 ((comparison) Sorting). *Given a sequence S of n elements taken from a totally ordered set with comparison $\leq$, return a sequence R containing the same elements but such that $R[i] \leq R[j]$ for $0 \leq i < j < n$.*

We them might have the following cost specification:

Cost Specification 1.16 (Parallel Sort). *The cost for parallel sorting on a sequence of length n and assuming the comparison function $<$ does constant work, is $O(n \log n)$ work and $O(\log^2 n)$ span.*

Note that in the cost specification we had to be clear about the cost of the comparison operation. In this book we make significant use of functions that take other functions as arguments. We therefore will often have to be careful about the cost of those functions. We might have another cost specificaiton:

Cost Specification 1.17 (Shallow Sorting). *The cost for parallel sorting on a sequence of length n and assuming the comparison function $<$ does constant work, is $O(n^2)$ work and $O(\log n)$ span.*

We note that this specification has asymptotically larger work but smaller span. As discussed in the last section, we typically first care about work and hence we should prefer the first specification, but there might be cases where this version is better.

We can now specify an algorithm, such as the following insertion sort.

Algorithm 1.18 (Insertion Sort).

```
$sort(f, S)$ =
  **if** $|S| = 0$ **then** $\langle\ \rangle$
  **else** $insert(f, S[0], sort(S[1, \ldots, n-1]))$
```

where f is the comparison function and S is the input sequence. The algorithm uses a function insert(f, e, S) that takes the comparison function f, an element e, and a sequence S sorted by f, and inserts e in the appropriate place. This is itself an algorithmic problem, since we are not specifying how it is implemented, but just specifying its functionality. We might also be given a cost specification, and let's say for a sequence of length n the cost of insert is $O(n)$ work and $O(\log n)$ span. Given this cost we can determine the overall asymptotic cost of sort using our composition rules described in the last section. Since the code does the inserts one after the other sequentially, we need to add both the work and span. Since there are n inserts the algorithm will have $n \times O(\times n) = O(n^2)$ work and $n \times O(\log n) = O(n \log n)$ span.

Similarly an ***abstract data type*** (ADT) specifies precisely an interface for operating on data in an abstract form. The specification will typically consist of a set of functions for accessing or manipulating the particular data type along with a definition of what each function does. The specification, however, does not specify how the data is structured or how the functions are implemented. This is hidden by the ADT. A ***data structure***, on the other hand, implements the specification by organizing the data in a particular form, typically in a way that allows an efficient implementation of the functions.

Example 1.19. *For example, a priority queue is an ADT with functions that might include `insert`, `findMin`, and `isEmpty`. Various data structures can be used to implement a priority queue, including binary heaps, arrays, and balanced binary trees.*

Some ADTs we will cover in this course include: sequences, sets, ordered sets, tables, priority queues, and graphs.

The terminology ADTs versus data structures is not as widely used as problems versus algorithms. In some other textbooks the term data structure is used to refer to both the specification and the implementation. We will try to avoid such ambiguous usage in this book.

Question 1.20. *Why is it important to distinguish between specification and implementations?*

There are several critical reasons for keeping a clean distinction between specification and implementation. Perhaps most importantly we want people to be able to use a specified problem or data type without having to know how it is implemented. In many cases the specification is quite simple, but an efficient algorithm or data structure that implements it is very complicated. The specification therefore abstracts the implementation. Secondly the implementation might change over time. As long as each implementation matches the same specification, and the user relied only on the specification, then he or she can continue using the new implementation without worrying about their code breaking. Thirdly, when we compare the performance of different algorithms or data structures it is important that we are not comparing apples with oranges. We have to make sure the algorithms we compare are solving the same problem, because subtle differences in the problem specification can make a significant difference in how efficiently that problem can be solved.

For these reasons, in this course we will put a strong emphasis on defining precise and concise specifications and then implementing those specifications using algorithms and data structures. When discussing solutions to problems we will emphasize general techniques that can be used to design them, such as divide-and-conquer, the greedy method, dynamic programming, and balance trees. It is important that in this course you learn how to design your own algorithms and data structures given a specification, and even how to specify your own problems and ADTs given a task at hand.

1.3 Algorithm-Design Techniques

One of the most difficult and probably most important tasks for any computer-science student to learn is to come up with their own algorithms. Students often point out that they often do not know where to start when designing an algorithm or data type that matches a given specification and cost bound.

Question 1.21. *Have you designed algorithms before? If so what approaches have you found helpful?*

Given an algorithmic problem, where do you even start? It turns out that most of the algorithms follow several well-known techniques. Here we will outline some such techniques (or approaches), which are key to both sequential and parallel algorithms. We note that the definition of these techniques is not meant to be formal, but rather just a guideline.

