Notes on Functional Programming with Haskell

H. Conrad Cunningham
cunningham@cs.olemiss.edu

Multiparadigm Software Architecture Group
Department of Computer and Information Science
University of Mississippi
201 Weir Hall
University, Mississippi 38677 USA

Fall Semester 2014

H. Conrad Cunningham, D.Sc.
Professor and Chair
Department of Computer and Information Science
University of Mississippi
201 Weir Hall
University, Mississippi 38677
USA

cunningham@cs.olemiss.edu

PREFACE TO 1995 EDITION

I wrote this set of lecture notes for use in the course Functional Programming (CSCI 555) that I teach in the Department of Computer and Information Science at the University of Mississippi. The course is open to advanced undergraduates and beginning graduate students.

The first version of these notes were written as a part of my preparation for the fall semester 1993 offering of the course. This version reflects some restructuring and revision done for the fall 1994 offering of the course—or after completion of the class. For these classes, I used the following resources:

Textbook – Richard Bird and Philip Wadler. *Introduction to Functional Programming,* Prentice Hall International, 1988 [2].

These notes more or less cover the material from chapters 1 through 6 plus selected material from chapters 7 through 9.

Software – Gofer interpreter version 2.30 (2.28 in 1993) written by Mark P. Jones, available via anonymous FTP from directory `pub/haskell/gofer` at the Internet site `nebula.cs.yale.edu`.

Gofer is an interpreter for a dialect of the "lazy" functional programming language Haskell. This interpreter was available on both MS-DOS-based PC-compatibles, 486-based systems executing FreeBSD ("UNIX"), and other UNIX systems.

Manual – Mark P. Jones. *An Introduction to Gofer* (Version 2.20), tutorial manual distributed as a part of the Gofer system [15].

In addition to the Bird and Wadler textbook and the Gofer manual, I used the following sources in the preparation of these lecture notes:

- Paul Hudak and Joseph H. Fasel. "A Gentle Introduction to Haskell", *ACM SIGPLAN NOTICES,* Vol. 27, No. 5, May 1992 [12].
- Paul Hudak, Simon Peyton Jones, and Philip Wadler. "Report on the Programming Language Haskell: A Non-strict, Purely Functional Language", *ACM SIGPLAN NOTICES,* Vol. 27, No. 5, May 1992 [13].
- E. P. Wentworth. *Introduction to Functional Programming using RUFL,* Department of Computer Science, Rhodes University, Grahamstown, South Africa, August 1990 [22].

  This is a good tutorial and manual for the Rhodes University Functional Language (RUFL), a Haskell-like language developed by Wentworth. I used RUFL for two previous offerings of my functional programming course, but switched to

Gofer for the fall semester 1993 offering. My use this source was indirect—via my handwritten lecture notes for the previous versions of the class.

- Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages", *ACM Computing Surveys*, Vol. 21, No. 3, pages 359–411, September 1989 [11].

- Rob Hoogerwoord. *The Design of Functional Programs: A Calculational Approach*, Doctoral Dissertation, Eindhoven Technical University, Eindhoven, The Netherlands, 1989 [10].

- A. J. T. Davie *An Introduction to Functional Programming Systems Using Haskell,* Cambridge University Press, 1992 [7].

- Anthony J. Field and Peter G. Harrison. *Functional Programming,* Addison Wesley, 1988 [8].

  This book uses the "eager" functional language Hope.

- J. Hughes. "Why Functional Programming Matters," *The Computer Journal*, Vol. 32, No. 2, pages 98–107, 1989 [14].

Although the Bird and Wadler textbook is excellent, I decided to supplement the book with these notes for several reasons:

- I wanted to use Gofer/Haskell language concepts, terminology, and example programs in my class presentations and homework exercises. Although close to Haskell, the language in Bird and Wadler differs from Gofer and Haskell somewhat in both syntax and semantics.

- Unlike the stated audience of the Bird and Wadler textbook, my students usually have several years of experience in programming using traditional languages like Pascal, C, or Fortran. This is both an advantage and a disadvantage. On the one hand, they have programming experience and programming language familiarity on which I can build. On the other hand, they have an imperative mindset that sometimes is resistant to the declarative programming approach. I tried to take both into account as I drafted these notes.

- Because of a change in the language used from RUFL to Gofer, I needed to rewrite my lecture notes in 1993 anyway. Thus I decided to invest a bit more effort and make them available in this form. (I expected about 25% more effort, but it probably took about 100% more effort. :-)

- The publisher of the Bird and Wadler textbook told me a few weeks before my 1993 class began that the book would not be available until halfway through the semester. Fortunately, the books arrived much earlier than predicted. In the future, I hope that these notes will give me a "backup" should the book not be available when I need it.

Overall, I was reasonably satisfied with the 1993 draft of the notes. However, I did not achieve all that I wanted. Unfortunately, other obligations did not allow me to substantially address these issues in the current revision. I hope to address the following shortcomings in any future revision of the notes.

- I originally wanted the notes to introduce formal program proof and synthesis concepts earlier and in a more integrated way than these notes currently do. But I did not have sufficient time to reorganize the course and develop the new materials needed. Also the desire to give nontrivial programming exercises led me to focus on the language concepts and features and informal programming techniques during the first half of the course.

- Gofer/Haskell is a relatively large language with many features. In 1993 I spent more time covering the language features than I initially planned to do. In the 1994 class I reordered a few of the topics, but still spent more time on language features. For future classes I need to rethink the choice and ordering of the language features presented. Perhaps a few of the language features should be omitted in an introductory course.

- Still yet there are a few important features that I did not cover. In particular, I did not discuss the more sophisticated features of the type system in any detail (e.g., type classes, instances, and overloading).

- I did not cover all the material that I have in covered in one or both of the previous versions of the course (e.g., cyclic structures, abstract data types, the eight queens problem, and applications of trees).

1997 Note: The 1997 revision is limited to the correction of a few errors. The spring semester 1997 class is using the new Hugs interpreter rather than Gofer and the textbook *Haskell: The Craft of Functional Programming* by Simon Thompson (Addison-Wesley, 1996).

2014 Note: The 2014 revision seeks primarily to update these Notes to use Haskell 2010 and the Haskell Platform (i.e., GHC and GHCi). The focus is on chapters 3, 5, 6, 7, 8, and 10, which are being used in teaching a Haskell-based functional programming module in CSci 450 (Organization of Programming Languages).

Acknowledgements

I thank the many students in the CSCI 555 classes who helped me find many typographical and presentation errors in the working drafts of these notes. I also thank those individuals at other institutions who have examined these notes and suggested improvements.

I thank Diana Cunningham, my wife, for being patient with all the late nights of work that writing these notes required.

The preparation of this document was supported by the National Science Foundation under Grant CCR-9210342 and by the Department of Computer and Information Science at the University of Mississippi.

Contents

1 INTRODUCTION

1.1 Course Overview

This is a course on functional programming.

As a course on *programming*, it emphasizes the analysis and solution of problems, the development of correct and efficient algorithms and data structures that embody the solutions, and the expression of the algorithms and data structures in a form suitable for processing by a computer. The focus is more on the human thought processes than on the computer execution processes.

As a course on *functional* programming, it approaches programming as the construction of definitions for (mathematical) functions and data structures. Functional programs consist of *expressions* that use these definitions. The execution of a functional program entails the evaluation of the expressions making up the program. Thus the course's focus is on problem solving techniques, algorithms, data structures, and programming notations appropriate for the functional approach.

This is not a course on functional programming *languages*. In particular, the course does not undertake an in-depth study of the techniques for implementing functional languages on computers. The focus is on the concepts for programming, not on the internal details of the technological artifact that executes the programs.

Of course, we want to be able to execute our functional programs on a computer and, moreover, to execute them efficiently. Thus we must become familiar with some concrete programming language and use an implementation of that language to execute our programs. To be able to analyze program efficiency, we must also become familiar with the basic techniques that are used to evaluate expressions. To be specific, this class will use a functional programming environment called GHC (Glasgow Haskell Compiler). GHC is distributed in a "batteries included" bundle called the the Haskell Platform . (That is, it bundles GHC with commonly used libraries and tools.) The language accepted by GHC is the "lazy" functional programming language Haskell 2010. A program processed by GHC evaluates expressions according to an execution model called *graph reduction*.

Being "practical" is not an overriding concern of this course. Although functional languages are increasing in importance, their use has not yet spread much beyond the academic and industrial research laboratories. While a student may take a course on C++ programming and then go out into industry and find a job in which the C++ knowledge and skills can be directly applied, this is not likely to occur with a course on functional programming.

However, the fact that functional languages are not broadly used does not mean that this course is impractical. A few industrial applications are being developed using various functional languages. Many of the techniques of functional programming

can also be applied in more traditional programming and scripting languages. More importantly, any time programmers learn new approaches to problem solving and programming, they become better programmers. A course on functional programming provides a novel, interesting, and, probably at times, frustrating opportunity to learn more about the nature of the programming task. Enjoy the semester!

1.2 Excerpts from Backus' 1977 Turing Award Address

This subsection contains excerpts from computing pioneer John Backus' 1977 ACM Turing Award Lecture published as article "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs [1]" (*Communications of the ACM*, Vol. 21, No. 8, pages 613–41, August 1978). Although functional languages like Lisp go back to the late 1950's, Backus's address did much to stimulate research community's interest in functional programming languages and functional programming.

Programming languages appear to be in trouble. Each successive language incorporates, with little cleaning up, all the features of its predecessors plus a few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. ... Each new language claims new and fashionable features, such as strong typing or structured control statements, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.

Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But there is a desperate need for a powerful methodology to help us think about programs, and no conventional language even begins to meet that need. In fact, conventional languages create unnecessary confusion in the way we think about programs. ...

In order to understand the problems of conventional programming languages, we must first examine their intellectual parent, the von Neumann computer. What is a von Neumann computer? When von Neumann and others conceived of it ... [in the 1940's], it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this ... concept.

In its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the

contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent form the store, then its address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must either be generated by a fixed rule (e.g., "add 1 to the program counter") or by an instruction that was sent through the tube, in which case its address must have been sent, and so on.

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. ...

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our ... old belief that there is only one kind of computer is the basis our our belief that there is only one kind of programming language, the conventional—von Neumann—language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem. [Note: Backus was one of the designers of Fortran and of Algol-60.]

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-at-time terms in much the same way the computer's bottleneck does.

Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements in the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control.

Our fixation on von Neumann languages has continued the primacy of the von Neumann computer, and our dependency on *it* has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures. ...

Note: In his Turing Award Address, Backus went on to describe FP, his proposal for a functional programming language. He argued that languages like FP would allow programmers to break out of the von Neumann bottleneck and find new ways of thinking about programming. Although languages like Lisp had been in existence since the late 1950's, the widespread attention given to Backus' address and paper stimulated new interest in functional programming to develop by researchers around the world.

Aside: Above Backus states that "the world of statements is a disorderly one, with few mathematical properties". Even in 1977 this was a bit overstated since Dijkstra's work on the weakest precondition calculus and other work on axiomatic semantics had already appeared. However, because of the referential transparency (discussed later) property of purely functional languages, reasoning can often be done in an equational manner within the context of the language itself. In contrast, the wp-calculus and other axiomatic semantic approaches must project the problem from the world of programming language statements into the world of predicate calculus, which is much more orderly.

1.3 Programming Language Paradigms

Reference: The next two subsections are based, in part, on Hudak's article "Conception, Evolution, and Application of Functional Programming Languages [13]" (*ACM Computing Surveys,* Vol. 21, No. 3, pages 359–411, September 1989).

Programming languages are often classified according to one of two different paradigms: imperative and declarative.

Imperative languages

A program in an imperative language has an *implicit state* (i.e., values of variables, program counters, etc.) that is modified (i.e., side-effected) by *constructs* (i.e., commands) in the source language.

As a result, such languages generally have an explicit notion of *sequencing* (of the commands) to permit precise and deterministic control of the state changes.

Imperative programs thus express *how* something is to be computed.

These are the "conventional" or "von Neumann languages" discussed by Backus. They are well suited to traditional computer architectures.

Most of the languages in existence today are in this category: Fortran, Algol, Cobol, Pascal, Ada, C, C++, Java, etc.

Declarative languages

A program in a declarative language has *no implicit* state. Any needed state information must be handled explicitly.

A program is made up of *expressions* (or terms) rather than commands.

Repetitive execution is accomplished by *recursion* rather than by sequencing.

Declarative programs express *what* is to be computed (rather than how it is to be computed).

Declarative programs are often divided into two types:

Functional (or applicative) languages

The underlying model of computation is the mathematical concept of a *function.*

In a computation a function is applied to zero or more arguments to compute a single result, i.e., the result is deterministic (or predictable).

Purely functional:	FP, Haskell, Miranda, Hope, Orwell
Hybrid languages:	Lisp, Scheme, SML
	(Scheme & SML have powerful declarative subsets)
Dataflow languages:	Id, Sisal

Relational (or logic) languages

The underlying model of computation is the mathematical concept of a *relation* (or a *predicate*).

A computation is the (nondeterministic) association of a group of values—with backtracking to resolve additional values.

Examples: Prolog (pure), Parlog, KL1

Note: Most Prolog implementations have imperative features such as the cut and the ability to assert and retract clauses.

1.4 Reasons for Studying Functional Programming

1. **Functional programs are easier to manipulate mathematically than imperative programs.**

   The primary reason for this is the property of *referential transparency*, probably the most important property of modern functional programming languages.

   Referential transparency means that, within some well-defined context, a variable (or other symbol) *always* represents the *same value*. Since a variable always has the same value, we can replace the variable in an expression by its value or vice versa. Similarly, if two subexpressions have equal values, we can replace one subexpression by the other. That is, "equals can be replaced by equals".

   Functional programming languages thus use the same concept of a variable that mathematics uses.

   On the other hand, in most imperative languages a variable represents an address or "container" in which values may be stored; a program may change the value stored in a variable by executing an assignment statement.

   Because of referential transparency, we can construct, reason about, and manipulate functional programs in much the same way we can any other mathematical expressions [2, 3]. Many of the familiar "laws" from high school algebra still hold; new "laws" can be defined and proved for less familiar primitives and even user-defined operators. This enables a relatively natural equational style of reasoning.

   For example, we may want to prove that a program meets its specification or that two programs are equivalent (in the sense that both yield the same "outputs" given the same "inputs").

   We can also construct and prove algebraic "laws" for functional programming. For example, we might prove that some operation (i.e., two-argument function) is commutative or associative or perhaps that one operation distributes over another.

   Such algebraic laws enable one program to be transformed into another equivalent program either by hand or by machine. For example, we might use the

laws to transform one program into an equivalent program that can be executed more efficiently.

2. **Functional programming languages have powerful abstraction mechanisms.**

   Speaking operationally, a function is an abstraction of a pattern of behavior.

   For example, if we recognize that a C or Pascal program needs to repeat the *same* operations for each member of a set of similar data structures, then we usually encapsulate the operations in a function or procedure. The function or procedure is an abstraction of the application of the operation to data structures of the given type.

   Now suppose instead that we recognize that our program needs to perform *similar*, but different, operations for each member of a set of similar data structures. Can we create an abstraction of the application of the similar operations to data structures of the given type?

   For instance, suppose we want to compute either the sum or the product of the elements of an array of integers. Addition and multiplication are similar operations; they are both associative binary arithmetic operations with identity elements.

   Clearly, C or Pascal programs implementing sums and products can go through the same pattern of operations on the array: initialize a variable to the identity element and then loop through the array adding or multiplying each element by the result to that point. Instead of having separate functions for each operation, why not just have one function and supply the operation as an argument?

   A function that can take functions as arguments or return functions as results is called a *higher-order function*. Most imperative languages do not fully support higher-order functions.

   However, in most functional programming languages functions are treated as *first class* values. That is, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions.

   Typically, functions in imperative languages are not treated as first-class values.

   The higher-order functions in functional programming languages enable very regular and powerful abstractions and operations to be constructed. By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

   A programmer needs to write fewer "lines of code" in a concise programming notation than in a verbose one. Thus the programmer should be able to complete the task in less time. Since, in general, a short program is easier to comprehend than a long one, a programmer is less likely to make an error in a short program than in a long one. Consequently, functional programming can lead to both increased programmer productivity and increased program reliability.

Caveat: Excessive concern for conciseness can lead to cryptic, difficult to understand programs and, hence, low productivity and reliability. Conciseness should not be an end in itself. The understandability and correctness of a program are more important goals.

Higher-order functions also increase the *modularity* of programs by enabling simple program fragments to be "glued together" readily into more complex programs [14].

3. **Functional programming enables new algorithmic approaches.**

   This is especially true for languages (like Haskell) that use what is called *lazy evaluation*.

   In a lazy evaluation scheme, the evaluation of an expression is deferred until the value of the expression is actually needed elsewhere in the computation. That is, the expression is evaluated on demand. This contrasts with what is called *eager evaluation* in which an expression is evaluated as soon as its inputs are available.

   For example, if eager evaluation is used, an argument (which may be an arbitrary expression) of a function call is evaluated before the body of the function. If lazy evaluation is used, the argument is not evaluated until the value is actually needed during the evaluation of the function body. If an argument's value is never needed, then the argument is expression is never evaluated.

   Why should we care? Well, this facility allows programmers to construct and use data structures that are conceptually unbounded or infinite in size. As long as a program never actually needs to inspect the entire structure, then a terminating computation is still possible.

   For example, we might define the list of natural numbers as a list beginning with 0, followed by the list formed by adding one to each element of the list of natural numbers.

   Lazy evaluation thus allows programmers to separate the data from the control. They can define a data structure without having to worry about how it is processed and they can define functions that manipulate the data structure without having to worry about its size or how it is created. This ability to separate the data from the control of processing enables programs to be highly modular [14].

   For example, we can define the list of even naturals by applying a function that filters out odd integers to the infinite list of naturals defined above. This definition has no operational control within it and can thus be combined with other functions in a modular way.

4. **Functional programming enables new approaches to program development.**

   As we discussed above, it is generally easier to reason about functional programs than imperative programs. It is possible to prove algebraic "laws" of functional programs that give the relationships among various operators in the language. We can use these laws to transform one program to another equivalent one.

   These mathematical properties also open up new ways to write programs.

   Suppose we want a program to break up a string of text characters into lines. Section 4.3 of the Bird and Wadler textbook [2] and Section 12.6 of these notes shows a novel way to construct this program.

   First, Bird and Wadler construct a program to do the *opposite* of what we want—to combine lines into a string of text. This function is very easy to write.

   Next, taking advantage of the fact that this function is the inverse of the desired function, they use the "laws" to manipulate this simple program to find its inverse. The result is the program we want!

5. **Functional programming languages encourage (massively) parallel execution.**

   To exploit a parallel processor, it must be possible to decompose a program into components that can be executed in parallel, assign these components to processors, coordinate their execution by communicating data as needed among the processors, and reassemble the results of the computation.

   Compared to traditional imperative programming languages, it is quite easy to execute components of a functional program in parallel [19]. Because of the referential transparency property and the lack of sequencing, there are no time dependencies in the evaluation of expressions; the final value is the same regardless of which expression is evaluated first. The nesting of expressions within other expressions defines the data communication that must occur during execution.

   Thus executing a functional program in parallel does not require the availability of a highly sophisticated compiler for the language.

   However, a more sophisticated compiler can take advantage of the algebraic laws of the language to transform a program to an equivalent program that can more efficiently be executed in parallel.

   In addition, frequently used operations in the functional programming library can be be optimized for highly efficient parallel execution.

   Of course, compilers can also be used to decompose traditional imperative languages for parallel execution. But it is not easy to find all the potential parallelism. A "smart" compiler must be used to identify unnecessary sequencing and find a safe way to remove it.

In addition to the traditional imperative programming languages, imperative languages have also been developed especially for execution on a parallel computer. These languages shift some of the work of decomposition, coordination, and communication to the programmer.

A potential advantage of functional languages over parallel imperative languages is that the functional programmer does not, in general, need to be concerned with the specification and control of the parallelism.

In fact, functional languages probably have the problem of too much potential parallelism. It is easy to figure out what can be executed in parallel, but it is sometimes difficult to determine what components should actually be executed in parallel and how to allocate them to the available processors. Functional languages may be better suited to the massively parallel processors of the future than most present day parallel machines.

6. **Functional programming is important in some application areas of computer science.**

   The artificial intelligence (AI) research community has used languages such as Lisp and Scheme since the 1960's. Some AI applications have been commercialized during the past two decades.

   Also a number of the specification, modeling, and rapid-prototyping languages that are appearing in the software engineering community have features that are similar to functional languages.

7. **Functional programming is related to computing science theory.**

   The study of functional programming and functional programming languages provides a good opportunity to learn concepts related to programming language semantics, type systems, complexity theory, and other issues of importance in the theory of computing science.

8. **Functional programming is an interesting and mind-expanding activity for students of computer science!?**

   Functional programming requires the student to develop a different perspective on programming.

1.5 Objections Raised Against Functional Programming

1. **Functional programming languages are inefficient toys!**

   This was definitely true in the early days of functional programming. Functional languages tended to execute slowly, require large amounts of memory, and have limited capabilities.