Remark 1.22. *To become good at designing algorithms, we would recommend gaining as much experience as possible by both formulating problems and solving them by applying the techniques that we discuss here.*

Brute Force. The brute force technique involves trying all possible (underlying) solutions to a problem. In particular the technique enumerates all candidate solutions, and for each it checks if it is valid, and either returns the best valid solution, or any valid solution.

For example, to sort a set of keys, we can try all permutations of those keys and test each one to see if it is sorted. We are guaranteed to find at least one that is sorted.

Question 1.23. *When might we find more than one valid solution (permutation) for the sorting problem? In this case do we care which one?*

However, there are $n!$ permutations and $n!$ is very large even for modest n—e.g. $100! \approx 10^{158}$. Therefore this approach to solving the sorting problem is not "tractable" for large problems, but it might be a viable approach for small problems. In some cases the number of candidate solutions is much smaller. In a later chapter, we will see that a brute force method for the maximum contiguous subsequence sum (MCSS) problem only requires trying about n^2 candidate solutions on a string of length n. However, as we will see, this is still not an efficient approach to solving the problem.

Question 1.24. *Does the brute-force technique parallelize well?*

The brute force technique is typically easy to parallelize. Most often enumerating all solutions (e.g. all permutations) is easy to do in parallel, and testing the solutions is inherently parallel. However, this does not mean it leads to efficient parallel algorithms. In a parallel algorithm we primarily care about the total work and only then about the level of parallelism.

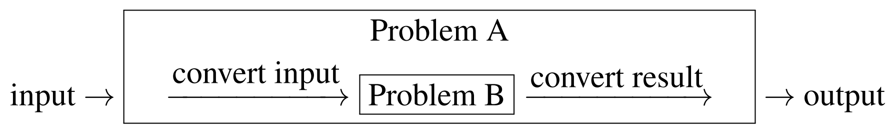
Question 1.25. *Given that the brute-force algorithms are often inefficient, why do we care about them?*

Despite its inefficiency, there are two reasons for why brute-force algorithms are often helpful. First, the brute force technique can be very useful is when testing the correctness of more efficient algorithms. Even if inefficient for large n the brute force technique could work well for testing small inputs. The brute-force technique is usually the simplest solution to a problem, but not always. The second reason is that brute-force algorithms are often easy to design and are a good starting point toward a more efficient algorithm.

Note that when solving ***optimization problems***, i.e., problems where we care about the optimal value of a solution, the problem definition might only return the optimal value and not the "underlying" solution that gives that value. For example, a shortest path problem on graphs might be defined such that it just returns the shortest distance between two vertices, but not a path that has that distance. Similarly when solving ***decision problems***, i.e., problems where we care about whether a solutions exists or not, the problem definition might only return a Boolean indicating whether a solution exists and not the solution. For both optimization and decision problems of this kind when we say try all "underlying solutions" we do not mean try all possible return values, which might just be a number or a Boolean, we mean try all possible solutions that lead to those values, e.g., all paths from a to b.

Exercise 1.26. *Describe a brute force algorithm for deciding whether an integer n is composite (not prime).*

Reducing to another problem. Sometimes the easiest approach to solving a problem is to reduce it to another problem for which known algorithms exist. More precisely, to reduce a problem A to a problem B, we use some preprocessing to convert the input of problem A to an input for problem B, then solve problem B on that input, and finally convert the result of problem B back to the result of problem A. This can be pictured as follows:



For example consider a function `maxVal` that finds the maximum of a set of numbers.

Question 1.27. *How would you use this function to find the minimum of a set of values?*

To implement another function, `minVal`, that finds the minimum of a set of numbers we could simply invert the sign of all the numbers, use `maxVal` on those numbers, and then invert the sign of the result. In general reductions can be much more interesting, and they can go between problems that look very different. In Chapter 3, for example, we reduce a problem on strings of characters to one on graphs.

When reducing one problem to another it is important to include the cost of converting the input and converting the result. Indeed to calculate the total work for reducing a problem A to B we can just add the work of the two conversions to the cost of an algorithm for solving B. We will say a reduction of A to B is efficient if the conversion takes no more work or span (asymptotically) than B. Thus an efficient reduction of problem A to problem B tells us that problem A is effectively as easy as problem B (at least within a constant factor). In the theoretical analysis of algorithms, reductions between problems is also often used in the other direction to show that some problem is at least as hard as another. In particular, if we know (or conjecture) that problem A is hard (e.g. requires exponential work), and we can efficiently reduce it to problem B (e.g. using polynomial work), then we know that B must also be hard. Indeed the theory of NP-complete problems is based on this idea.

Inductive techniques. The idea behind inductive techniques is to solve one or more smaller instances of the same problem, typically referred to as *subproblems*, to solve the original problem. The technique most often uses recursion to solve the subproblems. Common techniques that fall in this category include the following.