   However, research on implementation techniques has resulted in more efficient and powerful implementations today.

   Although functional language implementations will probably continue to increase in efficiency, they likely will never become as efficient as the implementations of imperative "von Neumann" languages are on traditional "von Neumann" architectures.

   However, new computer architectures may allow functional programs to execute competitively with the imperative languages on today's architectures. For example, computers based on the dataflow and graph reduction models of computation are more suited to execute functional languages than imperative languages.

   Also the ready availability of parallel computers may make functional languages more competitive because they more readily support parallelism than traditional imperative languages.

   Moreover, processor time and memory usage just aren't as important concerns as they once were. Both fast processors and large memories have become relatively inexpensive and readily available. Now it is common to dedicate one or more processors and several megabytes of memory to individual users of workstations and personal computers.

   As a result, the community can now afford to dedicate considerable computer resources to improving programmer productivity and program reliability; these are issues that functional programming may address better than imperative languages.

2. **Functional programming languages are not (and cannot be) used in the real world!**

   It is still true that functional programming languages are not used very widely in industry. But, as we have argued above, the functional style is becoming more important—especially as commercial AI applications have begun to appear.

   If new architectures like the dataflow machines emerge into the marketplace, functional programming languages will become more important.

   Although the functional programming community has solved many of the difficulties in implementation and use of functional languages, more research is needed on several issues of importance to the real world: on facilities for input/output, nondeterministic, realtime, parallel, and database programming.

More research is also needed in the development of algorithms for the functional paradigm. The functional programming community has developed functional versions of many algorithms that are as efficient, in terms of big-O complexity, as the imperative versions. But there are a few algorithms for which efficient functional versions have not yet been found.

3. **Functional programming is awkward and unnatural!**

   Maybe. It might be the case that functional programming somehow runs counter to the way that normal human minds work—that only mental deviants can ever become effective functional programmers. Of course, some people might say that about programming and programmers in general.

   However, it seems more likely that the awkwardness arises from the lack of education and experience. If we spend many years studying and doing programming in the imperative style, then any significantly different approach will seem unnatural.

   Let's give the functional approach a fair chance.

2 FUNCTIONS AND THEIR DEFINITIONS

2.1 Mathematical Concepts and Terminology

In mathematics, a *function* is a mapping from a set A into a set B such that each element of A is mapped into a unique element of B. The set A (on which f is defined) is called the *domain* of f. The set of all elements of B mapped to elements of A by f is called the *range* (or *codomain*) of f, and is denoted by $f(A)$.

If f is a function from A into B, then we write:

$$f : A \to B$$

We also write the equation $f(a) = b$ to mean that the *value* (or *result*) from applying function f to an element $a \in A$ is an element $b \in B$.

A function $f : A \to B$ is *one-to-one* (or *injective*) if and only if distinct elements of A are mapped to distinct elements of B. That is, $f(a) = f(a')$ if and only if $a = a'$.

A function $f : A \to B$ is *onto* (or *surjective*) if and only if, for every element $b \in B$, there is some element $a \in A$ such that $f(a) = b$.

A function $f : A \to B$ is a *one-to-one correspondence* (or *bijection*) if and only if f is one-to-one and onto.

Given functions $f : A \to B$ and $g : B \to C$, the *composition* of f and g, written $g \circ f$, is a function from A into C such that

$$(g \circ f)(a) = g(f(a)).$$

A function $f^{-1} : B \to A$ is an *inverse* of $f : A \to B$ if and only if, for every $a \in A$, $f^{-1}(f(a)) = a$.

An inverse exists for any one-to-one function.

If function $f : A \to B$ is a one-to-one correspondence, then there exists an inverse function $f^{-1} : B \to A$ such that, for every $a \in A$, $f^{-1}(f(a)) = a$ and that, for every $b \in B$, $f(f^{-1}(b)) = b$. Hence, functions that are one-to-one correspondences are also said to be *invertible*.

If a function $f : A \to B$ and $A \subseteq A'$, then we say that f is a *partial function* from A' to B and a *total function* from A to B. That is, there are some elements of A' on which f may be undefined.

A function $\oplus : (A \times A) \to A$ is called a *binary operation on A*. We usually write binary operations in infix form: $a \oplus a'$. (In computing science, we often call a function $\oplus : (A \times B) \to C$ a binary operation as well.)

Let $\oplus$ be a binary operation on some set A and x, y, and z be elements of A.

- Operation $\oplus$ is *associative* if and only if $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for any x, y, and z.
- Operation $\oplus$ is *commutative* (also called *symmetric*) if and only if $x \oplus y = y \oplus x$ for any x and y.
- An element e of set A is a *left identity* of $\oplus$ if and only if $e \oplus x = x$ for any x, a *right identity* if and only if $x \oplus e = x$, and an *identity* if and only if it is both a left and a right identity. An identity of an operation is sometimes called a *unit* of the operation.
- An element z of set A is a *left zero* of $\oplus$ if and only if $z \oplus x = z$ for any x, a *right zero* if and only if $x \oplus z = z$, and a *zero* if and only if it is both a right and a left zero.
- If e is the identity of $\oplus$ and $x \oplus y = e$ for some x and y, then x is a *left inverse* of y and y is a *right inverse* of x. Elements x and y are inverses of each other if $x \oplus y = e = y \oplus x$.
- If $\oplus$ is an *associative* operation, then $\oplus$ and A are said to form a *semigroup*.
- A semigroup that also has an *identity* element is called a *monoid*.
- If every element of a monoid has an inverse then the monoid is called a *group*.
- If a monoid or group is also commutative, then it is said to be *Abelian*.

2.2 Function Definitions

Note: Mathematicians usually refer to the positive integers as the *natural numbers*. Computing scientists usually include 0 in the set of natural numbers.

Consider the factorial function *fact*. This function can be defined in several ways. For any natural number, we might define *fact* with the equation

$$fact(n) = 1 \times 2 \times 3 \times \cdots \times n$$

or, more formally, using the product operator as

$$fact(n) = \prod_{i=1}^{i=n} i$$

or, in the notation that the instructor prefers, as

$$fact(n) = (\Pi\, i : 1 \leq i \leq n : i).$$

We note that $fact(0) = 1$, the *identity* element of the multiplication operation.

We can also define the factorial function with a *recursive* definition (or *recurrence relation*) as follows:

$$fact'(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times fact'(n-1), & \text{if } n \geq 1 \end{cases}$$

It is, of course, easy to see that the recurrence relation definition is equivalent to the previous definitions. But how can we prove it?

To prove that the above definitions of the factorial function are equivalent, we can use *mathematical induction* over the natural numbers.

2.3 Mathematical Induction over Natural Numbers

To prove a proposition $P(n)$ holds for any natural number n, one must show two things:

Base case $n = 0$**.** That $P(0)$ holds.

Inductive case $n = m+1$**.** That, if $P(m)$ holds for some natural number m, then $P(m+1)$ also holds. (The $P(m)$ assumption is called the *induction hypothesis*.)

Now let's prove that the two definitions $fact$ and $fact'$ are equivalent, that is, for all natural numbers n,

$$fact(n) = fact'(n).$$

Base case $n = 0$.

$$\begin{array}{ll} & fact(0) \\ = & \quad \{ \text{ definition of } fact \text{ (left to right) } \} \\ & (\Pi\, i : 1 \leq i \leq 0 : i) \\ = & \quad \{ \text{ empty range for } \Pi,\ 1 \text{ is the identity element of } \times \ \} \\ & 1 \\ = & \quad \{ \text{ definition of } fact' \text{ (first leg, right to left) } \} \\ & fact'(0) \end{array}$$

Inductive case $n = m{+}1$.

Given $fact(m) = fact'(m)$, prove $fact(m{+}1) = fact'(m{+}1)$.

$$\begin{array}{ll} & fact(m{+}1) \\ = & \quad \{ \text{ definition of } fact \text{ (left to right) } \} \\ & (\Pi\, i : 1 \leq i \leq m{+}1 : i) \\ = & \quad \{ \ m{+}1 > 0, \text{ so } m{+}1 \text{ term exists, split it out } \} \\ & (m{+}1) \times (\Pi\, i : 1 \leq i \leq m : i) \\ = & \quad \{ \text{ definition of } fact \text{ (right to left) } \} \\ & (m{+}1) \times fact(m) \\ = & \quad \{ \text{ induction hypothesis } \} \\ & (m{+}1) \times fact'(m) \\ = & \quad \{ \ m{+}1 > 0, \text{ definition of } fact' \text{ (second leg, right to left) } \} \\ & fact'(m{+}1) \end{array}$$

Therefore, we have proved $fact(n) = fact'(n)$ for all natural numbers n. QED

Note the equational style of reasoning we used. We proved that one expression was equal to another by beginning with one of the expressions and repeatedly "substituting equals for equals" until we got the other expression.

Each transformational step was justified by a definition, a known property of arithmetic, or the induction hypothesis.

Note that the structure of the inductive argument closely matches the structure of the recursive definition of $fact'$.

What does this have to do with functional programming? Many of the functions we will define in this course have a recursive structure similar to $fact'$. The proofs and program derivations that we do will resemble the inductive argument above.

Recursion, induction, and iteration are all manifestations of the same phenomenon.

3 FIRST LOOK AT HASKELL

Now let's look at our first function definition in the Haskell language, a program to implement the factorial function for natural numbers. (For the purposes of this course, remember that the natural numbers consist of 0 and the positive integers.)

In Section 2.2, we saw two definitions, *fact* and *fact′*, that are equivalent for all natural number arguments. We defined *fact* using the product operator as follows:

$$fact(n) = \prod_{i=1}^{i=n} i \ .$$

(We note that $fact(0) = 1$, which is the *identity* element of the multiplication operation.)

We also defined the factorial function with a *recursive* definition (or *recurrence relation*) as follows:

$$fact'(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times fact'(n-1), & \text{if } n \geq 1 \end{cases}$$

Since the domain of *fact′* is the set of natural numbers, a set over which induction is defined, we can easily see that this recursive definition is well defined. For $n = 0$, the base case, the value is simply 1. For $n \geq 1$, the value of $fact'(n)$ is recursively defined in terms of $fact'(n-1)$; the argument of the recursive application decreases toward the base case.

One way to translate the recursive definition *fact′* into Haskell is the following:

```
fact1 :: Int -> Int
fact1 n = if n == 0 then
              1
          else
              n * fact1 (n-1)
```

- The first line above is the *type signature* for function `fact1`. In general, type signatures have the syntax *object* :: *type*.

  Here object `fact1` is defined as a function (denoted by the "`->`" symbol) that takes one argument of type integer (denoted by the first `Int`) and returns a value of type integer (denoted by the last `Int`).

  Haskell does not have a built-in natural number type. Thus we choose type `Int` for the argument and result of `fact1`. (`Int` is a bounded integer type guaranteed to have at least the range $[-2^{29}, 2^{29}-1]$. Haskell also has the unbounded integer time `Integer`.

- The declaration for the function `fact1` begins on the second line. Note that it is an equation of the form *fname parms* = *body* where *fname* is the name of the function, *parms* are the parameters for the function, and *body* is an expression defining the function's result.

  A function may have zero or more parameters. The parameters are listed after the function name without being enclosed in parentheses and without commas separating them.

  The parameter identifiers may appear in the *body* of the function. In the evaluation of a function *application* the actual argument values are substituted for parameters in the *body*.

- Note that the function `fact1` is defined to be an `if-then-else` *expression*. Evaluation of the `if-then-else` yields the value `1` if argument `n` has the value `0` (i.e., `n == 0`) and the value `n * (fact1 (n-1))` otherwise.

- The `else` clause includes a recursive application of `fact1`. The expression `(n-1)` is the argument for the recursive application.

  Note that the value of the argument for the recursive application is less than the value of the original argument. For each recursive application of `fact` to a natural number, the argument's value moves closer to the termination value `0`.

- Unlike most conventional languages, the *indentation is significant* in Haskell. The indentation indicates the nesting of expressions.

- This Haskell function does not match the mathematical definition given above. What is the difference?

  Notice the domains of the functions. The evaluation of `fact1` will go into an "infinite loop" and eventually abort when it is applied to a negative value.

In Haskell there is *only* one way to form more complex expressions from simpler ones: *apply* a function.

Neither parentheses nor special operator symbols are used to denote function application; it is denoted by simply listing the argument expressions following the function name, for example:

```
f x y
```

However, the usual *prefix* form for a function application is not a convenient or natural way to write many common expressions. Haskell provides a helpful bit of syntactic sugar, the *infix* expression. Thus instead of having to write the addition of `x` and `y` as

```
add x y
```

we can write it as

```
x + y
```

as we have since elementary school.

Function application (i.e., juxtaposition of function names and argument expressions) has higher precedence than other operators. Thus the expression `f x + y` is the same as `(f x) + y`.

An alternative way to differentiate the two cases in the recursive definition is to use a different equation for each case. If the Boolean *guard* (e.g., `n == 0`) for an equation evaluates to true, then that equation is used in the evaluation of the function.

```
fact2 :: Int -> Int
fact2 n
  | n == 0    = 1
  | otherwise = n * fact2 (n-1)
```

Function `fact2` is equivalent to the `fact1`. The guards are evaluated in a top-to-bottom order. The `otherwise` guard succeeds if the `n == 0` guard fails; thus it is similar to the trailing `else` clause on the `if-then-else` expression used in `fact1`.

Another equivalent way to differentiate the two cases in the recursive definition is to use *pattern matching* as follows:

```
fact3 :: Int -> Int
fact3 0 = 1
fact3 n = n * fact3 (n-1)
```

The parameter pattern `0` in the first leg of the definition only matches arguments with value 0. Since Haskell checks patterns and guards in a top-to-bottom order, the `n` pattern matches all nonzero values. Thus `fact1`, `fact2`, and `fact3` are equivalent.

To stop evaluation from going into an "infinite loop" for negative arguments, we can remove the negative integers from the function's domain. One way to do this is by using guards to narrow the domain to the natural numbers as in the definition of `fact4` below:

```
fact4 :: Int -> Int
fact4 n
    | n == 0 =  1
    | n >= 1 =  n * fact4 (n-1)
```

Function `fact4` is undefined for negative arguments. If `fact4` is applied to a negative argument, the evaluation of the program encounters an error quickly and returns without going into an infinite loop.

In versions of Haskell before the 2010 standard, a perhaps more elegant way to narrow the domain is by using Haskell's special natural number patterns of the form `(n+k)` as shown below:

```
fact5 :: Int -> Int -- not valid in Haskell 2010
fact5 0     = 1
fact5 (n+1) = (n+1) * fact5 n
```

As before, the pattern `0` matches an argument with value 0. But the special pattern `(n+1)` only matches argument values that are *at least* 1; variable `n` is bound to the value that is one less than the argument value.

If `fact5` is applied to a negative argument, the evaluation of the program encounters an error immediately and returns without going into an infinite loop.

But, because the ***`n+k`*** *(e.g.,* ***`n+1`****) style of patterns introduced inconsistencies into the pattern matching feature, this style was removed from the 2010 Haskell standard.*

The five definitions we have looked at so far use recursive patterns similar to the recurrence relation $fact'$. Another alternative is to use the library function `product` and the list-generating expression `[1..n]` to define a solution that is like the function *fact*:

```
fact6 :: Int -> Int
fact6 n = product [1..n]
```

The list expression `[1..n]` generates a *list* of consecutive integers beginning with `1` and ending with `n`. The library function `product` computes the product of the elements of this finite list.

If `fact6` is applied to a negative argument, it will return the value 1. This is consistent with the function *fact* upon which it was based.

Which of the above six definitions for the factorial function is better?

Most people in the functional programming community would consider `fact5` and `fact6` as being better than the others. The choice between them depends upon whether one wants to trap the application to negative numbers as an error or to return the value 1.

4 USING THE INTERPRETER

This section from the Gofer/Hugs Notes was obsolete. The course now uses the Glasgow Haskell Compiler (GHC) and its interactive interface GHCi. The author removed the text but left the section as a placeholder for a future revision.

5 HASKELL BASICS

5.1 Built-in Types

The type system is an important part of Haskell; the compiler or interpreter uses the type information to detect errors in expressions and function definitions. To each expression Haskell assigns a *type* that describes the kind of value represented by the expression.

Haskell has both built-in types and facilities for defining new types. In the following we discuss the built-in types. Note that a Haskell type name begins with a capital letter.

Integers: `Int` and `Integer`

The `Int` data type is usually the integer data type supported directly by the host processor (e.g., 32- or 64-bits on most current processors), but it is guaranteed to have the range of at least a 30-bit, two's complement integer. The type `Integer` is an unbounded precision integer type. Haskell supports the usual integer literals (i.e., constants) and operations.

Floating point numbers: `Float` and `Double`

The `Float` and `Double` data types are the single and double precision floating point numbers on the host processor. Haskell floating point literals must include a decimal point; they may be signed or in scientific notation: `3.14159`, `2.0`, `-2.0`, `1.0e4`, `5.0e-2`, `-5.0e-2`.

Booleans: `Bool`

Boolean literals are `True` and `False` (note capitals). Haskell supports Boolean operations such as `&&` (and), `||` (or), and `not`.

Characters: `Char`

Haskell uses Unicode for its character data type. Haskell supports character literals enclosed in single quotes—including both the graphic characters (e.g., `'a'`, `'0'`, and `'Z'`) and special codes entered following the escape character backslash "\" (e.g., `'\n'` for newline, `'\t'` for horizontal tab, and `'\\'` for backslash itself).

In addition, a backslash character \ followed by a number generates the corresponding Unicode character code. If the first character following the backslash is `o`, then the number is in octal representation; if followed by `x`, then in hexadecimal notation; and otherwise in decimal notation.

For example, the exclamation point character can be represented in any of the following ways: `'!'`, `'\33'`, `'\o41'`, `'\x21'`

Functions: `t1 -> t2`

If `t1` and `t2` are types then `t1 -> t2` is the type of a function that takes an argument of type `t1` and returns a result of type `t2`. Function and variable names begin with lowercase letters optionally followed by a sequences of characters each of which is a letter, a digit, an apostrophe (′) (sometimes pronounced "prime"), or an underscore (_).

Haskell functions are first-class objects. They can be arguments or results of other functions or be components of data structures. Multi-argument functions are curried– that is, treated as if they take their arguments one at a time.

For example, consider the integer addition operation `(+)`. In mathematics, we normally consider addition as an operation that takes a *pair* of integers and yields an integer result, which would have the type signature

```
(+) :: (Int,Int) -> Int
```

In Haskell, we give the addition operation the type

```
(+) :: Int -> (Int -> Int)
```

or just

```
(+) :: Int -> Int -> Int
```

since `->` binds from the right.

Thus `(+)` is a one argument function that takes some `Int` argument and returns a function of type `Int -> Int`. Hence, the expression `((+) 5)` denotes a function that takes one argument and returns that argument plus `5`.

We sometimes speak of this `(+)` operation as being *partially applied* (i.e., to one argument instead of two).

This process of replacing a structured argument by a sequence of simpler ones is called *currying*, named after American logician Haskell B. Curry who first described it.

The Haskell library, called the *standard prelude*, contains a wide range of predefined functions including the usual arithmetic, relational, and Boolean operations. Some of these operations are predefined as *infix* operations.

Lists: `[t]`

The primary built-in data structure in Haskell is the *list*, a sequence of values. All the elements in a list must have the same type. Thus we declare lists with notation such as `[t]` to denote a list of zero or more elements of type `t`.

A *list* is is hierarchical data structure. It is either *empty* or it is a pair consisting of a *head element* and a *tail* that is itself a list of elements.

Empty square brackets (`[]`), pronounced "nil", represent the empty list.

A colon (`:`), pronounced "cons", represents the *list constructor* operation between a head element on the left and a tail list on the right.

For example, `[]`, `2:[]`, and `3:(2:[])` denote lists.

Haskell adds a bit of syntactic sugar to make expressing lists easier. The cons operations binds from the right. Thus `5:(3:(2:[]))` can be written `5:3:2:[]`. As a further convenience, a list can be written as a comma-separated sequence enclosed in brackets, e.g., `5:3:2:[]` as `[5,3,2]`.

Haskell supports two list selector functions, `head` and `tail`, such that:

`head (h:t)` $\Longrightarrow$ `h`, where `h` is the head element of list,
`tail (h:t)` $\Longrightarrow$ `t`, where `t` is the tail list.

Aside: Instead of `head`, Lisp uses `car` and other languages use `hd`, `first`, etc. Instead of `tail`, Lisp uses `cdr` and other languages use `tl`, `rest`, etc.

The Prelude supports a number of other useful functions on lists. For example, `length` takes a list and returns its length.

Note that lists are defined inductively. That is, they are defined in terms of a base element `[]` and the list constructor operation cons (`:`). As you would expect, a form of mathematical induction can be used to prove that list-manipulating functions satisfy various properties. We will discuss in Section 11.1.

Strings: `String`

In Haskell, a *string* is treated as a list of characters.. Thus the data type `String` is defined with a *type synonym* as follows:

```
type String = [Char]
```

In addition to the standard list syntax, a `String` literal can be given by a sequence of characters enclosed in double quotes. For example, `"oxford"` is shorthand for `['o','x','f','o','r','d']`.

Strings can contain any graphic character or any special character given as escape code sequence (using backslash). The special escape code `\&` is used to separate any character sequences that are otherwise ambiguous.

Example: `"Hello\nworld!\n"` is a string that has two newline characters embedded.

Example: `"\12\&3"` represents the list `['\12','3']`.

Because strings are represented as lists, all of the Prelude functions for manipulating lists also apply to strings.

```
head "oxford" ⟹ 'o'
tail "oxford" ⟹ "xford"
```

Consider a function to compute the length of a string:

```
len :: String -> Int
len s  =  if s == [] then 0 else 1 + len (tail s)
```

Simulation (this is not necessarily the series of steps actually taken during execution of the Haskell program):

```
len "five"
   ⟹ 1 + len (tail "five")
   ⟹ 1 + len "ive"
   ⟹ 1 + (1 + len (tail "ive"))
   ⟹ 1 + (1 + len "ve")
   ⟹ 1 + (1 + (1 + len (tail "ve")))
   ⟹ 1 + (1 + (1 + len "e"))
   ⟹ 1 + (1 + (1 + (1 + len (tail "e"))))
   ⟹ 1 + (1 + (1 + (1 + len "")))
   ⟹ 1 + (1 + (1 + (1 + 0)))
   ⟹ 1 + (1 + (1 + 1))
   ⟹ 1 + (1 + 2)
   ⟹ 1 + 3
   ⟹ 4
```

Note that the argument string for the recursive application of `len` is simpler (i.e., shorter) than the original argument. Thus `len` will eventually be applied to a `[]` argument and, hence, `len`'s evaluation will terminate.

The above definition of `len` only works for strings. How can we make it work for a list of integers or other elements?

For an arbitrary type `a`, we want `len` to take objects of type `[a]` and return an `Int` value. Thus its type signature could be:

```
len :: [a] -> Int
```

If `a` is a variable name (i.e., it begins with a lowercase letter) that does not already have a value, then the type expression `a` used as above is a *type variable*; it can represent an arbitrary type. All occurrences of a type variable appearing in a type signature must, of course, represent the same type.

An object whose type includes one or more type variables can be thought of having many different types and is thus described as having a *polymorphic type*. (Literally, its type has "many shapes".)

Polymorphism and first-class functions are powerful abstraction mechanisms: they allow irrelevant detail to be hidden.

Examples of polymorphism include:

```
head :: [a] -> a
tail :: [a] -> [a]
(:)  :: a -> [a] -> [a]
```

Tuples: `(t1,t2,···,tn)`

If `t1`, `t2`, ···, `tn` are types, where `n` is finite and `n` $\geq$ `2`, then `(t1,t2,···,tn)` is a type consisting of *n-tuples* where the various components have the type given for that position.

Unlike lists, the elements in a tuple may have different types. Also unlike lists, the number of elements in a tuple is fixed. The tuple is analogous to the `record` in Pascal or `structure` in C.

Examples:

```
(1,[2],3) :: (Int, [Int], Int)
(('a',False),(3,4)) :: ((Char, Bool), (Int, Int))

type Complex = (Float,Float)
makeComplex :: Float -> Float -> Complex
makeComplex r i = (r,i)
```

5.2 Programming with List Patterns

In the factorial examples we used integer and natural number patterns to break out various cases of a function definition into separate equations. Lists and other data types may also be used in patterns.

Pattern matching helps enable the *form of the algorithm* match the *form of the data structure*.

This is considered elegant. It is also pragmatic. The structure of the data often suggests the algorithm that is needed for a task.

In general, lists have two cases that need to be handled: the empty list and the nonempty list. Breaking a definition for a list-processing function into these two cases is usually a good place to begin.

5.2.1 Summation of a list (`sumlist`)

Consider a function `sumlist` to sum all the elements in a list of integers. That is, if the list is $v_1, v_2, v_3, \cdots, v_n$, then the sum of the list is the value resulting from inserting the addition operator between consecutive elements of the list: $v_1 + v_2 + v_3 + \cdots + v_n$.

Since addition is an *associative* operation (that is, $(x + y) + z = x + (y + z)$ for any integers x, y, and z), the above additions can be computed in any order.

What is the sum of an empty list?

Since there are no numbers to add, then, intuitively, zero seems to be the proper value for the sum.

In general, if some binary operation is inserted between the elements of a list, then the result for an empty list is the *identity* element for the operation. Since $0+x = x = x+0$ for all integers x, zero is the identity element for addition.

Now, how can we compute the sum of a nonempty list?

Since a nonempty list has at least one element, we can remove one element and add it to the sum of the rest of the list. Note that the "rest of the list" is a simpler (i.e., shorter) list than the original list. This suggests a recursive definition.

The fact that Haskell defines lists recursively as a cons of a head element with a tail list suggests that we structure the algorithm around the structure of the *beginning* of the list.

Bringing together the two cases above, we can define the function `sumlist` in Haskell as follows:

```
sumlist :: [Int] -> Int
sumlist []     = 0              -- nil list
sumlist (x:xs) = x + sumlist xs -- non-nil list
```

- All of the text between the symbol "`--`" and the end of the line represents a comment; it is ignored by the Haskell interpreter.
- This definition uses two *legs*. The equation in the first leg is used for nil list arguments, the second for non-nil arguments.
- Note the `(x:xs)` pattern in the second leg. The "`:`" is the list constructor operation *cons*.

  If this pattern succeeds, then the head element of the argument list is bound to the variable `x` and the tail of the argument list is bound to the variable `xs`. These bindings hold for the right-hand side of the equation.
- The use of the cons in the pattern simplifies the expression of the case. Otherwise the second leg would have to be stated using the `head` and `tail` selectors as follows:

  ```
  sumlist xs = head xs + sumlist (tail xs)
  ```

- Note the use of the simple name `x` to represent items of some type and the name `xs`, the same name with an `s` (for sequence) appended, to represent a list of that same type. This is a useful convention (adopted from the Bird and Wadler textbook) that helps make a definition easier to understand.
- Remember that patterns (and guards) are tested in the order of occurrence (i.e., left to right, top to bottom). Thus, in most situations, the cases should be listed from the most specific (e.g., nil) to the most general (e.g., non-nil).
- The length of a non-nil argument decreases by one for each successive recursive application. Thus `sumlist` will eventually be applied to a `[]` argument and terminate.

5.2.2 Length of a list (`length'`)

As another example, consider the function for the length of a list that we discussed earlier (as `len`). Using list patterns we can define `length'` as follows:

```
length' :: [a] -> Int
length' []     = 0              -- nil list
length' (_:xs) = 1 + length' xs -- non-nil list
```

Note the use of the wildcard pattern underscore "_". This represents a "don't care" value. In this pattern it matches the head, but no value is bound; the right-hand side of the equation does not need the actual value.

This definition is similar to the definition for `length` in the Prelude.

5.2.3 Removing adjacent duplicates (`remdups`)

Haskell supports more complicated list patterns than the ones used above. For example, consider the problem of removing adjacent duplicate elements from a list of integers. That is, we want to replace a group of adjacent elements having the same value by a single occurrence of that value.

The notion of adjacency is only meaningful when there are two or more of something. Thus, in approaching this problem, there seem to be three cases to consider:

- The argument is a list whose first two elements are duplicates; in which case one of them should be removed from the result.
- The argument is a list whose first two elements are not duplicates; in which case both elements are needed in the result.
- The argument is a list with fewer than two elements; in which case the remaining element, if any, is needed in the result.

Of course, we must be careful that sequences of more than two duplicates are handled properly.

Our algorithm thus can examine the first two elements of the list. If they are equal, then the first is discarded and the process is repeated recursively on the list remaining. If they are not equal, then the first element is retained in the result and the process is repeated on the list remaining. In either case the remaining list is one element shorter than the original list. When the list has fewer than two elements, it is simply returned as the result.

In Haskell, we can define function `remdups` as follows:

```
remdups :: [Int] -> [Int]
remdups (x:y:xs)
  | x == y = remdups (y:xs)
  | x /= y = x : remdups (y:xs)
remdups xs = xs
```

- Note the use of the pattern `(x:y:xs)`. This pattern match succeeds if the argument list has at least two elements: the head element is bound to `x`, the second element to `y`, and the tail to `xs`.
- Note the use of guards to distinguish between the cases where the two elements are equal (`==`) and where they are not equal (`/=`).
- In this definition the case patterns overlap, that is, a list with at least two elements satisfies both patterns. But since the cases are evaluated top to bottom, the list only matches the first pattern. Thus the second pattern just matches lists with fewer than two elements.

Note that `remdups` takes an argument of type `[Int]`. What if we wanted to make the list type polymorphic?

At first glance, it would seem to be sufficient to give `remdups` the polymorphic type `[a] -> [a]`. But the guards complicate the situation a bit.

Evaluation of the guards requires that Haskell be able to compare elements of the polymorphic type `a` for equality (`==`) and inequality (`/=`). For some types these comparisons may not be supported. (For example, suppose the elements are functions.) Thus we need to restrict the polymorphism to types in which the comparisons are supported.

We can restrict the range of types by using a *context* predicate. The following type signature restricts the polymorphism of type variable `a` to the built-in *class* `Eq`, the group of types for which both equality (`==`) and inequality (`/=`) comparisons have been defined:

```
remdups :: Eq a => [a] -> [a]
```

Another useful context is the class `Ord`, which is a subset of `Eq`. `Ord` denotes the class of objects for which the relational operators `<`, `<=`, `>`, and `>=` have been defined in addition to `==` and `/=`.

In most situations the type signature can be left off the declaration of a function. Haskell then attempts to infer an appropriate type. For `remdups`, the type inference mechanism would assign the type `Eq [a] => [a] -> [a]`. However, in general, it is good practice to give explicit type signatures.

5.2.4 More example patterns

The following table shows Haskell parameter patterns, corresponding arguments, and the result of the attempted match.

Pattern	Argument	Succeeds?	Bindings
`1`	`1`	yes	none
`x`	`1`	yes	`x ← 1`
`(x:y)`	`[1,2]`	yes	`x ← 1, y ← [2]`
`(x:y)`	`[[1,2]]`	yes	`x ← [1,2], y ← []`
`(x:y)`	`["olemiss"]`	yes	`x ← "olemiss", y ← []`
`(x:y)`	`"olemiss"`	yes	`x ← 'o', y ← "lemiss"`
`(1:x)`	`[1,2]`	yes	`x ← [2]`
`(1:x)`	`[2,3]`	no	
`(x:_:_:y)`	`[1,2,3,4,5,6]`	yes	`x ← 1, y ← [4,5,6]`
`[]`	`[]`	yes	none
`[x]`	`["Cy"]`	yes	`x ← "Cy"`
`[1,x]`	`[1,2]`	yes	`x ← 2`
`[x,y]`	`[1]`	no	
`(x,y)`	`(1,2)`	yes	`x ← 1, y ← 2`

5.3 Infix Operations

In Haskell, a binary operation is a function of type `t1 -> t2 -> t3` for some types `t1`, `t2`, and `t3`. We usually prefer to use infix syntax rather than prefix syntax to express the application of a binary operation. Infix operators usually make expressions easier to read; they also make statement of mathematical properties more convenient.

Often we use several infix operators in an expression. To ensure that the expression is not ambiguous (i.e., the operations are done in the desired order), we must either use parentheses to give the order explicitly (e.g., `((y * (z+2)) + x)`) or use syntactic conventions to give the order implicitly.

Typically the application order for adjacent operators of different kinds is determined by the relative precedence of the operators. For example, the multiplication (`*`) operation has a higher precedence (i.e., binding power) than addition (`+`), so, in the absence of parentheses, a multiplication will be done before an adjacent addition. That is, `x + y * z` is taken as equivalent to `(x + (y * z))`.

In addition, the application order for adjacent operators of the same binding power is determined by a *binding* (or *association*) order. For example, the addition (`+`) and subtraction `-` operations have the same precedence. By convention, they bind more strongly to the left in arithmetic expressions. That is, `x + y - z` is taken as equivalent to `((x + y) - z)`.

By convention, operators such as exponentiation (denoted by `^`) and cons bind more strongly to the right. Some other operations (e.g., division and the relational comparison operators) have no default binding order—they are said to have *free* binding.

Accordingly, Haskell provides the statements `infix`, `infixl`, and `infixr` for declaring a symbol to be an infix operator with free, left, and right binding, respectively. The first argument of these statements give the precedence level as an integer in the range 0 to 9, with 9 being the strongest binding. Normal function application has a precedence of 10.

The operator precedence table for a few of the common operations from the Prelude is below. We introduce the `++` operator in the next section.

```
infixr 8 ^                    -- exponentiation
infixl 7 *                    -- multiplication
infix  7 /                    -- division
infixl 6 +, -                 -- addition, subtraction
infixr 5 :                    -- cons
infix  4 ==, /=, <, <=, >=, > -- relational comparisons
infixr 3 &&                   -- Boolean AND
infixr 2 ||                   -- Boolean OR
```

5.4 Recursive Programming Styles

5.4.1 Appending lists (++)

Suppose we want a function that takes two lists and returns their concatenation, that is, *appends* the second list after the first. This function is a binary operation on lists much like + is a binary operation on integers.

Further suppose we want to introduce the infix operator symbol ++ for the append function. Since we want to evaluate lists lazily from their heads, we choose right binding for both cons and ++. Since append is, in a sense, an extension of cons (:), we give them the same precedence:

```
infixr 5 ++
```

Now let us consider the definition of the append function. The append operation must be defined in terms of application of already defined list operations and recursive applications of itself. The only applicable simpler operation is cons.

Note that cons operator takes an element as its left operand and a list as its right operand and returns a new list with the left operand as the head and the right operand as the tail.

Similarly, append must take a list as its left operand and a list as its right operand and return a new list with the left operand as the initial segment and the right operand as the final segment.

Given the definition of cons, it seems reasonable that an algorithm for ++ must consider the structure of its left operand. Thus we consider the cases for nil and non-nil left operands.

If the left operand is nil, then the right operand can be returned as the result. It is not necessary to evaluate the right operand!

If the left operand is non-nil, then the result consists of the left operand's head followed by the append of the left operand's tail and the right operand.

Again we have used the form of the data to guide the development of the form of the algorithm. In particular, we used the form of the left operand. Thus we have the following definition for ++ (which is similar to the definition in the Prelude):

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[] ++ xs     = xs           -- nil left operand
(x:xs) ++ ys = x:(xs ++ ys) -- non-nil left operand
```

Note the infix patterns given on the left-hand sides of the defining equations.

For the recursive application of `++`, the length of the left operand decreases by one. Hence the left operand of a `++` application eventually becomes nil, allowing the evaluation to terminate.

Simulation (this is not the actual series of steps taken during execution of the Haskell program):

```
[1,2,3] ++ [3,2,1]
  ⟹ 1:([2,3] ++ [3,2,1])
  ⟹ 1:(2:([3] ++ [3,2,1]))
  ⟹ 1:(2:(3:([] ++ [3,2,1])))
  ⟹ 1:(2:(3:[3,2,1]))
  =  [1,2,3,3,2,1]
```

The number of steps needed to evaluate `xs ++ ys` is proportional to the length of `xs`, the left operand. That is, it is $\mathcal{O}(n)$, where n is the length `xs`.

The append operation has a number of useful algebraic properties, for example, associativity and an identity element.

Associativity: For any finite lists `xs`, `ys`, and `zs`,
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.

Identity: For any finite list `xs`, `[] ++ xs = xs = xs ++ []`.

(Thus operation `++` over finite lists forms an algebraic structure called a *monoid*.) We will prove these and other properties in Section 11.

5.4.2 Reversing a list (`rev`)

Consider the problem of reversing the order of the elements in a list.

Again we can use the structure of the data to guide the algorithm development. If the argument is nil, then the function returns nil. If the argument is non-nil, then the function can append the head element at the back of the reversed tail.

```
rev :: [a] -> [a]
rev []     = []            -- nil argument
rev (x:xs) = rev xs ++ [x] -- non-nil argument
```

Given that evaluation of `++` terminates, we note that evaluation of `rev` also terminates because all recursive applications decrease the length of the argument by one.

Simulation (this is not the actual series of steps taken during execution):

```
rev "bat"
   ⟹ (rev "at") ++ "b"
   ⟹ ((rev "t") ++ "a") ++ "b"
   ⟹ (((rev "") ++ "t") ++ "a") ++ "b"
   ⟹ (("" ++ "t") ++ "a") ++ "b"
   ⟹ ("t" ++ "a") ++ "b"
   ⟹ ('t':("" ++ "a")) ++ "b"
   ⟹ "ta" ++ "b"
   ⟹ 't':("a" ++ "b")
   ⟹ 't':('a':("" ++ "b"))
   ⟹ 't':('a':"b")
   =  "tab"
```

How efficient is this function?

The evaluation of `rev` takes $\mathcal{O}(n^2)$ steps, where n is the length of the argument. There are $\mathcal{O}(n)$ applications of `rev`; for each application of `rev` there are $\mathcal{O}(n)$ applications of `++`.

Function `rev` has a number of useful properties, for example the following:

Distribution: For any finite lists `xs` and `ys`,
`rev (xs ++ ys) = rev ys ++ rev xs`.

Inverse: For any finite list `xs`, `rev (rev xs) = xs`.

5.4.3 Recursion Terminology

The function definitions examined so far are *backward recursive*. That is, for each case the recursive applications are embedded within another expression. Operationally, significant work is done after the recursive call returns.

The function definitions examined so far are also *linear recursive*. That is, only one recursive application of the function occurs in the expression for any leg. The advantage of a linear recursion versus a nonlinear one is that a linear recursion can be compiled into a *loop* in a straightforward manner.

Another recursive pattern is *forward recursion*. In a forward recursion, the outside expressions for cases are recursive applications of the function. Operationally, significant work is done as the recursive calls are being made (e.g., in the argument list of a recursive call).

A function definition is *tail recursive* if it is both forward recursive and linear recursive. In a tail recursion the last action performed before the return is a recursive call.

Tail recursive definitions are easy to compile into efficient loops. There is no need to save the state of the unevaluated expressions for the higher level calls; the result of a recursive call can be returned directly as the caller's result. This process is called *tail call optimization*, *tail call elimination*), or sometimes *proper tail calls*.

5.4.4 Tail recursive reverse (`reverse'`)

Now let's look at the problem of reversing a list again to see whether we can devise a "more efficient" tail recursive solution.

The common technique for converting a backward linear recursive definition like `rev` into a tail recursive definition is to use an *accumulating parameter* (also called an *accumulator*) to build up the desired result incrementally. A possible definition follows:

```
rev' [] ys     = ys
rev' (x:xs) ys = rev' xs (x:ys)
```

In this definition parameter `ys` is the accumulating parameter. The head of the first argument becomes the new head of the accumulating parameter for the tail recursive call. The tail of the first argument becomes the new first argument for the tail recursive call.

We know that `rev'` terminates because, for each recursive application, the length of the first argument decreases toward the base case of `[]`.

We note that `rev xs` is equivalent to `rev' xs []`. We will prove this in Section 11.6.

To define a single-argument replacement for `rev`, we can embed the definition of `rev'` as an *auxiliary* function within the definition of a new function `reverse'`. (This is similar to function `reverse` in the Prelude.)

```
reverse' :: [a] -> [a]
reverse' xs = rev xs []
              where rev []     ys = ys
                    rev (x:xs) ys = rev xs (x:ys)
```

The `where` clause introduces the *local definition* `rev'` that is only known within the right-hand side of the defining equation for the function `reverse'`.

How efficient is this function?

The evaluation of `reverse'` takes $\mathcal{O}(n)$ steps, where n is the length of the argument. There is one application of `rev'` for each element; `rev'` requires a single cons operation in the accumulating parameter.

Where did the increase in efficiency come from? Each application of `rev` applies `++`, a linear time (i.e., $\mathcal{O}(n)$) function. In `rev'`, we replaced the applications of `++` by applications of cons, a constant time (i.e., $\mathcal{O}(1)$) function.

In addition, a compiler or interpreter that does tail call optimization can translate this tail recursive call into a loop on the host machine.

5.4.5 Local definitions (`let` and `where`)

The `let` expression is useful whenever a nested set of definitions is required. It has the following syntax:

`let` *local_definitions* `in` *expression*

A `let` may be used anywhere that an expression my appear in a Haskell program.

For example, consider a function `f` that takes a list of integers and returns a list of their squares incremented by one:

```
f :: [Int] -> [Int]
f [] = []
f xs = let square a = a * a
           one      = 1
           (y:ys)   = xs
       in (square y + one) : f ys
```

- `square` represents a function of one variable.
- `one` represents a constant, that is, a function of zero variables.
- `(y:ys)` represents a pattern match binding against argument `xs` of `f`.
- Reference to `y` or `ys` when argument `xs` of `f` is nil results in an error.
- Local definitions `square`, `one`, `y`, and `ys` all come into scope simultaneously; their scope is the expression following the `in` keyword.
- Local definitions may access identifiers in outer scopes (e.g., `xs` in definition of `(y:ys)`) and have definitions nested within themselves.
- Local definitions may be recursive and call each other.

The `let` clause introduces symbols in a bottom-up manner: it introduces symbols before they are used.

The `where` clause is similar semantically, but it introduces symbols in a top-down manner: the symbols are used and then defined in a `where` that follows.

The `where` clause is more versatile than the `let`. It allows the scope of local definitions to span over several guarded equations while a `let`'s scope is restricted to the right-hand side of one equation.

For example, consider the definition:

```
g :: Int -> Int
g n | (n `mod` 3) == x = x
    | (n `mod` 3) == y = y
    | (n `mod` 3) == z = z
                       where x = 0
                             y = 1
                             z = 2
```

- The scope of this `where` clause is over *all three guards* and their respective right-hand sides. (Note that the `where` begins in the same column as the `=` rather than to the right as in `rev'`.)
- Note the use of the modulo function `mod` as an infix operator. The backquotes (`) around a function name denotes the infix use of the function.

5.4.6 Fibonacci numbers

The first two elements of the (second-order) Fibonacci sequence are 0 and 1; each successive element is the sum of the previous two elements. Element `n` of this sequence can be computed by the following Haskell function:

```
fib :: Int -> Int
fib 0         = 0
fib 1         = 1
fib n | n > 1 = fib (n-2) + fib (n-1)
```

This function definition uses the `n > 1` guard on the third pattern to match natural numbers at least 2.

An equivalent, but more efficient, definition uses forward recursion with two additional parameters to avoid recomputing Fibonacci numbers:

```
fib' :: Int -> Int
fib' n = fib'' n 0 1
         where fib'' 0 p q         = p
               fib'' n p q | n > 0 = fib'' (n-1) q (p+q)
```

As `n` gets large, `fib'` is much more efficient than `fib` as shown in by the following interaction with the Glasgow Haskell Compiler interactive environment GHCi.