- ***Divide and conquer.*** Divide the problem on size n into $k > 1$ independent subproblems on sizes $n_1, n_2, \ldots n_k$, solve the problem recursively on each, and combine the solutions to get the solution to the original problem. It is important than the subproblems are independent; this makes the approach amenable to parallelism because independent problems can be solved in parallel.
- ***Contraction.*** For a problem of size n generate a significantly smaller (contracted) instance (e.g., of size $n/2$), solve the smaller instances recursively, and then use the results to solve the original problem. Contraction only differs from divide and conquer in that it allows there to be only one subproblem.
- ***Dynamic programming.*** Like divide and conquer, dynamic programming divides the problem into smaller subproblems, solves the subproblems, and combines the solutions to the subproblems. The difference, though, is that the solutions to subproblems are reused multiple times. It is therefore important to store the solutions for reuse by building up a table of solutions.
- ***Greedy.*** For a problem on size n use some approach to find the "best" element by some greedy metric, pull this element out, and solve the problem on the remaining $n-1$ elements.

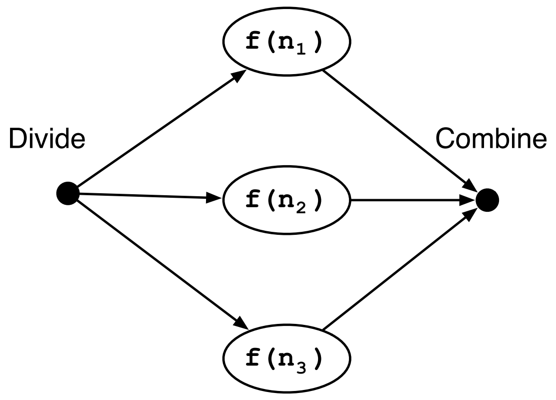


Figure 1.1: Structure of a divide-and-conquer algorithm illustrated ($k = 3$).

Randomization. Randomization is a powerful algorithm design technique that can be applied along with the aforementioned techniques. It often leads to simpler algorithms.

Question 1.28. *Do you know of a randomized divide-conquer algorithm?*

Randomization is also crucial in parallel algorithm design as it helps "break" the symmetry in a problem without global communication.

Question 1.29. *Have you heard of the term "symmetry breaking"? Can you think of a somewhat amusing real-life phenomena, where we engage in randomized symmetry breaking?*

We often perform randomized symmetry breaking when walking on a crowded sidewalk with many people coming in our direction. With each person we encounter, we may pick a random direction to turn and the other person responds, or the other way. If we recognize each other late, we may end up in a situation where the randomness becomes more apparent, as we attempt to get past each other but make the same (really opposite) decisions. Since we make essentially random decisions, the symmetry is eventually broken—or we run into each other.

We will cover several examples of randomized algorithms in this course. Formal cost analysis for randomized algorithms requires knowledge of probability theory, so in Chapter 9 we will spend some time covering the required probability theory.

1.4 What makes a good algorithmic solution?

When you encounter an algorithmic problem, you can look into your bag of techniques and, with practice, you will find a good solutions to the problem. When we say a *good solution* we mean:

1. **Correct:** Clearly correctness is most important.
2. **Low cost:** Out of the correct solutions, we would prefer the one with the lowest cost.

Question 1.30. *How do you show that your solution is good?*

Ideally you should prove the correctness and efficiency of your algorithm. For example, in exams, we might ask you do this. Even in real life, we would highly recommend making a habit of this as well. It is often too easy to convince yourself that a poor algorithm is correct and efficient.

Question 1.31. *How can you prove correct an algorithm designed by using an inductive technique? Can you think of a proof technique that might be useful?*

Algorithms designed with inductive techniques can be proved correct using (strong) induction.

Question 1.32. *How can you analyze an algorithm designed by using an inductive technique? Can you think of a technique that might be useful?*

We can often express the complexity of inductive algorithms with recursions and solve them to obtain a closed-form solution.

1.5 Pidgin ML

Pid·gin (Merriam-Webster) :
a language that is formed from a mixture of several languages when speakers of different languages need to talk to each other

In this book we describe our algorithms with code that is loosely based on the ML class of languages (SML, Caml, F#), however we include a significant amount of mathematical notation in the code, and sometimes include some English descriptions. We have made an effort to make the code, including mathematical notation, precise, except when English is used. We define our basic syntax and semantics here, and describe additional syntax and semantics as we need it. We refer to the code as pidgin ML. Although some authors like to make a distinction between algorithms and programs, we try not to do this in this book. The programs are the formal descriptions of the algorithm.