```
*Chap05> :set +s
*Chap05> fib 10
55
(0.00 secs, 1030504 bytes)
*Chap05> fib' 10
55
(0.00 secs, 1064536 bytes)
*Chap05> fib 20
6765
(0.01 secs, 5241848 bytes)
*Chap05> fib' 20
6765
(0.00 secs, 1027856 bytes)
*Chap05> fib 30
832040
(1.63 secs, 517915736 bytes)
*Chap05> fib' 30
832040
(0.00 secs, 1066728 bytes
```

Evaluation of `fib n` takes $\mathcal{O}(fib(\mathtt{n}))$ steps (i.e., reductions); `fib' n` takes $\mathcal{O}(\mathtt{n})$ steps.

Of course, we can change the parameter and return types to `Integer` to extend the domain of the Fibonacci function.

```
fibU :: Integer -> Integer
fibU n = fib'' n 0 1
         where fib'' 0 p q         = p
               fib'' n p q | n > 0 = fib'' (n-1) q (p+q)
```

```
*Chap05> fibU 1400
171084769023402272412497195132318214773827498980269200415508
374983434801725093580135931503892336784149493603823152250635
371361016671790887791259870264957823133253627917432203111969
046232293847634906170753886426961398933540586605703999270478
629695251633063663385111164638788547269868360792
(0.01 secs, 2097384 bytes)
```

5.5 More List Operations

5.5.1 Element selection (`!!`)

The list selection operator `!!` in the expression `xs!!n` selects element `n` of list `xs` where the head is in position 0. It is defined in the Prelude similar to the way `!!` is defined below:

```
infixl 9 !!

(!!) :: [a] -> Int -> a
xs     !! n | n < 0 = error "!! negative index"
[]     !! _         = error "!! index too large"
(x:_)  !! 0         = x
(_:xs) !! n         = xs !! (n-1)
```

5.5.2 List-breaking operations (`take` and `drop`)

Two additional useful functions from the Prelude are `take` and `drop` (shown below as `take'` and `drop'` to avoid name conflicts with the Prelude). Function `take` takes a number `n` and a list and returns the first `n` elements of the list. Function `drop` takes a number `n` and a list and returns the list remaining after the first `n` elements are removed.

```
take' :: Int -> [a] -> [a]
take' n _ | n <= 0 = []
take' _ []         = []
take' n (x:xs)     = x : take' (n-1) xs

drop' :: Int -> [a] -> [a]
drop' n xs | n <= 0 = xs
drop' _ []          = []
drop' n (_:xs)      = drop' (n-1) xs
```

Each of these functions has two arguments, a natural number and a list. But, when either the first argument is `0` (or negative) or the second argument is `[]`, it is not necessary to consider the value of the other argument in determining the result.

Thus each of these functions has three cases: when the first argument is zero, when the second argument is nil, and when neither is the case.

For the recursive applications of `take` and `drop`, both arguments decrease in size. Thus evaluation eventually terminates.

Examples: `take 2 "oxford"` $\Longrightarrow$ `"ox"`
`drop 2 "oxford"` $\Longrightarrow$ `"ford"`

For all naturals `n` and finite lists `xs`, functions `take` and `drop` satisfy the following property, which we will prove in Section 11.5: `take n xs ++ drop n xs = xs`

5.5.3 List-combining operations (`zip`)

Another useful function in the Prelude is `zip` (shown below as `zip'`) which takes two lists and returns a list of pairs of the corresponding elements. That is, the function fastens the lists together like a zipper. It's definition is similar to `zip'` given below:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys -- zip.1
zip' _      _      = []                 -- zip.1
```

Function `zip` applies a *tuple-forming* operation to the corresponding elements of two lists. It stops the recursion when either list argument becomes nil.

Example: `zip [1,2,3] "oxford"` $\Longrightarrow$ `[(1,'o'),(2,'x'),(3,'f')]`

The Prelude includes versions of `zip` that handle the tuple-formation for up to seven input lists: `zip3` $\cdots$ `zip7`.

5.6 Rational Arithmetic Package

As a larger example, suppose we want to implement a group of Haskell functions to do rational number arithmetic [2].

The first thing we must do is determine how to represent the rational numbers in Haskell. In mathematics we usually write rational numbers in the form $\frac{\mathtt{x}}{\mathtt{y}}$ where $\mathtt{y} \neq \mathtt{0}$. In Haskell a straightforward way to represent $\frac{\mathtt{x}}{\mathtt{y}}$ is by a tuple `(x,y)`.

Thus we define a type synonym `Rat` to denote this type.

```
type Rat = (Int, Int)
```

For example, `(1,7)`, `(-1,-7)`, `(3,21)`, and `(168,1176)` all represent $\frac{1}{7}$.

As with any value that can be expressed in many different ways, it is useful to define a single *canonical* (or *normal*) form for representing values in the rational number type `Rat`.

It is convenient for us to choose the form `(x,y)` where `y > 0` and `x` and `y` are relatively prime. (That is, they have no common divisors except 1.) We also represent zero canonically by `(0,1)`.

This representation has the advantage that the magnitudes of the numerator `x` and denominator `y` are kept small, thus reducing problems with overflow arising during arithmetic operations.

We require that the arithmetic operations in the package return values in the canonical form and that the package provide a function to convert (or normalize) rational numbers to this form. Thus we provide a function `normRat` that takes a `Rat` and returns the `Rat` in canonical form that has the same value as the argument.

An appropriate definition is shown below.

```
normRat :: Rat -> Rat
normRat (x,0) = error ( "Invalid rational number (in normRat) "
                      ++ showRat (x,0) ++ "\n" )
normRat (0,y) = (0,1)
normRat (x,y) = (a `div` d, b `div` d)
                where a = (signum' y) * x
                      b = abs y
                      d = gcd' a b
```

This function uses functions `signum`, `abs`, and `gcd` from the Prelude to get the sign (as -1, 0, or 1), absolute value, and greatest common divisor, respectively. It also uses the function `error` to denote an error termination of the program.

The function `error` from the Prelude causes evaluation of the program to halt and the argument string to be printed. This function is implemented as a primitive operation built into the Haskell interpreter. In `normRat`, it is used to print an error message when an invalid rational number representation is discovered. The `error` call uses function `showRat` defined later.

The function `signum` from the Prelude (shown here as `signum'`) takes a number (an object in class `Num`) and returns the integer `-1`, `0`, or `1` when the number is negative, zero, or positive, respectively. (Numbers are also ordered and hence in class `Ord`.)

```
signum' :: (Num a, Ord a) => a -> Int
signum' n | n == 0 =  0
          | n > 0  =  1
          | n < 0  = -1
```

The function `gcd` from the Prelude (show here as `gcd'`) takes two integers and returns their "greatest common divisor".

```
gcd' :: Int -> Int -> Int
gcd' x y = gcd'' (abs x) (abs y)
         where gcd'' x 0 = x
               gcd'' x y = gcd'' y (x `rem` y)
```

(Prelude operation `rem` returns the remainder from dividing its first operand by its second. Remember that enclosing the function name in backquotes as in `` `rem` `` allows a two-argument function to be applied in an infix form.)

The package must provide the usual arithmetic operations for values of type `Rat`: negation, addition, subtraction, multiplication, and division. We can implement these using the knowledge of arithmetic on fractions that we learned in elementary school. The arithmetic functions must return their results in canonical form.

```
negRat :: Rat -> Rat
negRat (a,b) = normRat (-a,b)

addRat, subRat, mulRat, divRat :: Rat -> Rat -> Rat
addRat (a,b) (c,d) = normRat (a*d + c*b, b*d)
subRat (a,b) (c,d) = normRat (a*d - c*b, b*d)
mulRat (a,b) (c,d) = normRat (a*c, b*d)
divRat (a,b) (c,d) = normRat (a*d, b*c)
```

The package must also provide the usual comparison operations for values of type `Rat`. To compare two rational numbers, we express their values in terms of a common denominator and then compare the numerators using the integer comparisons.

We can define the equality comparison function `eqRat` as follows. The other rational number comparisons are the same except that the corresponding integer comparisons are used.

```
eqRat :: Rat -> Rat -> Bool
eqRat (a,0) (c,d) = errRat (a,0)
eqRat (a,b) (c,0) = errRat (c,0)
eqRat (a,b) (c,d) = (a*d == b*c)

errRat (x,y) = error ("Invalid rational number"
                      ++ showRat (x,y))
```

Finally, to allow rational numbers to be displayed in the normal fractional representation, we include function `showRat` in the package. Function `show`, found in the Prelude, is used here to convert an integer to the usual string format.

```
showRat :: Rat -> String
showRat (a,b) = show a ++ "/" ++ show b
```

Just for fun, we could also include this useless function:

```
youDirtyRat :: String
youDirtyRat = "You dirty rat!\n" ++ youDirtyRat
```

5.7 Exercises

The following exercises call for the implementation of Haskell programs. For each function, informally argue that all the functions terminate. Take care that special cases and error conditions are handled in a reasonable way.

1. Write a Haskell function `xor` that takes two Booleans and returns the "exclusive-or" of the two values. An exclusive-or operation returns `True` when exactly one of its arguments is `True` and returns `False` otherwise.

2. Write a Haskell function `mult` that takes two natural numbers and returns their product. The function must not use the multiplication (`*`) or division (`/`) operators.

3. Write a Haskell function to compute the maximum value in a nonempty list of integers. Generalize the function by making it polymorphic, accepting a value from any ordered type.

4. Write a Haskell function `adjpairs` that takes a list and returns the list of all pairs of adjacent elements. For example, `adjpairs [2,1,11,4]` returns `[(2,1), (1,11), (11,4)]`.

5. Write a Haskell function `mean` that takes a list of integers and returns the mean (i.e., average) value for the list.

6. Hailstone functions [6, 9].

   (a) Write a Haskell function `hailstone` to implement the following function:

$$hailstone(n) = \begin{cases} 1, & \text{for } n = 1 \\ hailstone(n/2), & \text{for } n > 1, n \text{ even} \\ hailstone(3 * n + 1), & \text{for } n > 1, n \text{ odd} \end{cases}$$

   Note that an application of the `hailstone` function to the argument `3` would result in the following "sequence" of "calls" and would ultimately return the result `1`.

```
hailstone 3
   hailstone 10
      hailstone 5
         hailstone 16
            hailstone 8
               hailstone 4
                  hailstone 2
                     hailstone 1
```

   For further thought: What is the domain of the *hailstone* function?

(b) Write a Haskell function that computes the results of the `hailstone` function for each element of a list of positive integers. The value returned by the `hailstone` function for each element of the list should be displayed.

(c) Modify the `hailstone` function to return the function's "path." That is, each application of this path function should return a list of integers instead of a single integer. The list returned should consist of the arguments of the successive calls to the `hailstone` function necessary to compute the result. For example, the `hailstone 3` example above should return `[3,10,5,16,8,4,2,1]`.

7. Number base conversion.

   (a) Write a Haskell function `natToBin` that takes a natural number and returns its binary representation as a list of `0`'s and `1`'s with the most significant digit at the head. For example, `natToBin 23` returns `[1,0,1,1,1]`. (Note: Prelude function `rem` returns the remainder from dividing its first argument by its second. Enclosing the function name in backquotes as in `` `rem` `` allows a two-argument function to be applied in an infix form.)

   (b) Generalize `natToBin` to function `natToBase` that takes a base `b` ($b \geq 2$) and a natural number and returns the base `b` representation of the natural number as a list of integer digits with the most significant digit at the head. For example, `natToBase 5 42` returns `[1,3,2]`.

   (c) Write Haskell function `baseToNat`, the inverse of the `natToBase` function. For any base `b` ($b \geq 2$) and natural number `n`:

```
baseToNat b (natToBase b n) = n
```

8. Write a Haskell function `merge` that takes two increasing lists of integers and merges them into a single increasing list (without any duplicate values). A list is *increasing* if every element is less than (`<`) its successors. Successor means an element that occurs later in the list, i.e., away from the head. Generalize the function by making it polymorphic.

9. Design a package of set operations. Choose a Haskell representation for sets. Implement functions to make sets from lists and vice versa, to insert and delete elements from sets, to do set union, intersection, and difference, to test for equality and subset relationships, to determine cardinality, and so forth.

10. Bag operation package.

    A bag (or multiset) is a collection of elements; each element may occur one or more times in the bag. Choose an efficient representation for bags. Each bag should probably have a unique representation.

    Define a package of bag operations, including the following functions. For the functions that return bags, make sure that a valid representation of the bag is returned.

    `listToBag` takes a list of elements and returns a bag containing exactly those elements. The number of occurrences of an element in the list and in the resulting bag is the same.

    `bagToList` takes a bag and returns a list containing exactly the elements occurring in the bag. The number of occurrences of an element in the bag and in the resulting list is the same.

    `bagToSet` takes a bag and returns a list containing exactly the *set* of elements contained in the bag. Each element occurring one or more times in the bag will occur exactly once in the list returned.

    `bagEmpty` takes a bag and returns `True` if the bag is empty and `False` otherwise.

    `bagCard` takes a bag and returns its cardinality (i.e., the total number of occurrences of all elements).

    `bagElem` takes an element and a bag and returns `True` if the element occurs in the bag and `False` otherwise.

    `bagOccur` takes an element and a bag and returns the number of times the element occurs in the bag.

    `bagEqual` takes two bags and returns `True` if the two bags are equal (i.e., the same elements and same number of occurrences of each) and `False` otherwise.

    `bagSubbag` takes two bags and returns `True` if the first is a subbag of the second and `False` otherwise. X is a subbag of Y if every element of X occurs in Y at least as many times as it does in X.

    `bagUnion` takes two bags and returns their bag union. The union of bags X and Y contains all elements that occur in either X or Y; the number of occurrences of an element in the union is the number in X or in Y, whichever is greater.

    `bagIntersect` takes two bags and returns their bag intersection. The intersection of bags X and Y contains all elements that occur in both X and Y; the number of occurrences of an element in the intersection is the number in X or in Y, whichever is lesser.

`bagSum` takes two bags and returns their bag sum. The sum of bags X and Y contains all elements that occur in X or Y; the number of occurrences of an element is the sum of the number of occurrences in X and Y.

`bagDiff` takes two bags and returns the bag difference, first argument minus the second. The difference of bags X and Y contains all elements of X that occur in Y fewer times; the number of occurrences of an element in the difference is the number of occurrences in X minus the number in Y.

`bagInsert` takes an element and a bag and returns the bag with the element inserted. Bag insertion either adds a single occurrence of a new element to the bag or increases the number of occurrences of an existing element by one.

`bagDelete` takes an element and a bag and returns the bag with the element deleted. Bag deletion either removes a single occurrence of the element from the bag or decreases the number of occurrences of the element by one.

11. Unbounded precision arithmetic package for natural numbers (i.e., nonnegative integers).

    (a) Define a type synonym `BigNat` to represent these unbounded precision natural numbers as lists of `Int`. Let each element of the list denote a decimal digit of the "big natural" number represented, with the *least* significant digit at the head of the list and the remaining digits given in order of *increasing* significance. For example, the integer value 22345678901 is represented as `[1,0,9,8,7,6,5,4,3,2,2]`. Use the following "canonical" representation: the value `0` is represented by the list `[0]` and positive numbers by a list without "leading" `0` digits (i.e., `126` is `[6,2,1]` not `[6,2,1,0,0]`). You may use the nil list `[ ]` to denote an error value.

    Define a package of basic arithmetic operations, including the following functions. Make sure that `BigNat` values returned by these functions are in canonical form.

    `intToBig` takes a nonnegative `Int` and returns the `BigNat` with the same value.

    `strToBig` takes a `String` containing the value of the number in the "usual" format (i.e., decimal digits, left to right in order of *decreasing* significance with zero or more leading spaces, but with no spaces or punctuation embedded within the number) and returns the `BigNat` with the same value.

    `bigToStr` takes a `BigNat` and returns a `String` containing the value of the number in the "usual" format (i.e., left to right in order of *decreasing* significance with no spaces or punctuation).

`bigComp` takes two `BigNat`s and returns the `Int` value `-1` if the value of the first is less than the value of the second, the value `0` if they are equal, and the value `1` if the first is greater than the second.

`bigAdd` takes two `BigNat`s and returns their sum as a `BigNat`.

`bigSub` takes two `BigNat`s and returns their difference as a `BigNat`, first argument minus the second.

`bigMult` takes two `BigNat`s and returns their product as a `BigNat`.

(b) Use the package to generate a table of factorials for the naturals 0 through 25. Print the values from the table in two *right-justified* columns, with the number on the left and its factorial on the right. (Allow about 30 columns for `25!`.)

(c) Use the package to generate a table of Fibonacci numbers for the naturals 0 through 50.

(d) Generalize the package to handle signed integers. Add the following new function:

`bigNeg` returns the negation of its `BigNat` argument.

(e) Add the following functions to the package:

`bigDiv` takes two `BigNat`s and returns, as a `BigNat`, the quotient from dividing the first argument by the second.

`bigRem` takes two `BigNat`s and returns, as a `BigNat`, the remainder from dividing the first argument by the second.

12. Define the following set of text-justification functions. You may want to use Prelude functions like `take`, `drop`, and `length`.

`spaces' n` returns a string of length `n` containing only space characters (i.e., the character `' '`).

`left' n xs` returns a string of length `n` in which the string `xs` begins at the head (i.e., left end). Examples: `left' 3 "ab"` yields `"ab "`; `left' 3 "abcd"` yields `"abc"`.

`right' n xs` returns a string of length `n` in which the string `xs` ends at the tail (i.e., right end). Examples: `right' 3 "bc"` yields `" bc"`; `right' 3 "abcd"` yields `"bcd"`.

`center' n xs` returns a string of length `n` in which the string `xs` is approximately centered. Example: `center' 4 "bc"` yields `" bc "`.

13. Consider simple mathematical expressions consisting of integer constants, variable names, parentheses, and the binary operators `+`, `-`, `*`, and `/`. For the purposes of this exercise, an *expression* is a string that satisfies the following (extended) BNF grammar and lexical conventions :

    expression ::= *term* { *addOp term* }
    term ::= *factor* { *mulOp factor* }
    factor ::= *number*
    | *identifier*
    | `(` *expression* `)`

    - The characters in an input string are examined left to right to form "lexical tokens". The tokens of the expression "language" consist of *addOp*s, *mulOp*s, *identifier*s, *number*s, and left and right parentheses.
    - An expression may contain space characters at any position except within a lexical token.
    - An *addOp* token is either a "`+`" or a "`-`"; a *mulOp* token is either a "`*`" or a "`/`".
    - An *identifier* is a string of one or more contiguous characters such that the leftmost character is a letter and the remaining characters are either letters, digits, or underscore characters. Examples: "`Hi1`", "`lo23_1`", "`this_is_2_long__`"
    - A *number* is a string of one or more contiguous characters such that all (including the leftmost) are digits. Examples: "`1`", "`23456711`"
    - All *identifier* and *number* tokens extend as far to the right as possible. For example, consider the string " `A123  12B3+2 )` ". (Note the space and right parenthesis characters). This string consists of the six tokens "`A123`", "`12`", "`B3`", "`+`", "`2`", and "`)`".

    Define a Haskell function `valid` that takes a `String` and returns `True` if the string is an *expression* as described above and returns `False` otherwise.

    Hint: If you need to return more than one value from a function, you can do so by returning a tuple of those values. This tuple can be decomposed by Prelude functions such as `fst` and `snd`.

    Hint: Use of the `where` or `let` features can simplify many functions. You may find Prelude functions such as `span`, `isSpace`, `isDigit`, `isAlpha`, and `isAlphanum` useful.