Remark 1.33 (The lambda calculus)**.** *As with most functional programming languages, the ML class of languages are based on the* **lambda calculus** *(or λ calculus), a computational model developed by Alonzo Church in 1932. The lambda calculus is a very simple language consisting of expressions e which can only have three forms:*

x,	:	*a* variable name,
$(\lambda\, x \,.\, e)$	:	*a* function definition*, where x is the argument and e is the body, or*
$e_1\ e_2$	:	*a* function application*, where e_1 and e_2 are expressions;*

and effectively only a single rule for processing expressions, called **beta reduction***. For any function application for which the left hand expression is a function definition, beta reduction "applies the function" by making the transformation:*

$$(\lambda\, x \,.\, e_1)\ e_2 \longrightarrow e_1[x/e_2]$$

where $e_1[x/e_2]$ roughly means for every (free) occurrence of x in e_1, substitute it with e_2. Computation in the lambda calculus consists of applying beta reduction until there is nothing left to reduce.

In the early 30s Church argued that anything that can be "effectively computed" can be computed with the lambda calculus, and therefore that it is a universal mechanism for computation. However, it was not until a few years later when Alan Turing developed the Turing machine and showed its equivalence to the lambda calculus that the concept of universality became widely accepted. The fact that the models were so different, but equivalent in what they can compute, was a powerful argument for the universality of the models. We now refer to the hypothesis that anything that can be computed can be computed with the lambda calculus, or equivalently the Turing machine, as the **Church-Turing hypothesis***, and refer to any computational model that is computationally equivalent to the lambda calculus as* **Church-Turing complete***.*

Although the lambda calculus allows beta reduction to be applied in any order, most functional programming languages use a specific order. The ML class of languages use call-by-value, which means that for a function application $(\lambda\, x \,.\, e_1)\ e_2$, the expression e_2 must be evaluated to a value before applying beta reduction. Other languages, such as Haskell, allow for the beta reduction before e_2 becomes a value. If during beta reduction e_2 is copied into each variable x in the body, this reduction order is called call-by-name, and if e_2 is shared, it is called call-by-need. Call-by-name is inefficient since it creates redundant computations, while call-by-need is both inherently sequential and not well suited for analyzing costs. In this book we, therefore, only use call-by-value. All these reduction orders are Church-Turing complete.

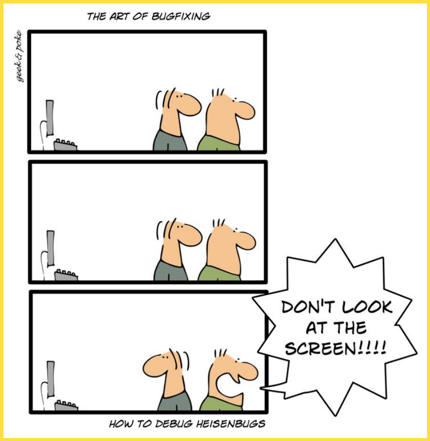
Functional algorithms. Perhaps most importantly all the algorithms in this book are purely "functional". Here by "functional" we are not referring to whether an algorithm works or not, but rather to a style of algorithms, or computations, that do not have side effects and use higher-order functions. A computation that does not have ***side effects*** is often referred to as being ***pure*** and means that any sub-computation evaluates to a value while not having any effects on the rest of the computation. This is as opposed to imperative algorithms which are based on side-effecting other parts of the computation by using shared modifiable state. A ***higher-order function*** is a function that takes another function as an argument or returns a function. With higher-order functions, functions can be created during the computation, stored in data structures, and generally treated in any way other data is.

Since this is not a course on programming, we will not belabor the general advantages of functional algorithms and programming, but we do mention two properties of functional algorithms that are particularly useful for parallelism, and these are the reasons we choose to use a functional language in this book.

Question 1.34. *Can you think of reasons for why purely functional programming can help in designing and implementing parallel algorithms?*

The first property is that (purely) functional algorithms are safe for parallelism. In particular any components can be executed in parallel without affecting each other. In an imperative setting, we need to worry about what effects the parallel components have on each other via modifiable shared state since, depending on the exact relative timing of components, these effects might cause computations to return very different results. Such effects that can change outcomes based on timing are called ***race conditions***. Race conditions make it much harder to reason about the correctness and the efficiency of parallel algorithms. They also make it harder to debug parallel code since each time the code is run, it might give a different answer.

Remark 1.35.
The term **Heisenbug** *was coined in the early 80s to refer to a type of bug that goes away when you try to locate or study it and comes back when you are not studying it. They are named after the famous Heisenberg uncertainty principle which roughly says that if you localize one property, you will loose information about another complementary property. Often the most difficult Heisenbugs to find have to do with race conditions in parallel or concurrent code. These are sometimes also called concurrency bugs.*



The second property is that higher-order functions (even in a language that is not pure) help with parallel thinking since they encourage high-level parallel constructs. For example,