    Hint: You may want to consider organizing your program as a simple recursive descent recognizer for the expression language.

14. Extend the mathematical expression recognizer of the previous exercise to *evaluate* integer expressions with the given syntax. The four binary operations have their usual meanings.

    Define a function `eval e st` that evaluates expression `e` using symbol table `st`. If the expression `e` is syntactically valid, `eval` returns a pair `(True,val)` where `val` is the value of `e`. If `e` is not valid, `eval` returns `(False,0)`.

    The symbol table consists of a list of pairs, in which the first component of a pair is the variable name (a string) and the second is the variable's value (an integer).

    Example: `eval "(2+x) * y" [("y",3),("a",10),("x",8)]` yields `(True,30)`.

6 HIGHER-ORDER FUNCTIONS

6.1 Maps

Consider the following two functions.‘ Notice their type signatures and patterns of recursion.

The first, `squareAll`, takes a list of integers and returns the corresponding list of squares of the integers.

```
squareAll :: [Int] -> [Int]
squareAll []     = []
squareAll (x:xs) = (x * x) : squareAll xs
```

The second, `lengthAll`, takes a list of lists and returns the corresponding list of the lengths of the element lists; it uses the Prelude function `length`.

```
lengthAll :: [[a]] -> [Int]
lengthAll []       = []
lengthAll (xs:xss) = (length xs) : lengthAll xss
```

Although these standard functions take different kinds of data (a list of integers versus a list of polymorphically typed lists) and apply different operations (squaring versus list length), they exhibit the same *pattern of computation*. That is, both take a list and apply some function to each element to generate a resulting list of the same size as the original.

The combination of polymorphic typing and higher-order functions allow us to abstract this pattern of computation into a standard function.

Most programming languages support *first-order functions*; in a first-order function the arguments and the result are ordinary data items.

Some programming languages support *higher-order functions*; in a higher-order function the arguments and the result may be functions. Haskell supports higher-order functions; in fact, it does not distinguish between first-order and higher-order functions.

A higher-order function is not such a mysterious beastie. For example, we used a higher-order function frequently in calculus class. The differentiation operator is a function that takes a function and returns a function.

Haskell abstracts the pattern of computation common to `squareAll` and `lengthAll` as the very useful function `map` found in the Prelude (shown as `map'` below to avoid a name conflict):

```
map' :: (a -> b) -> [a] -> [b]  -- map
map' f []     = []
map' f (x:xs) = f x : map' f xs
```

Function `map` takes a function `f` of type `a -> b` and a list of type `[a]`, applies the function to each element of the list, and produces a list of type `[b]`.

Thus we can redefine `squareAll` and `lengthAll` as follows:

```
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x

lengthAll2 :: [[a]] -> [Int]
lengthAll2 xss = map' length xss
```

Above we defined Prelude function `map` as a recursive function that transforms the elements of a list one by one. However, it is often more useful to think of `map` in one of two ways:

1. as a powerful list operator that transforms every element of the list. We can combine `map` with other powerful operators to quickly construct powerful list processing programs.

   We can consider `map` as operating on every element of the list simultaneously. In fact, an implementation could use separate processors to transform each element: this is essentially the `map` operation in Google's `mapReduce` distributed "big data" processing framework.

2. as a operator node in a dataflow network. A stream of data objects flows into the `map` node. The `map` node transforms each object by applying the argument function. Then the data flows out to the next node of the network. The lazy evaluation of the Haskell functions enables such an implementation.

6.2 Filters

Consider the following two functions.

The first, `getEven`, takes a list of integers and returns the list of those integers that are even (i.e., are multiples of 2). The function preserves the relative order of the elements in the list.

```
getEven :: [Int] -> [Int]
getEven []            = []
getEven (x:xs)
          | even x    = x : getEven xs
          | otherwise = getEven xs
```

The second, `doublePos`, takes a list of integers and returns the list of doubles of the positive integers from the input list; it preserves the order of the elements.

```
doublePos :: [Int] -> [Int]
doublePos []          = []
doublePos (x:xs)
          | 0 < x     = (2 * x) : doublePos xs
          | otherwise = doublePos xs
```

Function `even` is from the Prelude; it returns `True` if its argument is evenly divisible by 2 and returns `False` otherwise.

Haskell abstracts the pattern of computation common to `getEven` and `doublePos` as the useful function `filter` found in the Prelude (shown as `filter'` below to avoid a name conflict):

```
filter' :: (a -> Bool) -> [a] -> [a]  -- filter
filter' _ []    = []
filter' p (x:xs)
    | p x       = x : xs'
    | otherwise = xs'
                where xs' = filter' p xs
```

Function `filter` takes a predicate `p` of type `a -> Bool` and a list of type `[a]` and returns a list containing those elements that satisfy `p`, in the same order as the input list. Note that the keyword `where` begins in the same column as the `=` in the defining equations; thus the scope of the definition of `xs'` extends over both legs of the definition.

Therefore, we can redefine `getEven` and `doublePos` as follows:

```
getEven2 :: [Int] -> [Int]
getEven2 xs = filter' even xs

doublePos2 :: [Int] -> [Int]
doublePos2 xs = map' dbl (filter' pos xs)
                where dbl x  = 2 * x
                      pos x = (0 < x)
```

Note that function `doublePos2` exhibits both the `filter` and the `map` patterns of computation.

The standard higher-order functions `map` and `filter` allowed us to restate the four-line definitions of `getEven` and `doublePos` in just one line, except that `doublePos` required two lines of local definitions. (In Sections 6.5, 6.8, and 7.2.4 we see how to eliminate the local definitions.)

6.3 Folds

The `++` operator concatenates two lists of some type into one list. But suppose we want to concatenate several lists together into one list. In particular, we want a function `concat'` to concatenate a list of lists of some type into a list of that type with the order of the input lists and their elements preserved. The following function does that.

```
concat' :: [[a]] -> [a]  -- concat
concat' []       =  []                 -- nil list of lists
concat' (xs:xss) =  xs ++ concat' xss  -- non-nil list of lists
```

This definition for `concat'` is similar to the definition for `concat` in the Prelude.

Remember the `sumlist` function we developed in Section 5.2.1.

```
sumlist :: [Int] -> Int
sumlist []     = 0               -- nil list
sumlist (x:xs) = x + sumlist xs  -- non-nil list
```

What do `concat'` and `sumlist` have in common?

Both functions exhibit the same pattern of computation. They both take a list of elements and insert a binary operator between all the consecutive elements of the list in order to reduce the list to a single value. Function `concat'` uses the binary operation `++`; `sumlist` uses `+`.

In addition, note that `concat'` returns `[]` when its argument is nil; if this is a recursive call, the return value is appended to the right of the previous results. Similarly, `sumlist` returns `0` when its argument is nil. The values `[]` and `0` are the identity elements for `++` and `+`, respectively.

For example, given the operation `+` and list `[x1,x2,x3,x4]`, `sumlist` computes the value of `x1 + (x2 + (x3 + (x4 + 0)))`. Note the binding (i.e., grouping by parentheses) from the right end of the list (i.e., the tail) toward the left. Also note the `0` as the rightmost operand.

Haskell abstracts the pattern of computation common to `concat'` and `sumlist` as the function `foldr` (pronounced "fold right") found in the Prelude (shown as `foldrX` below to avoid a name conflict).

```
foldrX :: (a -> b -> b) -> b -> [a] -> b  -- foldr
foldrX f z []     = z
foldrX f z (x:xs) = f x (foldrX f z xs)
```

The first argument of `foldr` is a binary operation (with type `a -> b -> b`) and the third argument is the list (with type `[a]`) upon which to operate. The second argument is a "seed" value (with type `a`) used to start the operation on the right; it is also the value returned by `foldr` for nil lists.

Note how the second leg implements the right binding of the operation:

$$\texttt{foldr } (\oplus)\ \texttt{z } [\texttt{x}_1,\texttt{x}_2,\ldots,\texttt{x}_n] = (\texttt{x}_1 \oplus (\texttt{x}_2 \oplus (\cdots (\texttt{x}_n \oplus \texttt{z}) \cdots)))$$

We can see that the rightmost application of the function $\oplus$ has the last element of the list as its left operand and the seed value as its right operand (i.e., $\texttt{x}_n \oplus \texttt{z}$). Typically the seed value will be the *right identity* for the operation $\oplus$. (An element e is a *right identity* of $\oplus$ if and only if $x \oplus e = x$ for any x.)

In Haskell, `foldr` is called a *fold* operation. Other languages sometimes call this a *reduce* or *insert* operation.

Using `foldr`, we can now restate the definitions for `concat'` and `sumlist` (as `concat2` and `sumlist2` below):

```
concat2:: [[a]] -> [a]
concat2 xss = foldrX (++) [] xss

sumlist2 :: [Int] -> Int
sumlist2 xs = foldrX (+) 0 xs
```

As further examples, consider the folding of the Boolean operators `&&` ("and") and `||` ("or") over lists of Boolean values as Prelude functions `and` and `or` (shown as `and'` and `or'` below to avoid name conflicts):

```
and', or' :: [Bool] -> Bool  -- and, or
and' xs = foldrX (&&) True xs
or'  xs = foldrX (||) False xs
```

Although their definitions look different, `and'` and `or'` are actually identical to functions `and` and `or` in the Prelude.

As noted above, function `foldr` binds (i.e., groups by parentheses) the operation from the right toward the left. An alternative fold operation is one that binds from the left toward the right. The `foldl` (pronounced "fold left") function implements this computational pattern. It uses the seed value to start the operation at the head of the list. Typically, the seed value will be the *left identity* of the operation.

The definition of `foldl` from the Prelude is similar to `foldlX` shown below:

```
foldlX :: (a -> b -> a) -> a -> [b] -> a  -- foldl
foldlX f z []     = z
foldlX f z (x:xs) = foldlX f (f z x) xs
```

Note how the second leg of `foldlX` implements the left binding of the operation. In the recursive call of `foldlX` the "seed value" argument is used as an accumulating parameter.

$$\texttt{foldl}\ (\oplus)\ \texttt{z}\ [\texttt{x}_1,\texttt{x}_2,\ldots,\texttt{x}_n] = (\cdots((\texttt{z} \oplus \texttt{x}_1) \oplus \texttt{x}_2) \oplus \cdots \texttt{x}_n)$$

Also note how the types of `foldr` and `foldl` differ.

If $\oplus$ is an associative binary operation of type `t -> t -> t` with identity element `z` (i.e., a monoid), then:

$$\texttt{foldr}\ (\oplus)\ \texttt{z xs} = \texttt{foldl}\ (\oplus)\ \texttt{z xs}$$

Bird and Wadler call this property the *first duality theorem* [2].

Since both `+` and `++` are associative operations with identity elements, `sumlist'` and `concat''` can be implemented with either `foldr` or `foldl`.

Which is better?

Depending upon the nature of the operation, an implementation using `foldr` may be more efficient than `foldl` or vice versa. We defer a more complete discussion of the efficiency until we study evaluation strategies in Section 13.

As a rule of thumb, however, if the operation $\oplus$ is *nonstrict* (see Section 6.4) in either argument, then it is usually better to use `foldr`. That form takes better advantage of lazy evaluation.

If the operation $\oplus$ is *strict* (see Section 6.4) in both arguments, then it is often better (i.e., more efficient) to use the optimized version of `foldl` called `foldl'`. We'll discuss this more in Section 13.6.

As examples of the use of function `foldl'` (from module `Data.List`), consider the following functions, which are similar to `sum` and `product` in the Prelude (shown below as `sum'` and `product'` to avoid name conflicts):

```
sum', product' :: Num a => [a] -> a
sum' xs     = foldl' (+) 0 xs  -- sum
product' xs = foldl' (*) 1 xs  -- product
```

Note that these functions can operate on lists of elements from some type in class `Num`.

6.4 Strictness

Some expressions cannot be reduced to a simple value, for example, `1/0`. The attempted evaluation of such expressions either return an error immediately or cause the interpreter to go into an "infinite loop".

In our discussions of functions, it is often convenient to assign the symbol $\perp$ (pronounced "bottom") as the value of expressions like `1/0`. We use $\perp$ is a polymorphic symbol—as a value of every type.

The symbol $\perp$ is not in the Haskell syntax and the interpreter cannot actually generate the value $\perp$. It is merely a name for the value of an expression in situations where the expression cannot really be evaluated. It's use is somewhat analogous to use of symbols such as ∞ in mathematics.

Although we cannot actually produce the value $\perp$, we can, conceptually at least, apply any function to $\perp$.

If `f` $\perp$ `=` $\perp$, then we say that the function is *strict*; otherwise, it is *nonstrict* (sometimes called *lenient*).

That is, a strict argument of a function must be evaluated before the final result can be computed. A nonstrict argument of a function may not need to be evaluated to compute the final result.

Assume that lazy evaluation is being used and consider the function `two` that takes an argument of any type and returns the integer value two.

```
two :: a -> Int
two x = 2
```

The function `two` is nonstrict. The argument expression is not evaluated to compute the final result. Hence, `two` $\perp$ `= 2`.

Strict examples: Function `rev` (discussed in Section 5.4.2) is strict in its one argument. The arithmetic operations (e.g., `+`) are strict in both arguments.

Nonstrict examples: Operation `++` is strict in its first argument, but nonstrict in its second argument. Boolean functions `&&` and `||` from the Prelude are also strict in their first arguments and nonstrict in their second arguments.

```
(&&), (||) :: Bool -> Bool -> Bool
False && x = False  -- second argument not evaluated
True  && x = x

False || x = x
True  || x = True   -- second argument not evaluated
```

6.5 Currying and Partial Application

Consider the following two functions:

```
add :: (Int,Int) -> Int
add (x,y) = x + y

add' :: Int -> (Int -> Int)
add' x y  = x + y
```

These functions are closely related, but they are not identical.

For all integers `x` and `y`, `add (x,y) = add' x y`. But functions `add` and `add'` have different types.

Function `add` takes a 2-tuple `(Int,Int)` and returns an `Int`. Function `add'` takes an `Int` and returns a function of type `Int -> Int`.

What is the result of the application `add 3`? An error.

What is the result of the application `add' 3`? The result is a function that "adds 3 to its argument".

What is the result of the application `(add' 3) 4`? The result is the integer value `3 + 4`.

By convention, function application (denoted by the juxtaposition of a function and its argument) binds to the left. That is, `add' x y = ((add' x) y)`.

Hence, the higher-order functions in Haskell allow us to replace any function that takes a tuple argument by an equivalent function that takes a sequence of simple arguments corresponding to the components of the tuple. This process is called *currying*. It is named after American logician Haskell B. Curry, who first exploited the technique.

Function `add'` above is similar to the function `(+)` from the Prelude (i.e., the addition operator).

We sometimes speak of the function `(+)` as being *partially applied* in the expression `((+) 3)`. In this expression, the first argument of the function is "frozen in" and the resulting function can be passed as an argument, returned as a result, or applied to another argument.

Partially applied functions are very useful in conjunction with other higher-order functions.

For example, consider the partial applications of the relational comparison operator `(<)` and multiplication operator `(*)` in the function `doublePos3`. This function, which is equivalent to the function `doublePos` discussed in Section 6.2, doubles the positive integers in a list:

```
doublePos3 :: [Int] -> [Int]
doublePos3 xs = map' ((*) 2) (filter' ((<) 0) xs)
```

Related to the notion of currying is the *property of extensionality*. Two functions `f` and `g` are extensionally equal if `f x = g x` for all `x`.

Thus instead of writing the definition of `g` as

```
f, g :: a -> a
f x = some_expression

g x = f x
```

we can write the definition of `g` as simply:

```
g = f
```

6.6 Operator Sections

Expressions such as `((*) 2)` and `((<) 0)`, used in the definition of `doublePos3` in the previous subsection, can be a bit confusing because we normally use these operators in infix form. (In particular, it is difficult to remember the `((<) 0)` returns `True` for positive integers.)

Also, it would be helpful to be able to use the division operator to express a function that halves (i.e., divides by two) its operand? The function `((/) 2)` does not do it; it divides 2 by its operand.

We can use the function `flip` from the Prelude to state the halving operation. Function `flip` takes a function and two additional arguments and applies the argument function to the two arguments with their order reversed. (This is show below as `flip'` to avoid a name conflict.)

```
flip' :: (a -> b -> c) -> b -> a -> c  -- flip
flip' f x y = f y x
```

Thus we can express the halving operator with the expression `(flip (/) 2)`.

Because expressions such as `((<) 0)` and `(flip (/) 2)` are quite common in programs, Haskell provides a special, more compact and less confusing, syntax.

For some infix operator $\oplus$ and arbitrary expression e, expressions of the form `(`$e\oplus$`)` and `(`$\oplus e$`)` represent `((`$\oplus$`)` e`)` and `(flip (`$\oplus$`)` e`)`, respectively. Expressions of this form are called *operator sections*.

Examples:

`(1+)` is the successor function, which returns the value of its argument plus 1.

`(0<)` is a test for a positive integer.

`(/2)` is the halving function.

`(1.0/)` is the reciprocal function.

`(:[])` is the function that returns the singleton list containing the argument.

Suppose we want to sum the cubes of list of integers. We can express the function in the following way:

```
sumCubes :: [Int] -> Int
sumCubes xs = sum' (map' (^3) xs)
```

Above `^` is the exponentiation operator and `sum` is the list summation function defined in the Prelude as:

```
sum = foldl' (+) 0  -- sum
```

6.7 Combinators

The function `flip` in the previous subsection is an example of a useful type of function called a combinator.

A *combinator* is a function without any free variables. That is, on right side of a defining equation there are no variables or operator symbols that are not bound on the left side of the equation.

For historical reasons, `flip` is sometimes called the `C` combinator.

There are several other useful combinators in the Prelude.

The combinator `const` (shown below as `const'`) is the constant function constructor; it is a two-argument function that returns its first argument. For historical reasons, this combinator is sometimes called the `K` combinator.

```
const' :: a -> b -> a  -- const in Prelude
const' k x = k
```

Example: `(const 1)` takes any argument and returns the value 1.

What does `sum (map (const 1) xs)` do?

Function `id` (shown below as `id'`) is the identity function; it is a one-argument function that returns its argument unmodified. For historical reasons, this function is sometimes called the `I` combinator.

```
id' :: a -> a  -- id in Prelude
id' x = x
```

Combinators `fst` and `snd` (shown below as `fst'` and `snd'`) extract the first and second components, respectively, of 2-tuples.

```
fst' :: (a,b) -> a  -- fst in Prelude
fst' (x,_) = x

snd' :: (a,b) -> b  -- snd in Prelude
snd' (_,y) = y
```

Similarly, `fst3`, `snd3`, and `thd3` extract the first, second, and third components, respectively, of 3-tuples.

An interesting example that uses a combinator is the function `reverse` as defined in the Prelude (shown below as `reverse'`):

```
reverse' :: [a] -> [a]          -- reverse in Prelude
reverse' = foldlX (flip' (:)) []
```

Function `flip (:)` takes a list on the left and an element on the right. As this operation is folded through the list from the left it attaches each element as the new head of the list.

6.8 Functional Composition

The functional composition operator allows several "smaller" functions to be combined to form "larger" functions. In Haskell this combinator is denoted by the period (`.`) symbol and is defined in the Prelude as follows:

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Composition's default binding is from the right and its precedence is higher than all the operators we have discussed so far except function application itself.

Functional composition is an associative binary operation with the identity function `id` as its identity element:

```
f . (g . h) = (f . g) . h
id . f  = f . id
```

An advantage of the use of functional composition is that some expressions can be written more concisely. For example, the function

```
doit x = f1 (f2 (f3 (f4 x)))
```

can be written more concisely as

```
doit = f1 . f2 . f3 . f4
```

This form defines function `doit` as being equal to the composition of the other functions. It leaves the parameters of `doit` as implicit; `doit` has the same parameters as the composition.

If `doit` is defined as above, then we can use the function in expressions such as `map (doit) xs`. Of course, if this is the only use of the `doit` function, we can eliminate it completely and use the composition directly, e.g., `map (f1 . f2 . f3 . f4) xs`.

As an example, consider the function `count` that takes two arguments, an integer `n` and a list of lists, and returns the number of the lists from the second argument that are of length `n`. Note that all functions composed below are single-argument functions: `length`, `(filter (== n))`, `(map length)`.

```
count :: Int -> [[a]] -> Int
count n
    | n >= 0    = length . filter' (== n) . map' length
    | otherwise = const' 0   -- discard 2nd arg, return 0
```

Thus composition is a powerful form of "glue" that can be used to "stick" simpler functions together to build more powerful functions [14].

Remember the function `doublePos` that we discussed in Sections 6.2 and 6.5.

```
doublePos3 xs = map' ((*) 2) (filter' ((<) 0) xs)
```

Using composition and operator sections we can restate its definition as follows:

```
doublePos4 :: [Int] -> [Int]
doublePos4 = map' (2*) . filter' (0<)
```

Consider a function `last` to return the last element in a non-nil list and a function `init` to return the initial segment of a non-nil list (i.e., everything except the last element). These could quickly and concisely be written as follows:

```
last' = head . reverse'            -- last in Prelude
init' = reverse' . tail . reverse' -- init in Prelude
```

However, since these definitions are not very efficient, the Prelude implements functions `last` and `init` in a more direct and efficient way similar to the following:

```
last2 :: [a] -> a    -- last in Prelude
last2 [x]    = x
last2 (_:xs) = last2 xs

init2 :: [a] -> [a]  -- init in Prelude
init2 [x]    = []
init2 (x:xs) = x : init2 xs
```

The definitions for Prelude functions `any` and `all` are similar to the definitions show below; they take a predicate and a list and apply the predicate to each element of the list, returning `True` when any and all, respectively, of the individual tests evaluate to `True`.

```
any', all' :: (a -> Bool) -> [a] -> Bool
any' p = or' . map' p   -- any in Prelude
all' p = and' . map' p  -- all in Prelude
```

The functions `elem` and `notElem` test for an object being an element of a list and not an element, respectively. They are defined in the Prelude similarly to the following:

```
elem', notElem' :: Eq a => a -> [a] -> Bool
elem'    = any' . (==)  -- elem in Prelude
notElem' = all' . (/=)  -- notElem in Prelude
```

These are a bit more difficult to understand since `any`, `all`, `==`, and `/=` are two-argument functions. Note that expression `elem x xs` would be evaluated as follows:

```
   elem' x xs
⟹   { expand elem' }
   (any' . (==)) x xs
⟹   { expand composition }
   any' ((==) x) xs
```

The composition operator binds the first argument with `(==)` to construct the first argument to `any'`. The second argument of `any'` is the second argument of `elem'`.

6.9 Lambda Expressions

Remember the function `squareAll2` we examined in the section on maps (Section 6.1):

```
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x
```

We introduced the local function definition `sq` to denote the function to be passed to `map`. It seems to be a waste of effort to introduce a new symbol for a simple function that is only used in one place in an expression. Would it not be better, somehow, to just give the defining expression itself in the argument position?

Haskell provides a mechanism to do just that, an anonymous function definition. For historical reasons, these nameless functions are called *lambda expressions*. They begin with a backslash `\` and have the syntax:

`\` *atomicPatterns* `->` *expression*

For example, the squaring function (`sq`) could be replaced by a lambda expression as `(\x -> x * x)`. The pattern `x` represents the single argument for this anonymous function and the expression `x * x` is its result.

A lambda expression to average two numbers is `(\x y -> (x+y)/2)`.

An interesting example that uses a lambda expression is the function `length` as defined in the Prelude—similar to `length'` below. (Note that this uses the optimized function `foldl'` from the `Data.List` module.)

```
length' :: [a] -> Int   -- length in Prelude
length'  = foldl' (\n _ -> n+1) 0
```

The anonymous function `(\n _ -> n+1)` takes an integer "counter" and a polymorphic value and returns the "counter" incremented by one. As this function is folded through the list from the left, this function counts each element of the second argument.

6.10 List-Breaking Operations

In Section 5.5.2 we looked at the list-breaking functions `take` and `drop`. The Prelude also includes several higher-order list-breaking functions that take two arguments, a predicate that determines where the list is to be broken and the list to be broken.

The Prelude includes the functions `span`, `break`, `takeWhile`, `dropWhile`, `takeUntil`, and `dropUntil`.

Here we look at Prelude functions `takeWhile` and `dropWhile`. As you would expect, function `takeWhile` "takes" elements from the beginning of the list "while" the elements satisfy the predicate and `dropWhile` "drops" elements from the beginning of the list "while" the elements satisfy the predicate. The Prelude definitions are similar to the following:

```
takeWhile':: (a -> Bool) -> [a] -> [a]  -- takeWhile
takeWhile'  p []  = []
takeWhile'  p (x:xs)
    | p x         = x : takeWhile' p xs
    | otherwise   = []

dropWhile' :: (a -> Bool) -> [a] -> [a]  -- dropWhile
dropWhile'  p []  = []
dropWhile'  p xs@(x:xs')
    | p x         = dropWhile' p xs'
    | otherwise   = xs
```

Note the use of the pattern `xs@(x:xs')` . This pattern matches a non-nil list with `x` and `xs'` binding to the head and tail, respectively, as usual. Variable `xs` binds to the entire list.

As an example, suppose we want to remove the leading blanks from a string. We can do that with the expression `dropWhile ((==) ' ')`.

As with `take` and `drop`, the above functions can also be related by a "law". For all finite lists `xs` and predicates `p` on the same type:

```
takeWhile p xs ++ dropWhile p xs = xs
```

6.11 List-Combining Operations

In Section 5.5.3 we also looked at the function `zip`, which takes two lists and returns a list of pairs of the corresponding elements. Function `zip` applies an operation, in this case *tuple-construction*, to the corresponding elements of two lists.

We can generalize this pattern of computation with the function `zipWith` in which the operation is an argument to the function.

```
zipWith' :: (a->b->c) -> [a]->[b]->[c] -- zipWith
zipWith' z  (x:xs)  (y:ys) = z x y : zipWith' z xs ys
zipWith' _  _     _        = []
```

Using a lambda expression to state the tuple-forming operation, the Prelude defines `zip` in terms of `zipWith`:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' = zipWith' (\x y -> (x,y))
```

Or can be written more simply as:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' = zipWith' (,)
```

The `zipWith` function also enables us to define operations such as the scalar product of two vectors in a concise way.

```
sp :: Num a => [a] -> [a] -> a
sp xs ys = sum' (zipWith' (*) xs ys)
```

The Prelude includes versions of `zipWith` that take up to seven input lists: `zipWith3` $\cdots$ `zipWith7`.

6.12 Rational Arithmetic Revisited

Remember the rational number arithmetic package developed in Section 5.6. In that package we defined a function `eqRat` to compare two rational numbers for equality using the appropriate set of integer comparisons. We also noted that the other comparison operations can be defined similarly.

Because the comparison operations are similar, they are good candidates for use of higher-order function. We can abstract out the common pattern of comparisons into a function that takes the corresponding integer comparison as an argument.

To compare two rational numbers, we express their values in terms of a common denominator and then compare the numerators using the integer comparisons. We can thus abstract the comparison into a higher-order function `compareRat` that takes an appropriate integer relation and the two rational numbers.

```
compareRat :: (Int -> Int -> Bool) -> Rat -> Rat -> Bool
compareRat r (a,b) (c,d)
    |  b == 0    = errRat (a,0)
    |  d == 0    = errRat (c,0)
    |  otherwise = r (a*d) (b*c)
```

Then we can define the rational number comparisons in terms of `compareRat`. (Note that this redefines function `eqRat` from the chapter 5 package.)

```
eqRat,neqRat,ltRat,leqRat,gtRat,geqRat :: Rat -> Rat -> Bool
eqRat   = compareRat (==)
neqRat  = compareRat (/=)
ltRat   = compareRat (<)
leqRat  = compareRat (<=)
gtRat   = compareRat (>)
geqRat  = compareRat (>=)
```

6.13 Cosequential Processing

This example deals with the coordinated processing of two ordered sequences, that is, *cosequential processing*. An example of a cosequential processing function is a function to merge two ascending lists of some type into a single ascending list.

Note: A list is *ascending* if every element is `<=` all of its successors in the list. Successor means an element that occurs later in the list (i.e., away from the head). A list is *increasing* if every element is `<` its successors. Similarly, a list is *descending* or *decreasing* if every element is `>=` or `>`, respectively, its successors.

Here we approach the problem of merging two lists in a general way.

Using the typical recursive technique, a general merge function must examine the "heads" of its two input lists, compute an appropriate value for the initial segment of the result list, and then repeat the process recursively on the remaining portions of the input lists to generate the final segment of the output list. For the recursive call, at least one of the input lists must become shorter.

What cases must we consider?

Of course, for both input lists, we must deal with nil and non-nil lists. Thus there are four general cases to consider for the merge.

Moreover, for the case of two non-nil lists, we must consider the relative ordering of the head elements of the respective lists, that is, whether the head of the first list is less than, equal to, or greater than the head of the second list. Thus there are three subcases to consider.

Consequently, we have six cases to consider in all. Thus the general structure of a merge function is as follows. (This is not necessarily a valid Haskell script.)

```
gmerge :: [a] -> [b] -> [c]
gmerge []        []        = e1
gmerge []        bs@(y:ys) = e2 bs
gmerge as@(x:xs) []        = e3 as
gmerge as@(x:xs) bs@(y:ys)
       | keya x < keyb y   = f4 x y ++ gmerge (g4 as) (h4 bs)
       | keya x == keyb y  = f5 x y ++ gmerge (g5 as) (h5 bs)
       | keya x > keyb y   = f6 x y ++ gmerge (g6 as) (h6 bs)
```

In this general function definition:

- `keya` and `keyb` represent the "functions" to extract the values to be compared from the first and second arguments, respectively. This general function definition assumes that the two arguments are ordered in the same way by the values of these keys.

- `e1`, `e2`, `e3` represent the "functions" to compute the values to be returned in the respective base cases.
- `f4`, `f5`, and `f6` represent the "functions" to compute the initial segments of values to be returned in the respective recursive cases.
- `g4`, `g5`, and `g6` represent the "functions" to compute the first arguments for the recursive calls in the respective recursive cases.
- `h4`, `h5`, and `h6` represent the "functions" to compute the second arguments for the recursive calls in the respective recursive cases.

This general function can be specialized to handle a specific situation. We can replace the applications of the various "functions" with the needed expressions and, in many cases, replace `++` by the more efficient cons.

For example, suppose we want to implement the specific function we mentioned above, that is, a function `merge1` to merge two ascending lists of some type into an ascending list.

```
merge1 :: Ord a => [a] -> [a] -> [a]
merge1 []        []        = []
merge1 []        bs@(y:ys) = bs
merge1 as@(x:xs) []        = as
merge1 as@(x:xs) bs@(y:ys)
       | x < y             = x : merge1 xs bs
       | x == y            = x : merge1 xs bs
       | x > y             = y : merge1 as ys
```

We note that case 1 can combined with case 2 and that case 4 can be combined with case 5. Further, the top-to-bottom order of testing can be exploited to make the definition more concise. Doing these transformations we get `merge2`.

```
merge2 :: Ord a => [a] -> [a] -> [a]
merge2 []        bs        = bs
merge2 as        []        = as
merge2 as@(x:xs) bs@(y:ys)
       | x <= y            = x : merge2 xs bs
       | x > y             = y : merge2 as ys
```

Other specializations of the general merge function will give slightly different function definitions.

Now let us consider the general merge function again. We could, of course, define it directly as a higher-order function.

```
gmerge :: Ord d =>
    (a -> d) ->        -- keya
    (b -> d) ->        -- keyb
    [c] ->             -- e1
    ([b] -> [c]) ->    -- e2
    ([a] -> [c]) ->    -- e3
    (a -> b -> [c]) -> -- f4
    (a -> b -> [c]) -> -- f5
    (a -> b -> [c]) -> -- f6
    ([a] -> [a]) ->    -- g4
    ([a] -> [a]) ->    -- g5
    ([a] -> [a]) ->    -- g6
    ([b] -> [b]) ->    -- h4
    ([b] -> [b]) ->    -- h5
    ([b] -> [b]) ->    -- h6
    [a] -> [b] -> [c]
gmerge keya keyb e1 e2 e3 f4 f5 f6 g4 g5 g6 h4 h5 h6
    = gmerge'
      where
        gmerge'  []  []        = e1
        gmerge'  []  bs@(y:ys) = e2 bs
        gmerge'  as@(x:xs)  [] = e3 as
        gmerge'  as@(x:xs)  bs@(y:ys)
            | keya x <  keyb y = f4 x y ++ gmerge' (g4 as) (h4 bs)
            | keya x == keyb y = f5 x y ++ gmerge' (g5 as) (h5 bs)
            | keya x >  keyb y = f6 x y ++ gmerge' (g6 as) (h6 bs)
```

Thus we can define `merge1` using various combinators as follows:

```
merge1' :: Ord a => [a] -> [a] -> [a]
merge1' = gmerge  id id   -- keya, keyb
   [] id id               -- e1, e2, e3
   (const . (:[]))        -- f4
   (const . (:[]))        -- f5
   (flip (const . (:[]))) -- f6
   tail tail id           -- g4, g5, g6
   id id tail             -- h4, h5, h6
```

The only "tricky" arguments above are for `f4`, `f5`, and `f6`. These two-argument functions must return their first, first, and second arguments, respectively, as singleton lists.

6.14 Exercises

1. Suppose you need a Haskell function `times` that takes a list of integers and returns the product of the elements (e.g, `times [2,3,4]` returns `24`). Write the following versions as Haskell functions.

   (a) A version that uses the Prelude function `foldr`.

   (b) A version that uses backward recursion to compute the product. (Do not use the list-folding Prelude functions such as `foldr` or `product`.)

   (c) A version which uses forward recursion to compute the product. (Hint: use a tail-recursive auxiliary function with an accumulating parameter.)

2. For each of the following specifications, write a Haskell function that has the given arguments and result. Use the functions `map`, `filter`, and `foldr` as appropriate.

   (a) Function `numof` takes a value and a list and returns the number of occurrences of the value in the list.

   (b) Function `ellen` takes a list of character strings and returns a list of the lengths of the corresponding strings.

   (c) Function `ssp` takes a list of integers and returns the sum of the squares of the positive elements of the list.

3. Write a Haskell function `map2` that takes a list of functions and a list of values and returns the list of results of applying each function in the first list to the corresponding value in the second list.

4. Write a Haskell function `fmap` that takes a value and a list of functions and returns the list of results from applying each function to the argument value. (For example, `fmap 3 [((*) 2), ((+) 2)]` yields `[6,5]`.)

5. A list `s` is a *prefix* of a list `t` if there is some list `u` (perhaps nil) such that `s ++ u == t`. For example, the prefixes of string `"abc"` are `""`, `"a"`, `"ab"`, `"abc"`.

   A list `s` is a *suffix* of a list `t` if there is some list `u` (perhaps nil) such that `u ++ s == t`. For example, the suffixes of `"abc"` are `"abc"`, `"bc"`, `"c"`, and `""`.

   A list `s` is a *segment* of a list `t` if there are some (perhaps nil) lists `u` and `v` such that `u ++ s ++ v == t`. For example, the segments of string `"abc"` consist of the prefixes and the suffixes plus `"b"`.

   Write the following Haskell functions. You may use functions appearing early in the list to implement later ones.

   (a) Write a function `prefix` such that `prefix xs ys` returns `True` if `xs` is a prefix of `ys` and returns `False` otherwise.

(b) Write a function `suffixes` such that `suffixes xs` returns the list of all suffixes of list `xs`. (Hint: Generate them in the order given in the example of `"abc"` above.)

(c) Define a function `indexes` such that `indexes xs ys` returns a list of all the positions at which list `xs` appears in list `ys`. Consider the first character of `ys` as being at position 1. For example, `indexes "ab" "abaabbab"` returns `[1,4,7]`. (Hint: Remember functions like `map`, `filter`, `zip`, and the functions you just defined?)

(d) Define a function `sublist` such that `sublist xs ys` returns `True` if list `xs` appears as a segment of list `ys` and returns `False` otherwise.

6. Assume that the following Haskell type synonyms have been defined:

```
type Word  = String   -- word, characters left-to-right
type Line  = [Word]   -- line, words left-to-right
type Page  = [Line]   -- page, lines top-to-bottom
type Doc   = [Page]   -- document, pages front-to-back
```

Further assume that values of type `Word` do not contain any space characters. Implement the following Haskell text-handling functions:

(a) `npages` that takes a `Doc` and returns the number of `Page`s in the document.

(b) `nlines` that takes a `Doc` and returns the number of `Line`s in the document.

(c) `nwords` that takes a `Doc` and returns the number of `Word`s in the document.

(d) `nchars` that takes a `Doc` and returns the number of `Char`s in the document (not including spaces of course).

(e) `deblank` that takes a `Doc` and returns the `Doc` with all blank lines removed. A blank line is a line that contains no words.

(f) `linetext` that takes a `Line` and returns the line as a `String` with the words appended together in left-to-right order separated by space characters and with a newline character '`\n`' appended to the right end of the line. (For example, `linetext ["Robert", "Khayat"]` yields `"Robert Khayat\n"`.)

(g) `pagetext` that takes a `Page` and returns the page as a `String`—applies `linetext` to its component lines and appends the result in a top-to-bottom order.

(h) `doctext` that takes a `Doc` and returns the document as a `String`—applies `pagetext` to its component lines and appends the result in a top-to-bottom order.

(i) `wordeq` that takes a two `Doc`s and returns `True` if the two documents are *word equivalent* and `False` otherwise. Two documents are word equivalent if they contain exactly the same words in exactly the same order regardless of page and line structure. For example, `[[["Robert"],["Khayat"]]]` is word equivalent to `[[["Robert","Khayat"]]]`.

7 MORE LIST NOTATION

7.1 Sequences

Haskell provides a compact notation for expressing arithmetic sequences.

An arithmetic sequence (or progression) is a sequence of elements from an enumerated type (i.e., a member of class `Enum`) such that consecutive elements have a fixed difference. `Int`, `Integer`, `Float`, `Double`, and `Char` are all predefined members of this class.

- `[m..n]` produces the list of elements from `m` up to `n` in steps of one if `m <= n`; it produces the nil list otherwise.

  Example: `[1..5]` $\Longrightarrow$ `[1,2,3,4,5]`
  `[5..1]` $\Longrightarrow$ `[]`

  This feature is implemented with Prelude function `enumFromTo` applied as `enumFromTo m n`.

- `[m,m'..n]` produces the list of elements from `m` in steps of `m'-m`. Note that if `m' > m` then the list is increasing up to `n`; if `m' < m` then it is decreasing.

  Example: `[1,3..9]` $\Longrightarrow$ `[1,3,5,7,9]`
  `[9,8..5]` $\Longrightarrow$ `[9,8,7,6,5]`
  `[9,8..11]` $\Longrightarrow$ `[]`

  This feature is implemented with Prelude function `enumFromThenTo` applied as `enumFromThenTo m' m n`.

- `[m..]` and `[m,m'..]` produce potentially infinite lists beginning with `m` and having steps 1 and `m'-m` respectively.

  These features are implemented with Prelude functions `enumFrom` applied as `enumFrom m` and `enumFromThen` applied as `enumFromThen m m'`.

Of course, we can provide our own functions for sequences. Consider the following function to generate a geometric sequence.

A geometric sequence (or progression) is a sequence of elements from an ordered, numeric type (i.e., a member of both classes `Ord` and `Num`) such that consecutive elements have a fixed ratio.

```
geometric :: (Ord a, Num a) => a -> a -> a -> [a]
geometric r m n | m > n     = []
                | otherwise = m : geometric r (m*r) n
```

Example: `geometric 2 1 10` $\Longrightarrow$ `[1,2,4,8]`

7.2 List Comprehensions

The *list comprehension* is another powerful and compact notation for describing lists. A list comprehension has the form

`[` *expression* `|` *qualifiers* `]`

where *expression* is any Haskell expression.

The *expression* and the *qualifiers* in a comprehension may contain variables that are local to the comprehension. The values of these variables are bound by the *qualifiers*. For each group of values bound by the qualifiers, the comprehension generates an element of the list whose value is the *expression* with the values substituted for the local variables.

There are three kinds of *qualifiers* that can be used in Haskell: generators, filters, and local definitions.

Generators:

A *generator* is qualifier of the form *pat* `<-` *exp* where *exp* is a list-valued expression. The generator extracts each element of *exp* that matches the pattern *pat* in the order that the elements appear in the list; elements that do not match the pattern are skipped.

Example: `[ n*n | n<-[1..5]]` $\Longrightarrow$ `[1,4,9,16,25]`

Filters:

A Boolean-valued expression may also be used as a *qualifier* in a list comprehension. These expressions act as *filters*; only values that make the expression `True` are used to form elements of the list comprehension.

Example: `[ n*n | even n ]` $\Longrightarrow$ `(if even n then [n*n] else [])`
Above variable `n` is global to this expression, not local to the comprehension.

Local Definitions:

A qualifier of the form `let` *pat* `=` *expr* introduces a local definition into the list comprehension.

Example: `[ n*n | let n = 2 ]` $\Longrightarrow$ `[4]`

Note: `n == 2` is a filter; `n = 2` is a local definition.

The real power of list comprehensions come from using several qualifiers separated by commas on the right side of the vertical bar `|`.

- Generators appearing later in the list of qualifiers vary more quickly than those that appear earlier. Speaking operationally, the generation "loop" for the later generator is nested within the "loop" for the earlier.

  Example: `[ (m,n) | m<-[1..3], n<-[4,5] ]`
  $\Longrightarrow$ `[ (1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]`

- Qualifiers appearing later in the list of qualifiers may use values generated by qualifiers appearing earlier, but not vice versa.

  Example: `[ n*n | n<-[1..10], even n ]` $\Longrightarrow$ `[4,16,36,64,100]`

  Example: `[ (m,n) | m<-[1..3], n<-[1..m] ]`
  $\Longrightarrow$ `[ (1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]`

- The generated values may or may not be used in the *expression*.