Definition 1.36 (pidgin ML expressions)**.**

$$
\begin{array}{rcll}
e & := & v & \textit{primitive values} \\
 & | & x & \textit{variables} \\
 & | & \textbf{fn}\ p \Rightarrow e & \textit{functions} \\
 & | & e_1\ e_2 & \textit{function application} \\
 & | & e_1\ op\ e_2 & \textit{infix operations} \\
 & | & e_1\ ,\ e_2 & \textit{sequential pair} \\
 & | & e_1\ ||\ e_2 & \textit{parallel pair} \\
 & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 & \textit{if then else} \\
 & | & \textbf{let}\ b^+\ \textbf{in}\ e\ \textbf{end} & \textit{local bindings} \\
 & | & (e) & \textit{parenthesization}
\end{array}
$$

Every e with or without a subscript is another expression. The syntax for p and b will be defined below, and the superscript $^+$ indicates that the b is repeated one or more times.

instead of thinking about a loop that iterates over the elements of an array to generate the sum, which is completely sequential, higher-order functions allow us (or even encourage us) to define a "reduce" function. In addition to taking the array as an argument, the reduce takes a binary associative function as another argument. It then sums the array based on that binary associative function. The advantage for parallelism is that the reduce can use a tree of sums instead of summing one after the other. It also allows for any binary associative function (e.g. maximum, minimum, multiplication, ...). In general, thinking in higher order functions encourages working at a higher level of abstraction, moving us away from the one-at-a-time (loop) way of thinking that is detrimental to parallelism.

We note that even though the algorithms that we design are purely functional, this does not mean that they cannot be coded in imperative languages—one just needs to be much more careful when coding imperatively. Some imperative parallel languages, in fact, encourage programming purely functional algorithms. The techniques that we describe thus are applicable in the imperative setting as well.

Syntax and Semantics. ***Syntax*** refers to the structure of the program (algorithm) itself, while ***semantics*** refers to what the program computes. In the book since we are interested in analyzing the cost of algorithms, we are interested in not just what algorithms compute, but how the algorithm compute. Semantics that capture how algorithms compute are called operational semantics, and when augmented with specific costs, cost semantics. Here we will interleave describing both the syntax and an informal description of the operational semantics. We will talk about cost semantics in Chapter 4.

The syntax of pidgin ML consists of expressions, which can have various forms as specified in Definition 1.36. Expressions are defined recursively (inductively) as many of the forms can contain other expressions as their parts. We will define more forms of expressions as we need them. A complete list of expressions is given in the Appendix. Some syntax is simply to make it easier (sweeter) to read or write code without adding any real power. This is often referred to as ***syntactic sugar***.

The semantics of pidgin ML involves "evaluating" expressions. Every closed expression (no undefined variables) either evaluates to a value, or does not terminate. In functional languages the values include arbitrary functions. Evaluating an expression is performed by starting with the whole program, which is an expression, inductively evaluating its parts, and then putting these parts together to form the result. We now define the syntax and semantics (informally) for each expression.

The ***primitive values*** in pidgin ML include the set of integers (denoted as $\mathbb{Z}$), the set of Boolean values (i.e. true and false, denoted as $\mathbb{B}$), and a variety of built in functions on them such as *plus, minus, multiply, divide,* and *negate* on integers and *not, or,* and *and* on Booleans. It might seem odd to refer to functions as primitive values, but in functional languages functions are indeed values and can be passed around in the same way. Primitive values evaluate to themselves.

The ***variables*** are names that are bound to some value by their context (by either a surrounding **let** or function definition). In pidgin ML variable names consist of only alphabetic characters (a-z and A-Z), and optionally end with some number of "primes", or with a single character subscript which can be an alphabetic character or digit (0-9). For example X', X_1, X_l and *myVar* are valid variable names, but *myVar1*, *my-var* and *my_var* are not. In a given expression a variable might not be bound (its context might be outside the expression). In this case the variable is said to be ***free*** in the expression. An expression is ***closed*** if it has no free variables. A variable evaluates to the value to which it is bound.

A ***function application expression***, $e_1\ e_2$, applies the function generated by evaluating e_1 to the value generated by evaluating e_2. In particular lets say e_1 evaluates to the function f and e_2 evaluates to the value v, then we apply f to v, as described in more detail below. All functions in pidgin ML have exactly one argument, but an argument can be a pair. For example the expression $f(a, b)$ applies the function f to the ordered pair of a and b, and the expression $g(a, b, c)$ is equivalent to $g(a, (b, c))$ and applies the function g to the pair whose first element is a and second element is the pair of b and c. A function application evaluates to the result of applying f to v, defined in more detail below.