  Example: `[ 27 | n<-[1..3]]` $\Longrightarrow$ `[27,27,27]`

  Example: `[ x | x<-[1..3], y<-[1..2]]` $\Longrightarrow$ `[1,1,2,2,3,3]`

7.2.1 Example: Strings of spaces

Consider a function `spaces` that takes a number and generates a string with that many spaces.

```
spaces :: Int -> String
spaces n = [ ' ' | i<-[1..n]]
```

Note that when `n < 1` the result is the empty string.

7.2.2 Example: Prime number test

Consider a Boolean function `isPrime` that takes a nonzero natural number and determines whether the number is prime. (As I am sure you remember, a prime number is a natural number whose only natural number factors are 1 and itself.)

```
isPrime :: Int -> Bool
isPrime n | n > 1 = (factors n == [])
    where factors m = [ x | x<-[2..(m-1)], m `mod` x == 0 ]
isPrime   _       =  False
```

7.2.3 Example: Squares of primes

Consider a function `sqPrimes` that takes two natural numbers and returns the list of squares of the prime numbers in the inclusive range from the first up to the second.

```
sqPrimes :: Int -> Int -> [Int]
sqPrimes m n = [ x*x | x<-[m..n], isPrime x ]
```

Alternatively, this function could be defined using `map` and `filter` as follows:

```
sqPrimes' :: Int -> Int -> [Int]
sqPrimes' m n = map (\x -> x*x) (filter isPrime [m..n])
```

7.2.4 Example: Doubling positive elements

We can use a list comprehension to define (our, by now, old and dear friend) the function `doublePos`, which doubles the positive integers in a list.

```
doublePos5 :: [Int] -> [Int]
doublePos5 xs = [ 2*x | x<-xs, 0 < x ]
```

7.2.5 Example: Concatenate a list of lists of lists

Consider a program `superConcat` that takes a list of lists of lists and concatenates the elements into a single list.

```
superConcat :: [[[a]]] -> [a]
superConcat xsss = [ x | xss<-xsss, xs<-xss, x<-xs ]
```

Alternatively, this function could be defined using Prelude functions `concat` and `map` and functional composition as follows:

```
superConcat' :: [[[a]]] -> [a]
superConcat' = concat . map concat
```

7.2.6 Example: First occurrence in a list

Reference: This example is based on the one given in Section 3.3, page 58, of the Bird and Wadler textbook [2].

Consider a function `position` that takes a list and a value of the same type. If the value occurs in the list, `position` returns the position of the value's first occurrence; if the value does not occur in the list, `position` returns 0.

Strategy: *Solve a more general problem first, then use it to get the specific solution desired.*

In this problem, we generalize the problem to finding *all* occurrences of a value in a list. This more general problem is actually easier to solve.

```
positions :: Eq a => [a] -> a -> [Int]
positions xs x = [ i | (i,y)<-zip [1..length xs] xs, x == y]
```

Note the use of `zip` to pair an element of the list with its position within the list and the filter to remove those pairs not involving the value `x`. The "zipper" functions can be very useful within list comprehensions.

Now that we have the positions of all the occurrences, we can use `head` to get the first occurrence. Of course, we need to be careful that we return 0 when there are no occurrences of `x` in `xs`.

```
position :: Eq a => [a] -> a -> Int
position xs x = head ( positions xs x ++ [0] )
```

Because of lazy evaluation, this implementation of `position` is not as inefficient as it first appears. The function `positions` will, in actuality, only generate the head element of its output list.

Also because of lazy evaluation, the upper bound `length xs` can be left off the generator in `positions`. In fact, the function is more efficient to do so.

7.3 Exercises

1. Show the list (or string) yielded by each of the following Haskell list expressions. Display it using fully specified list bracket notation, e.g., expression `[1..5]` yields `[1,2,3,4,5]`.

   (a) `[7..11]`

   (b) `[11..7]`

   (c) `[3,6..12]`

   (d) `[12,9..2]`

   (e) `[ n*n | n <- [1..10], even n ]`

   (f) `[ 7 | n <- [1..4] ]`

   (g) `[ x | (x:xs) <- ["Did", "you", "study?"] ]`

   (h) `[ (x,y) | x <- [1..3], y <- [4,7] ]`

   (i) `[ (m,n) | m <- [1..3], n <- [1..m] ]`

   (j) `take 3 [ [1..n] | n <- [1..]  ]`

2. Translate the following expressions into expressions that use list comprehensions. For example, `map (*2) xs` could be translated to `[ x*2 | x <- xs ]`.

   (a) `map (\x -> 2*x-1) xs`

   (b) `filter p xs`

   (c) `map (^2) (filter even [1..5])`

   (d) `foldr (++) [] xss`

   (e) `map snd (filter (p .  fst) (zip xs [1..]))`

8 MORE ON DATA TYPES

8.1 User-Defined Types

In addition to the built-in data types we have discussed, Haskell also allows the definition of new data types using declarations of the form:

$$\texttt{data}\ \mathit{Datatype}\ a_1\ a_2\ \cdots\ a_n = \mathit{constr}_1 \mid \mathit{constr}_2 \mid \cdots \mid \mathit{constr}_m$$

where:

- *Datatype* is the name of a new type constructor of *arity* n ($n \geq 0$). As with the built-in types, the name of the data type must begin with a capital letter.
- $a_1\ a_2\ \cdots\ a_n$ are distinct type variables representing the n parameters of the data type.
- constr_1, constr_2, $\cdots$, constr_m ($m \geq 1$) are the data constructors that describe the ways in which the elements of the new data type are constructed.

For example, consider a new data type `Color` whose possible values are the colors on the flag of the USA. The names of the data constructors (the color constants in this case) must also begin with capital letters.

```
data Color = Red | White | Blue
             deriving (Show, Eq)
```

`Color` is an example of an *enumerated type*, a type that consists of a finite sequence of *nullary* (i.e., the arity–number of paramters–is zero) data constructors.

The optional `deriving` clause declares that these new types are automatically added as instances of the classes listed. In this case, `Show` and `Eq` enable objects of type `Color` to be converted to a `String` and compared for equality, respectively. Depending upon the nature of the data type, it is possible derive instances of standard classes such as `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`.

We can use the type and data constructor names defined with `data` in declarations, patterns, and expressions in the same way that the built-in types can be used.

```
isRed :: Color -> Bool
isRed Red = True
isRed _   = False
```

Data constructors can also have associated values. For example, the constructor `Grayscale` below takes an integer value.

```
data Color' = Red' | Blue' | Grayscale Int
              deriving (Show, Eq)
```

Constructor `Grayscale` implicitly defines a constructor function with the type `Int -> Color'`.

Consider a data type `Point` which has a type parameter. The following defines a polymorphic type; both of the values associated witha the constructor `Pt` must be of type `a`. Constructor `Pt` implicitly defines a constructor function of type `a -> a -> Point a`.

```
data Point a = Pt a a
               deriving (Show, Eq)
```

A type like `Point` is often called a *tuple type* since it is essentially a Cartesian product of other types. Types like `Color` and `Color'`, which have multiple data constructors, are called (disjoint) *union* types.

As another example, consider a polymorphic set data type that represents a set as a list of values as follows. Note that the name `Set` is used both as the type constructor and a data constructor—in general, do not use a symbol in multiple ways.

```
data Set a = Set [a]
             deriving (Show, Eq)
```

Now we can write a function `makeSet` to transform a list into a `Set`. This function uses the funciton `nub` from the `Data.List` module to remove duplicates from a list.

```
makeSet :: Eq a => [a] -> Set a
makeSet xs = Set (nub xs)
```

As we have seen previously (Section 5.1), programmers can also define type synonyms. As in user-defined types, synonyms may have parameters. For example, the following might define a matrix of some polymorphic type as a list of lists of that type.

```
type Matrix a = [[a]]
```

We can also use special types to encode error conditions. For example, suppose we want an integer division operation that returns an error message if there is an attempt to divide by 0 and returns the quotient otherwise. We can define and use a union type `Result` as follows:

```
data Result a = Ok a | Err String
                deriving (Show, Eq)

divide :: Int -> Int -> Result Int
divide _  0 = Err "Divide by zero"
divide x  y = Ok (x `div` y)
```

Then we can use this operation in the definition of another function `f` that returns the maximum `Int` value `maxBound` when a division by 0 occurs.

```
f :: Int -> Int -> Int
f x y  = return (divide x y)
         where return (Ok z)  = z
               return (Err s) = maxBound
```

The auxiliary function `return` can be avoided by using the Haskell `case` expression as follows:

```
f' x y = case divide x y of
              Ok z  -> z
              Err s -> maxBound
```

This case expression evaluates the expression `divide x y`, matches its result against the patterns of the alternatives, and returns the right-hand-side of the first matching pattern.

8.2 Recursive Data Types

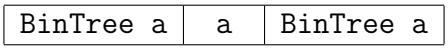
Types can also be recursive.

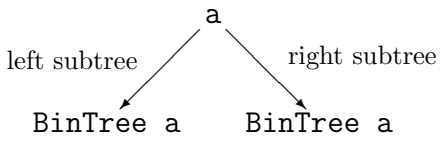
For example, consider the user-defined type `BinTree`, which defines a binary tree with values of a polymorphic type.

```
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
                 deriving (Show, Eq)
```

This data type represents a binary tree with a value in each node. The tree is either "empty" (denoted by `Empty`) or it is a "node" (denoted by `Node`) that consists of a value of type `a` and "left" and "right" subtrees. Each of the subtrees must themselves be objects of type `BinTree`.

Thus a binary tree is represented as a three-part "record" in which the left and right subtrees are represented as nested binary trees. There are no explicit "pointers".





Consider a function `flatten` to return the list of the values in binary tree in the order corresponding to a left-to-right in-order traversal.

Example: `flatten (Node (Node Empty 3 Empty) 5 (Node (Node Empty 7 Empty) 1 Empty))` yields `[3,5,7,1]`.

```
flatten :: BinTree a -> [a]
flatten Empty        = []
flatten (Node l v r) = flatten l ++ [v] ++ flatten r
```

The second leg of `flatten` requires two recursive calls. However, as long as the input tree is finite, each recursive call receives a tree that is simpler (e.g., shorter) than the input. Thus all recursions eventually terminate when `flatten` is called with an `Empty` tree.

Function `flatten` can be rendered more efficiently using an accumulating parameter and cons as in the following:

```
flatten' :: BinTree a -> [a]
flatten' t = inorder t []
             where inorder Empty xs        = xs
                   inorder (Node l v r) xs =
                       inorder l (v : inorder r xs)
```

Auxiliary function `inorder` builds up the list of values from the right using cons.

To extend the example further, consider a function `treeFold` that folds an associative operation `op` with identity element `i` through a left-to-right in-order traversal of the tree.

```
treeFold :: (a -> a -> a) -> a -> BinTree a -> a
treeFold op i Empty        = i
treeFold op i (Node l v r) = op (op (treeFold op i l) v)
                                (treeFold op i r)
```

Now let's consider a slightly different formulation of a binary tree: a tree in which values are only stored at the leaves.

```
data Tree a = Leaf a | Tree a :^: Tree a
              deriving (Show, Eq)
```

This definition introduces the constructor function name `Leaf` as the constructor for leaves and the infix construction operator "`:^:`" as the constructor for internal nodes of the tree. (A constructor operator symbol must begin with a colon.)

These constructors allow such trees to be defined conveniently. For example, the tree

```
((Leaf 1 :^: Leaf 2) :^: (Leaf 3 :^: Leaf 4))
```

generates a complete binary tree with height 3 and the integers 1, 2, 3, and 4 at the leaves.

Suppose we want a function `fringe`, similar to function `flatten` above, that displays the leaves in a left-to-right order. We can write this as:

```
fringe :: Tree a -> [a]
fringe (Leaf v)  = [v]
fringe (l :^: r) = fringe l ++ fringe r
```

As with `flatten` and `flatten'` above, function `fringe` can also be rendered more efficiently using an accumulating parameter as in the following:

```
fringe' :: Tree a -> [a]
fringe' t = leaves t []
            where leaves (Leaf v)  = ((:) v)
                  leaves (l :^: r) = leaves l . leaves r
```

Auxiliary function `leaves` builds up the list of leaves from the right using cons.

8.3 Exercises

1. For trees of type `Tree` defined in Section 8.2, implement a tree-folding function similar to `treeFold`.

2. For trees of type `BinTree` defined in Section 8.2, implement a version of `treeFold` that uses an accumulating parameter. (Hint: `foldl`.)

3. In a binary search tree all values in the left subtree of a node are less than the value at the node and all values in the right subtree are greater than the value at the node. Given binary search trees of type `BinTree` defined in Section 8.2, implement the following Haskell functions:

   `makeTree` that takes a list and returns a perfectly balanced (i.e., minimal height) `BinTree` such that `flatten (makeTree xs) = sort xs`. Prelude function `sort` returns its argument rearranged into ascending order.

   `insertTree` that takes an element and a `BinTree` and returns the `BinTree` with the element inserted at an appropriate position.

   `elemTree` that takes an element and a `BinTree` and returns `True` if the element is in the tree and `False` otherwise.

   `heightTree` that takes a `BinTree` and returns its height. Assume that height means the number of levels in the tree. (A tree consisting of exactly one node has a height of 1.)

   `mirrorTree` that takes a `BinTree` and returns its mirror image. That is, it takes a tree and returns the tree with the left and right subtrees of every node swapped.

   `mapTree` that takes a function and a `BinTree` and returns the `BinTree` of the same shape except each node's value is computed by applying the function to the corresponding value in the input tree.

   `showTree` that takes a `BinTree` and displays the tree in a parenthesized, left-to-right, in-order traversal form. (That is, the traversal of a tree is enclosed in a pair of parentheses, with the traversal of the left subtree followed by the traversal of the right subtree.)

   Extend the package to support both insertion and deletion of elements. Keep the tree balanced using a technique such the AVL balancing algorithm.

4. Implement the package of functions described in the previous exercise for the data type `Tree` defined in Section 8.2.

5. Each node of a general (i.e., multiway) tree consists of a label and a list of (zero or more) subtrees (each a general tree). We can define a general tree data type in Haskell as follows:

```
data Gtree a = Node a [Gtree a]
```

For example, tree `(Node 0 [ ])` consists of a single node with label 0; a more complex tree, `(Node 0 [ Node 1 [ ], Node 3 [ ], Node 7 [ ] ])`, consists of root node with three single-node subtrees.

Implement a "map" function for general trees, i.e., write Haskell function

```
mapGtree :: (a -> b) -> Gtree a -> Gtree b
```

that takes a function and a `Gtree` and returns the `Gtree` of the same shape such that each label is generated by applying the function to the corresponding label in the input tree.

6. We can introduce a new Haskell type for the natural numbers (i.e., nonnegative integers) with the statement

```
data Nat = Zero | Succ Nat
```

where the constructor `Zero` represents the value 0 and constructor `Succ` represents the "successor function" from mathematics. Thus `(Succ Zero)` denotes 1, `(Succ (Succ Zero))` denotes 2, and so forth. Implement the following Haskell functions.

`intToNat` that takes a nonnegative `Int` and returns the equivalent `Nat`, for example, `intToNat 2` returns `(Succ (Succ Zero))`.

`natToInt` that takes a `Nat` and returns the equivalent value of type `Int`, for example, `natToInt (Succ Zero)` returns 1.

`addNat` that takes two `Nat`s and returns their sum as a `Nat`. This function cannot use integer addition.

`mulNat` that takes two `Nat`s and returns their product as a `Nat`. This function cannot use integer multiplication or addition.

`compNat` that takes two `Nat`s and returns the value `-1` if the first is less than the second, `0` if they are equal, and `1` if the first is greater than the second. This function cannot use the integer comparison operators.

7. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Att (Seq a) a
```

`Nil` represents the empty sequence. `Att xz y` represents the sequence in which *last element* `y` is "attached" at the right end of the *initial sequence* `xz`.

Note that `Att` is similar to the ordinary "cons" for Haskell lists except that elements are attached at the opposite end of the sequences. `(Att (Att (Att Nil 1) 2) 3)` represents the same sequence as the ordinary list `(1:(2:(3:[])))`.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists.

`lastSeq` takes a nonempty `Seq` and returns its last (i.e., rightmost) element.

`initialSeq` takes a nonempty `Seq` and returns its initial sequence (i.e., sequence remaining after the last element removed).

`lenSeq` takes a `Seq` and returns the number of elements that it contains.

`headSeq` takes a nonempty `Seq` and returns its head (i.e., leftmost) element.

`tailSeq` takes a nonempty `Seq` and returns the `Seq` remaining after the head element is removed.

`conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

`appSeq` takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

`revSeq` takes a `Seq` and returns the `Seq` with the same elements in reverse order.

`mapSeq` takes a function and a `Seq` and returns the `Seq` resulting from applying the function to each element of the sequence in turn.

`filterSeq` that takes a predicate and a `Seq` and returns the `Seq` containing only those elements that satisfy the predicate.

`listToSeq` takes an ordinary Haskell list and returns the `Seq` with the same values in the same order (e.g., `headSeq (listToSeq xs) = head xs` for nonempty `xs`.)

`seqToList` takes a `Seq` and returns the ordinary Haskell list with the same values in the same order (e.g., `head (seqToList xz) = headSeq xz` for nonempty `xz`.)

8. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Unit a | Cat (Seq a) (Seq a)
```

The constructor `Nil` represents the empty sequence; `Unit` represents a single-element sequence; and `Cat` represents the "concatenation" (i.e., append) of its two arguments, the second argument appended after the first.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists. (Do not convert back and forth to lists.)

`toSeq` that takes a list and returns a corresponding `Seq` that is balanced.

`fromSeq` that takes a `Seq` and returns the corresponding list.

`appSeq` that takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

`conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

`lenSeq` that takes a `Seq` and returns the number of elements that it contains.

`revSeq` that takes a `Seq` and returns a `Seq` with the same elements in reverse order.

`headSeq` that takes a nonempty `Seq` and returns its head (i.e., leftmost or front) element. (Be careful!)

`tailSeq` that takes a nonempty `Seq` and returns the `Seq` remaining after the head is removed.

`normSeq` that takes a `Seq` and returns a `Seq` with unnecessary embedded `Nil`s removed. (For example, `normSeq (Cat (Cat Nil (Unit 1)) Nil)` returns `(Unit 1)`.)

`eqSeq` that takes two `Seq`s and returns `True` if the sequences of values are equal and returns `False` otherwise. Note that two `Seq` "trees" may be structurally different yet represent the same sequence of values.

For example, `(Cat Nil (Unit 1))` and `(Cat (Unit 1) Nil)` have the same sequence of values (i.e., `[1]`). But `(Cat (Unit 1) (Unit 2))` and `(Cat (Unit 2) (Unit 1))` do not represent the same sequence of values (i.e., `[1,2]` and `[2,1]`, respectively).

Also `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` has the same sequence of values as `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` (i.e., `[1,2,3]`).

In general what are the advantages and disadvantages of representing lists this way?

9 INPUT/OUTPUT

This section from the Gofer/Hugs Notes is obselete. It is left in until the Haskell Notes can be fully revised.

10 PROBLEM SOLVING

A bit of the instructor's philosophy:

- Programming is the essence of computing science.
- Problem solving is the essence of programming.

10.1 Polya's Insights

The mathematician George Polya (1887–1985), a Professor of Mathematics at Stanford University, said the following in the preface to his book *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving* [21].

> Solving a problem means finding a way out of a difficulty, a way around an obstacle, attaining an aim which was not immediately attainable. Solving problems is the specific achievement of intelligence, and intelligence is the specific gift of mankind: solving problems can be regarded as the most characteristically human activity. ...
>
> Solving problems is a practical art, like swimming, or skiing, or playing the piano: you learn it only by imitation and practice. ... if you wish to learn swimming you have to go into the water, and if you wish to become a problem solver you have to solve problems.
>
> If you wish to derive the most profit from your effort, look out for such features of a problem at hand as may be useful in handling the problems to come. A solution that you have obtained by your own effort or one that you have read or heard, but have followed with real interest and insight, may become a *pattern* for you, a model that you can imitate with advantage in solving similar problems. ...
>
> Our knowledge about any subject consists of *information* and *know-how*. If you have genuine *bona fide* experience of mathematical work on any level, elementary or advanced, there will be no doubt in your mind that, in mathematics, know-how is much more important than mere possession of information. ...
>
> What is know-how in mathematics? The ability to solve problems—not merely routine problems but problems requiring some degree of independence, judgment, originality, creativity. Therefore, the first and foremost duty ... in teaching mathematics is to emphasize *methodical work in problem solving.*

What Polya says for mathematics holds just as much for computing science.

In his book *How to Solve It* [20], Polya states four phases of problem solving. These steps are important for programming as well.

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan, checking each step.
4. Reexamine and reconsider the solution. (And, of course, reexamine the understanding of the problem, the plan, and the way the plan was carried out.)

10.2 Problem-Solving Strategies

The Bird and Wadler textbook gives several examples of problem-solving strategies that are sometimes useful in functional programming (indeed all programming).

Solve a more general problem first. That is, solve a "harder" problem than the specific problem at hand, then use the solution of the "harder" problem to get the specific solution desired.