An ***infix expression***, $e_1\ op\ e_2$, involve two expressions and an infix operator op. The infix operators include $+$ (plus), $-$ (minus), $\times$ (multiply), $/$ (divide), $<$ (less), $>$ (greater), $\vee$ (or), and $\wedge$ (and). We will introduce more as we need them, and a complete list is given in the appendix. For all these operators the infix expression $e_1\ op\ e_2$ is just syntactic sugar for $f(e_1, e_2)$ where f is the function corresponding to the operator op (see parenthesized names that follow each operator above). We use standard precedence rules on the operators to indicate how they get parsed. For example in the expression

$$3 + 4 \times 5$$

Definition 1.37 (pidgin ML patterns)**.**

$$\begin{array}{rcll} p & := & x & \textit{variable} \\ & \mid & (p) & \textit{parenthesization} \\ & \mid & p_1, p_2 & \textit{pairs} \end{array}$$

the $\times$ has a higher precedence than $+$ and therefore the expression is equivalent to $3 + (4 \times 5)$. Furthermore all operators are left associative unless stated otherwise. That is to say that $a\ op_1\ b\ op_2\ c = (a\ op_1\ b)\ op_2\ c$ if op_1 and op_2 have the same precedence. For example

$$5 - 4 + 2$$

will evaluate to $(5-4)+2 = 3$ not $5-(4+2) = -1$ since $-$ and $+$ have the same precedence.

There are two special infix operators: "," and "||", for generating ordered pairs either sequentially or in parallel. The ***comma*** operator "," as in the infix expression (e_1, e_2), evaluates e_1 and e_2 sequentially, one after the other, and returns the ordered pair consisting of the two resulting values. The ***parallel*** operator "||", as in the infix expression $(e_1 \mid\mid e_2)$, evaluates e_1 and e_2 in parallel, at the same time, and returns the ordered pair consisting of the two resulting values. The two operators are identical in terms of what they return. However, when we talk about cost models in the next chapter we will see they have different costs since one is sequential and the other parallel. The comma and parallel operators have the weakest, and equal, precedence.

A ***lambda expression***, **fn** $p \Rightarrow e$, defines an unnamed function where p is a pattern containing the variables of the argument and e is an expression for the body. The pattern can be a single variable or can contain multiple variables in a ***pattern*** as defined in Definition 1.37. When a function is applied to an argument, the pattern is used to separate the argument into its parts. For example if function **fn** $(x, y) \Rightarrow e$ is applied to the pair $(2, 3)$ then x is given value 2 and y is given value 3. Any free occurrences of the variables x and y in the expression e will now be bound to the values 2 and 3 respectively. We can think of function application as substituting the argument (or its parts) into the free occurrences of the variables in its body e.

Definition 1.40 (pidgin ML bindings)**.**

$$\begin{array}{rcll} b & := & x(p) = e & \textit{function binding} \\ & \mid & p = e & \textit{variable binding} \end{array}$$

Example 1.38. *The expression:*

$$(\textbf{fn}\ (x, y) \Rightarrow x/y)\ (8, 2)$$

evaluates to 4 *since* 8 *and* 2 *are bound to* x *and* y*, respectively, and then divided. The expression:*

$$(\textbf{fn}\ (f, x) \Rightarrow f(x, x))\ (\texttt{plus}, 3)$$

evaluates to 6 *since* f *is bound to the function* `plus`*,* x *is bound to* 3*, and then* `plus` *is applied to the pair* $(3, 3)$*. The expression:*

$$(\textbf{fn}\ x \Rightarrow (\textbf{fn}\ y \Rightarrow x + y))\ 3$$

evaluates to a function that adds 3 *to any integer.*

Remark 1.39.

The definition

$$(\textbf{fn}\ x \Rightarrow (\textbf{fn}\ y \Rightarrow f(x, y)))$$

takes a function f *of a pair of arguments and converts it into a function that takes one of the arguments and returns a function which takes the second argument. This technique can be generalized to functions with multiple arguments and is often referred to as* **currying***, named after Haskell Curry (1900-1982), who developed the idea. It has nothing to do with the popular dish from Southern Asia, although that might be an easy way to remember the term.*

An ***if-then-else expression***, **if** e_1 **then** e_2 **else** e_3, evaluates the expression e_1, which must return a Boolean. If the value of e_1 is true then the result of the if-then-else expression is the result of evaluating e_2, otherwise it is the result of evaluating e_3. This allows for conditional evaluation of expressions. Later we will also add the **case** expression that allows conditional evaluation.

The ***let expression***, **let** b^+ **in** e **end**, consists of a sequence of bindings b^+, which define local variables, followed by an expression e, in which those bindings are visible. The syntax for a single binding b is given in Definition 1.40, and the superscript $+$ means that b is repeated one or more times. Each binding b is either a variable binding or a function binding. Each ***variable binding***, $p = e$, consists of a ***pattern***, p, on the left and an expression, e, on the right. The expression is evaluated and its value is assigned to the variable(s) in the pattern. The value of the expression must therefore match the structure of the pattern. For example if the pattern on the left is a pair of variables (x, y) then the expression on the right must evaluate to a pair. The two elements of the pair are assigned to the variables x and y, respectively. Each ***function binding***, $x(p) = e$, consists of a function name, x (technically a variable), the arguments for the function, p, which are themselves a pattern, and the body of the function, e. A let expression **let** b^+ **in** e **end** evaluates to the result of evaluating e giving the variable bindings defined in b.

Example 1.41. *In the following expression:*

```
1  let
2      $x = 2 + 3$
3      $f(w) = (w \times 4, w - 2)$
4      $(y, z) = f(x - 1)$
5  in
6      $x + y + z$
7  end
```

line 2 binds the variable x to $2 + 3 = 5$; line 3 defines a function $f(w)$ which returns a pair; line 4 applies the function f to $x - 1 = 4$ returning the pair $(4 \times 4, 4 - 2) = (16, 2)$, which y and z are bound to, respectively (i.e., $y = 16$ and $z = 2$); and finally in line 6 x, y and z are added giving $5 + 16 + 2$. The result of the expression is therefore 23.

We need to be careful about defining which variables each binding can see, as this is important in being able to define recursive functions. In pidgin ML the expression on the right of each binding in a **let** can see all the variables defined in previous variable bindings, and can see the function name variables of all binding (including itself) within the **let**. Therefore the function binding $x(p) = e$ is not equivalent to the variable binding $x =$ **fn** $p \Rightarrow e$, since in the prior x can be used in e and in the later it cannot. Function bindings therefore allow for the definition of recursive functions. Indeed they allow for mutually recursive functions since the body of function bindings within the same **let** can reference each other.

Example 1.42. *The expression:*

```
let
    $f(i)$ = if $(i < 2)$ then $i$ else $i \times f(i)$
in
    $f(5)$
end
```

will evaluate to the factorial of 5, *i.e.,* $5 \times 4 \times 3 \times 2 \times 1$, *which is* 120.

Definition 1.43 (Pidgin ML types)**.**

$$\begin{array}{rcll} t & := & \mathbb{Z} \mid \mathbb{B} & \textit{primitive types} \\ & \mid & t_1 \times t_2 & \textit{product types (pairs)} \\ & \mid & t_1 \to t_2 & \textit{function types} \end{array}$$

Types. In pidgin ML all values have a type and we assume that pidgin ML is statically typed, such that if an expression evaluates it will always generate a value of the same type. We can think of types as disjoint sets of values [2]. For example the set of integers is the integer type $\mathbb{Z}$, and the set $\{true, false\}$ is the Boolean type $\mathbb{B}$.

In set theory the Cartesian product of two sets A and B is the set of all ordered pairs from A and B. The type of a pair is therefore the Cartesian product of its elements. For example a pair consisting of an integer and a Boolean has type $int \times bool$. This is often called a product type. Functions also have types representing the mapping of their input to their output. If a function has input type A and resulting type B then its type is $A \to B$. A definition of these types is given in Definition 1.43.

It is often useful to have a value be one of many alternative types—for example, either an integer or a Boolean. For this purpose we use ***variant types*** (sometimes called disjoint union or tagged union types). The idea is that every variant is labeled with a unique identifier. The syntax for variant types includes a way to define new types with type bindings in a **let**, a way to create labeled values of that type, and a way to conditionally branch on the variants using a **case** expression.

A new variant type can be defined using a ***type binding*** in the bindings of a **let** expression. A type binding with m variants has the form:

$$\textbf{type}\ x = l_1\ \textbf{of}\ t_1 \mid l_2\ \textbf{of}\ t_2 \mid \cdots \mid l_m\ \textbf{of}\ t_m$$

where each l_i is a ***label*** and each t_i is the value type associated with that label. If there is no value associated with a label (see below) then the **of** t_i is dropped. A type binding, binds the name of the type as well as the names of the labels. These bindings have the same scope as the function name of a function binding—i.e. they are visible to the expression of the **let** after the **in**, and on the right hand side of all bindings in the **let**.

[2]Technically this is not quite right, but the reasons why are beyond the scope of this book.

Example 1.44. *With variant types we can define a type that is either a Boolean or integer, using, for example,* `bool` *as the label for when it is a Boolean and* `int` *as the label for when it is an integer:*

$$\textbf{type}\ \mathtt{boolOrInt} = \mathtt{bool}\ \textbf{of}\ \mathbb{B} \mid \mathtt{int}\ \textbf{of}\ \mathbb{Z}$$

We can also define a type that has one of multiple labels, none of which is associated with a value:

$$\textbf{type}\ \mathtt{color} = \mathtt{Red} \mid \mathtt{Green} \mid \mathtt{Blue}$$

With variant types the Boolean type $\mathbb{B}$ can be define as

$$\textbf{type}\ \mathbb{B} = \mathtt{true} \mid \mathtt{false}$$

To create a value of the particular variant we simply use the label l. In particular if l **of** t is one of the variants of type t', then for a value of type v, $l(v)$ will generate a value of type t' with label l.

We use the ***case expression*** to branch on the labels of a variant type, where the expression for a type with m variants has the form:

```
case e of
    p1 ⇒ e1
  | p2 ⇒ e2
  | ···
  | pm ⇒ en
```

The **case** expression evaluates e to value v which is then pattern matched with the labels in the patterns p_i. For the first pattern p_i that matches, the e_i is evaluated with the bindings from p_i.

Example 1.45. *The following* **case** *expression will convert a value of* `boolOrInt` *into an integer by treating* `true` *as 1 and* `false` *as 0:*

```
toInt(ib) =
   case ib of
      bool of b ⇒ if b then 0 else 1
    | int of i ⇒ i
```

The variant types we use also allow for the recursive definition of types. This is done by including the name of the type itself in one or more of the variants, as in:

```
type x = ... | l of (..,x,..) | ...
```

This can be used to define a type corresponding to lists of integers:

$$\textbf{type}\ \mathtt{list} = \mathtt{nil} \mid \mathtt{cons}\ \textbf{of}\ \mathbb{Z} \times \mathtt{list}$$

Definition 1.47 (Pidgin ML variant types)**.** *The following definitions extend the definition of expressions e, bindings b and types t.*

$$
\begin{array}{lcll}
b & := & \textbf{type}\ x = [x\ [\textbf{of}\ t]^?\ \underline{|}\ ]^+ & \textit{type binding} \\
e & := & \textbf{case}\ e\ \textbf{of}\ [p \Rightarrow e\ \underline{|}\ ]^+ & \textit{case} \\
t & := & x & \textit{defined types}
\end{array}
$$

The notation $[X]^?$ means that X is optional, and the notation $[X\ \underline{|}\ ]^+$ indicates that X is repeated one or more times, each separated by the $|$ character.

Example 1.46. *The following is a recursive variant type defining binary trees.*

```
type tree = Leaf | Node of tree × int × tree
```

We can then create a tree using, for example

```
Node(Node(Leaf,1,Node(Leaf,2,Leaf)),3,Node(Leaf,4,Leaf))
```

which corresponds to the tree:

```
          3
       /     \
      1        4
    /   \     /  \
  Leaf   2   Leaf Leaf
       /   \
     Leaf   Leaf
```

We can also define a function that counts the number of leaves of a tree recursively as follows:

```
countLeaves(T) =
  case T of
    Leaf ⇒ 1
  | Node of (L,R) ⇒ countLeaves(L) + countLeaves(R)
```

The new syntax for variant types is summarized in Definition 1.47.

Loops, recursion and while. We note that pidgin ML has no explicit syntax for loops. Loops can be implemented with recursion, and in Chapter 5 we will describe `map` and `tabulate`, which can be though of as parallel loops. We however sometimes find the following syntactic sugar convenient for defining what are often referred to as while loops. The ***while loop*** can appear as one of the bindings b in a **let** expression and has the syntax:

```
$xs$ =
    **start** $p$ **and**
    **while** $e_c$ **do**
        $b^+$
```

This is defined to be equivalent to the pair of bindings:

```
$f$ $xs$ = **if** not $e_c$ **then** $xs$
           **else** **let** $b^+$ **in** $f$ $xs$ **end**
$xs$ = $f$ $p$
```

where the xs, p, e_c and b^+ are substituted verbatim. The variables xs are initialized to the pattern p and then the while loop iterates as long as the expression e_c remains true, reevaluating the bindings b^+ on each iteration. The variables are passed from one iteration of the while to the next each of which might redefine them in the bindings. After the while, the variables take on the value they had at the end of the last iteration.

Example 1.48. *The following code sums the squares of the integers from 1 to n.*

```
sumSquares($n$) = **let**
    $(n, s)$ =
        **start** $(n, 0)$ **and**
        **while** $n > 0$ **do**
            $s = s + n^2$
            $n = n - 1$
**in** $s$ **end**
```

The sum s is initially set to 0 and then each iteration adds n^2 to it and decrements n. When n reaches 0 the loop exits with the latest version of s (and n). So sumSquares(3) *evaluates to* $3^2 + 2^2 + 1^2 = 14$. *By definition it is equivalent to:*

```
sumSquares($n$) = **let**
    $f(n, s)$ = **if** not $(n > 0)$ **then** $(n, s)$
                **else** **let**
                        $s = s + n \times n$
                        $n = n - 1$
                     **in** $f(n, s)$ **end**
    $(n, s) = f(n, 0)$
**in** $s$ **end**
```

1.6 Problems

1-1 Repeated Strings

Describe a brute force algorithm for finding the length of the longest repeated string in a string of characters. For example for the string "`axabaxcabaxa`" the longest repeated string is `abax` and has length 4. You can assume you are given a routine `find`(S, w) that returns how many times the string w appears in the string S. Your algorithm should generate about n^2 candidate solutions, where n is the length of the input string.

.