Sometimes a solution of the more general problem is actually easier to find because the problem is simpler to state or more symmetrical or less obscured by special conditions. The general solution can often be used to solve other related problems.

Often the solution of the more general problem can actually lead to a more efficient solution of the specific problem.

Example: We have already seen one example of this technique: finding the first occurrence of an item in a list (page 58 of Bird/Wadler [2], page 83 of these notes).

First, we devised a program to find all occurrences in a list. Then we selected the first occurrence from the set of all occurrences. (Lazy evaluation of Haskell programs means that this use of a more general solution differs very little in efficiency from a specialized version.)

Example: Another example from Bird/Wadler is the fast Fibonacci program on page 128. This program optimizes the efficiency of computing `fib n` by computing `(fib n, fib (n+1))` instead. This is a harder problem, but it actually gives us more information to work with and, hence, provides more opportunity for optimization.

Solve a simpler problem first. Then adapt or extend the solution to solve the original problem.

Often the mass of details in a problem description makes seeing a solution difficult. In the previous technique we made the problem easier by finding a more general problem to solve. In this technique, we move in the other direction: we find a more specific problem that is similar and solve it.

At worst, by solving the simpler problem we should get a better understanding of the problem we really want to solve. The more familiar we are with a problem, the more information we have about it, and, hence, the more likely we will be able to solve it.

At best, by solving the simpler problem we will find a solution that can be easily extended to build a solution to the original problem.

Example: Section 4.1 of Bird/Wadler gives an good example of the application of this technique. The problem is to convert a positive integer of up to six digits to the English words for that number (e.g., 369027 becomes "three hundred and sixty-nine thousand and twenty-seven").

To deal with the complexity of this problem, Bird and Wadler first simplifies the problem to converting a two-digit number to words. Once that is done, they extend the solution to three digits, and then to six digits.

Reuse "off-the-shelf" solutions to standard subproblems. We have been doing this all during this semester, especially since we began began studying polymorphism and higher-order functions.

The basic idea is to identify standard patterns of computation (e.g., standard prelude functions such as `length`, `take`, `zip`, `map`, `filter`, `foldr`) that will solve some aspects of the problem and then combine these standard patterns with your own specialized functions to construct a solution to the problem.

Example: Section 4.2 of Bird/Wadler gives develops a package of functions to do arithmetic on variable length integers. The functions take advantage of several of the standard prelude functions.

Solve a related problem. Then transform the solution of the related problem to get a solution to the original problem.

Perhaps we can find an entirely different problem formulation (i.e., stated in different terms) for which we can readily find a solution. Then that solution can be converted into a solution to the problem at hand.

For example, we may take some problem and recast it in terms of some mathematical or logical framework, solve the corresponding problem in that framework, and then interpret the results for the original problem. The simplification provided by the framework may reveal solutions that are obscured in the details

of the problem. We can also take advantage of the theory and techniques that have been found previously for the mathematical framework.

Example: The example given in Section 4.3 of the Bird and Wadler textbook is quite interesting. The problem is to develop a text processing package, in particular a function to break a string of text up into a list of lines.

This is not trivial. However, the "inverse" problem is trivial. All that is needed to convert a list of lines to a string of text is to insert linefeed characters between the lines.

The example proceeds by first solving the inverse problem (line-folding) and then uses it to calculate what the line-breaking program must be.

Separate concerns. That is, partition the problem into logically separate problems, solve each problem separately, then combine the solutions to the subproblems to construct a solution to the problem at hand.

As we have seen in the above strategies, when a problem is complex and difficult to attack directly, we search for simpler, but related, problems to solve, then build a solution to the complex problem from the simpler problems.

Example: Section 4.5 of Bird/Wadler shows the development of a program to print a calendar for any year. It approaches the problem by first separating it into two independent subproblems: building a calendar and assembling a picture. After solving each of these simpler problems, the more complex problem can be solved easily by combining the two solutions

Divide and conquer. This is a special case of the "solve a simpler problem first" strategy. In this technique, we must divide the problem into subproblems that are the same as the original problem except that the size of the input is smaller.

This process of division continues recursively until we get a problem that can be solved trivially, then we combined we reverse the process by combining the solutions to subproblems to form solutions to larger problems.

Example: Section 6.4 of Bird/Wadler shows the development of divide and conquer programs for sorting and binary search.

There are, of course, other strategies that can be used to approach problem solving.

11 HASKELL "LAWS"

11.1 Stating and Proving Laws

In Section 1 we defined *referential transparency* to mean that, within some well-defined context, a variable (or other symbol) always represents the same value. This allows one expression to be replaced by an equivalent expression or, more informally, "equals to be replaced by equals" .

Referential transparency is probably the most important property of purely functional programming languages like Haskell. It allows us to state and prove various "laws" or identities that hold for functions and to use these "laws"to transform programs into equivalent ones.

We have already seen a number of these laws. Again consider the append operator (`++`) for *finite lists*.

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[] ++ xs     = xs              -- append.1
(x:xs) ++ ys = x:(xs ++ ys)    -- append.2
```

The append operator has two useful properties that we have already seen:

Associativity: For any finite lists `xs`, `ys`, and `zs`,
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.

Identity: For any finite list `xs`, `[] ++ xs = xs = xs ++ []`.

Note: Thus append over a finite list forms a monoid.

How do we prove these properties?

The answer is, of course, induction. But we need a type of induction that allows us to prove theorems over the set of all finite lists. In fact, we have already been using this form of induction in the informal arguments that the list-processing functions terminate.

Induction over the natural numbers is a special case of a more general form of induction called *structural induction*. This type of induction is over the syntactic structure of recursively (inductively) defined objects. Such objects can be partially ordered by a complexity ordering from the most simple (minimal) to the more complex.

If we think about the usual axiomization of the natural numbers (i.e., Peano's postulates), then we see that `0` is the only simple (minimal) object and that the successor function `((+) 1)` is the only constructor.

In the case of finite lists, the only simple object is the nil list `[]` and the only constructor is the cons operator.

To prove a proposition `P(x)` holds for any finite object `x`, one must prove the following cases:

Base cases. That `P(e)` holds for each simple (minimal) object `e`.

Inductive cases. That, for all object constructors `C`, if `P(x)` holds for some arbitrary object(s) `x`, then `P(C(x))` also holds.

That is, we can assume `P(x)` holds, then prove that `P(C(x))` holds. This shows that the constructors preserve proposition `P`.

To prove a proposition `P(xs)` holds for any finite list `xs`, the above reduces to the following cases:

Base case `xs = [].` That `P([])` holds.

Inductive case `xs = (a:as).` That, if `P(as)` holds, then `P(a:as)` also holds.

One, often useful, strategy for discovering proofs of laws is the following:

- Determine whether induction is needed to prove the law. Some laws can be proved directly from the definitions and other previously proved laws.
- Carefully choose the induction variable (or variables).
- Identify the base and inductive cases.
- For each case, use *simplification* independently on each side of the equation. Often, it is best to start with the side that is the most complex.

  Simplification means to substitute the right-hand side of a *definition* or the induction hypothesis for some expression matching the left-hand side.
- Continue simplifying each expression as long as possible.

  Often we can show that the two sides of an equation are the same or that simple manipulations (perhaps using previously proved laws) will show that they are the same.
- If necessary, identify subcases and prove each subcase independently.

A formal proof of a case should, in general, be shown as a calculation that transforms one side of the equation into the other by substitution of equals for equals. This formal proof can be constructed from the calculation suggested in the above strategy.

Now that we have the mathematical machinery we need, let's prove that `++` is associative for all finite lists. The following proofs assume that all arguments of the functions are defined.

11.2 Associativity of ++

Prove: For any finite lists `xs`, `ys`, and `zs`,
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.

Proof:

There does not seem to be a non-inductive proof, thus we proceed by structural induction over the finite lists. But on which variable(s)?

By examining the definition of `++`, we see that it has two legs differentiated by the value of the left operand. The right operand is not decomposed. To use this definition in the proof, we need to consider the left operands of the `++` in the associative law. Thus we choose to do the induction on `xs`, the leftmost operand, and consider two cases.

Base case `xs = [].`

First, we simplify the left-hand side.

```
   [] ++ (ys ++ zs)
=     { append.1 (left to right), omit outer parentheses }
   ys ++ zs
```

We do not know anything about `ys` and `zs`, so we cannot simplify further.

Next, we simplify the right-hand side.

```
   ([] ++ ys) ++ zs
=     { append.1 (left to right), omit parentheses around ys }
   ys ++ zs
```

Thus we have simplified the two sides to the same expression.

Of course, a formal proof can be written more elegantly as:

```
   [] ++ (ys ++ zs)
=     { append.1 (left to right) }
   ys ++ zs
=     { append.1 (right to left, applied to left operand) }
   ([] ++ ys) ++ zs
```

Thus the base case is established.

Note the equational style of reasoning. We proved that one expression was equal to another by beginning with one of the expressions and repeatedly substituting "equals for equals" until we got the other expression.

Each transformational step was justified by a definition, a known property, or (as we see later) the induction hypothesis. We normally do not state justifications like "omit parentheses" or "insert parentheses".

Inductive case `xs = (a:as)`.

Assume `as ++ (ys ++ zs) = (as ++ ys) ++ zs`;
prove `(a:as) ++ (ys ++ zs) = ((a:as) ++ ys) ++ zs`.

First, we simplify the left-hand side.

```
   (a:as) ++ (ys ++ zs)
=     { append.2 (left to right) }
   a:(as ++ (ys ++ zs))
=     { induction hypothesis }
   a:((as ++ ys) ++ zs)
```

We do not know anything further about `as`, `ys`, and `zs`, so we cannot simplify further.

Next, we simplify the right-hand side.

```
   ((a:as) ++ ys) ++ zs
=     { append.2 (left to right, on inner ++) }
   (a:(as ++ ys)) ++ zs
=     { append.2 (left to right, on outer ++) }
   a:((as ++ ys) ++ zs)
```

Thus we have simplified the two sides to the same expression.

Again, a formal proof can be written more elegantly as follows:

```
   (a:as) ++ (ys ++ zs)
=     { append.2 (left to right) }
   a:(as ++ (ys ++ zs))
=     { induction hypothesis }
   a:((as ++ ys) ++ zs)
=     { append.2 (right to left, on outer ++) }
   (a:(as ++ ys)) ++ zs
=     { append.2 (right to left, on inner ++) }
   ((a:as) ++ ys) ++ zs
```

Thus the inductive case is established.

Therefore, we have proven the `++` associativity property. Q.E.D.

Note: The above proof and the ones that follow assume that the arguments of the functions are all defined (i.e., not equal to $\perp$).

You should practice writing proofs in the "more elegant" form given above. This end-to-end calculational style is more useful for synthesis of programs.

Notes:

- Make sure you need to use induction.
- Choose the induction variable carefully.
- Be careful with parentheses. Substitutions, comparisons, and pattern matches must done with the fully parenthesized forms of definitions, laws, and expressions in mind, that is, with parentheses around all binary operations, simple objects, and the entire expression. We often omit "unneeded" parentheses to make the expression more readable.
- Start with the more complex side of the equation. You have more information with which to work.

11.3 Identity Element for ++

Prove: For any finite list `xs`, `[] ++ xs = xs = xs ++ []`.

Proof:

The equation `[] ++ xs = xs` follows directly from `append.1`. Thus we consider the equation `xs ++ [] = xs`, which we prove by structural induction on `xs`.

Base case `xs = []`.

```
   [] ++ []
=     { append.1 (left to right) }
   []
```

This establishes the base case.

Inductive case `xs = (a:as)`.

Assume `as ++ [] = as`; prove `(a:as) ++ [] = (a:as)`.

```
   (a:as) ++ []
=     { append.2 (left to right) }
   a:(as ++ [])
=     { induction hypothesis }
   a:as
```

This establishes the inductive case.

Therefore, we have proved that `[]` is the identity element for `++`. Q.E.D.

11.4 Relating `length` and `++`

Suppose that the list `length` function is defined as follows:

```
length :: [a] -> Int
length []     = 0               -- length.1
length (_:xs) = 1 + length xs   -- length.2
```

Prove: For all finite lists `xs` and `ys`: `length (xs++ys) = length xs + length ys`.

Proof: Because of the way `++` is defined, we choose `xs` as the induction variable.

Base case `xs = []`.

```
   length [] + length ys
=     { length.1 (left to right) }
   0 + length ys
=     { 0 is identity for addition }
   length ys
=     { append.1 (right to left) }
   length ([] ++ ys)
```

This establishes the base case.

Inductive case `xs = (a:as)`.

Assume `length (as ++ ys) = length as + length ys`;
prove `length ((a:as) ++ ys) = length (a:as) + length ys`.

```
   length ((a:as) ++ ys)
=     { append.2 (left to right) }
   length (a:(as ++ ys))
=     { length.2 (left to right) }
   1 + length (as ++ ys)
=     { induction hypothesis }
   1 + (length as + length ys)
=     { associativity of addition }
   (1 + length as) + length ys
=     { length.2 (right to left, value of a arbitrary) }
   length (a:as) + length ys
```

This establishes the inductive case.

Therefore, `length (xs ++ ys) = length xs + length ys`. Q.E.D.

Note: The proof uses the associativity and identity properties of integer addition.

11.5 Relating `take` and `drop`

Remember the definitions for the list functions `take` and `drop`.

```
take :: Int -> [a] -> [a]
take 0     _      = []              -- take.1
take _     []     = []              -- take.2
take (n+1) (x:xs) = x : take n xs   -- take.3

drop :: Int -> [a] -> [a]
drop 0     xs     = xs              -- drop.1
drop _     []     = []              -- drop.2
drop (n+1) (_:xs) = drop n xs       -- drop.3
```

Prove: For any natural numbers `n` and finite lists `xs`,
`take n xs ++ drop n xs = xs`.

Proof:

Note that both `take` and `drop` use both arguments to distinguish the cases. Thus we must do an induction over all natural numbers `n` and all finite lists `xs`.

We would expect four cases to consider, the combinations from `n` being zero and nonzero and `xs` being nil and non-nil. But an examination of the definitions for the functions reveal that the cases for `n = 0` collapse into a single case.

Base case `n = 0`.

```
   take 0 xs ++ drop 0 xs
=     { take.1, drop.1 (both left to right) }
   [] ++ xs
=     { ++ identity }
   xs
```

This establishes the case.

Base case `n = m+1`, `xs = []`.

```
   take (m+1) [] ++ drop (m+1) []
=     { take.2, drop.2 (both left to right) }
   [] ++ []
=     { ++ identity }
   []
```

This establishes the case.

Inductive case `n = m+1, xs = (a:as)`.

Assume `take m as ++ drop m as = as`;
prove `take (m+1) (a:as) ++ drop (m+1) (a:as) = (a:as)`.

```
   take (m+1) (a:as) ++ drop (m+1) (a:as)
=     { take.3, drop.3 (both left to right) }
   (a:(take m as)) ++ drop m as
=     { append.2 (left to right) }
   a:(take m as ++ drop m as)
=     { induction hypothesis }
   (a:as)
```

This establishes the case.

Therefore, the property is proved. Q.E.D.

11.6 Equivalence of Functions

What do we mean when we say two functions are equivalent?

Usually, we mean that the "same inputs" yield the "same outputs". For example, single argument functions `f` and `g` are equivalent if `f x = g x` for all `x`.

In Section 5.4 we defined two versions of a function to reverse the elements of a list. Function `rev` uses backward recursion and function `reverse` (called `reverse'` in Section 5.4) uses a forward recursive auxiliary function `rev'`.

```
rev :: [a] -> [a]
rev []     = []                                  -- rev.1
rev (x:xs) = rev xs ++ [x]                       -- rev.2

reverse :: [a] -> [a]
reverse xs = rev' xs []                          -- reverse.1
             where rev' [] ys     = ys           -- reverse.2
                   rev' (x:xs) ys = rev' xs (x:ys) -- reverse.3
```

To show `rev` and `reverse` are equivalent, we must prove that, for all finite lists `xs`:

```
rev xs = reverse xs
```

If we unfold (i.e., simplify) `reverse` one step, we see that we need to prove:

```
rev xs = rev' xs []
```

Thus let's try to prove this by structural induction on `xs`.

Base case `xs = []`.

```
   rev []
=     { rev.1 (left to right) }
   []
=     { reverse.2 (right to left) }
   rev' [] []
```

This establishes the base case.

Inductive case `xs = (a:as)`

Given `rev as = rev' as []`, prove `rev (a:as) = rev' (a:as) []`.

First, we simplify the left side.

```
   rev (a:as)
=     { rev.2 (left to right) }
   rev as ++ [a]
```

Then, we simplify the right side.

```
   rev' (a:as) []
=     { reverse.3 (left to right) }
   rev' as [a]
```

Thus we need to show that `rev as ++ [a] = rev' as [a]`. But we do not know how to proceed from this point. Maybe another induction. But that would probably just bring us back to a point like this again. We are stuck!

Let's look back at `rev xs = rev' xs []`. This is difficult to prove directly. Note the asymmetry, one argument for `rev` versus two for `rev'`.

Thus let's look for a new, more symmetrical, problem that might be easier to solve. Often it is easier to find a solution to a problem that is symmetrical than one which is not.

Note the place we got stuck above (proving `rev as ++ [a] = rev' as [a]`) and also note the equation `reverse.3`. Taking advantage of the identity element for `++`, we can restate our property in a more symmetrical way as follows:

```
rev xs ++ [] = rev' xs []
```

Note that the constant `[]` appears on both sides of the above equation. We can now apply the following generalization heuristic. (That is, we try to solve a "harder" problem.)

Heuristic: Generalize by replacing a constant (or any subexpression) by a variable.

Thus we try to prove the more general proposition:

```
rev xs ++ ys = rev' xs ys
```

The case `ys = []` gives us what we really want to hold. Intuitively, this new proposition seems to hold. Now let's prove it formally. Again we try structural induction on `xs`.

Base case `xs = [].`

```
   rev [] ++ ys
=     { rev.1 (left to right) }
   [] ++ ys
=     { append.1 (left to right) }
   ys
=     { reverse.2 (right to left) }
   rev' [] ys
```

This establishes the base case.

Inductive case `xs = (a:as).`

Assume `rev as ++ ys = rev' as ys` for any finite list `ys`;
prove `rev (a:as) ++ ys = rev' (a:as) ys`.

```
   rev (a:as) ++ ys
=     { rev.2 (left to right) }
   (rev as ++ [a]) ++ ys
=     { ++ associativity, Note 1 }
   rev as ++ ([a] ++ ys)
=     { singleton law, Note 2 }
   rev as ++ (a:ys)
=     { induction hypothesis }
   rev' as (a:ys)
=     { reverse.3 (right to left) }
   rev' (a:as) ys
```

This establishes the inductive case.

Note 1: We could apply the induction hypothesis here, but it does not seem profitable. Keeping the expressions in terms of `rev` and `++` as long as possible seems better; we know more about those expressions.

Note 2: The singleton law is `[x] ++ xs = x:xs` for any element `x` and finite list `xs` of the same type. Proof of this is left as an exercise for the reader.

Therefore, we have proved `rev xs ++ ys = rev' xs ys`, and, hence,
`rev xs = reverse xs`.

The key to the performance improvement here is the solution of a "harder" problem: function `rev'` does both the reversing and appending of a list while `rev` separates the two actions.

11.7 Exercises

1. Prove for all `x` of some type and finite lists `xs` of the same type (singleton law):

   ```
   [x] ++ xs = (x:xs)
   ```

2. Consider the definition for `length` given in Section 5.2.2 and the following definition for `len`:

   ```
   len :: Int -> [a] -> Int
   len n [ ]    = n              -- len.1
   len n (_:xs) = len (n+1) xs   -- len.2
   ```

   Prove for any finite list `xs`: `len 0 xs = length xs`.

3. Prove for all finite lists `xs` and `ys` of the same type:

   ```
   reverse (xs ++ ys) = reverse ys ++ reverse xs
   ```

   Hint: The function `reverse` (called `reverse'` in Section 5.4) uses forward recursion. Backward recursive definitions are generally easier to use in inductive proofs. In Section 5.4 we also defined a backward recursive function `rev` and proved that `rev xs = reverse xs` for all finite lists `xs`. Thus, you may find it easier to substitute `rev` for `reverse` and instead prove:

   ```
   rev (xs ++ ys) = rev ys ++ rev xs
   ```

4. Prove for all finite lists `xs` of some type: `reverse (reverse xs) = xs`

5. Prove for all natural numbers `m` and `n` and all finite lists `xs`:

   ```
   drop n (drop m xs) = drop (m+n) xs
   ```

6. Consider the rational number package from Section 5.6. Prove for any `BigRat` value `r`:

   ```
   addRat r (0,1)  =  normRat r  =  addRat (0,1) r
   ```

7. Consider the two definitions for the Fibonacci function in Section 5.4.6. Prove for any natural number `n`:

   ```
   fib n = fib' n
   ```

   Hint: First prove, for `n` $\geq$ `2`:

   ```
   fib'' n p q = fib'' (n-2) p q + fib'' (n-1) p q
   ```

8. Prove that function `id` is the identity element of functional composition.

9. Prove that functional composition is associative.

10. Prove for all finite lists `xs` and `ys` of the same type and function `f` on that type:

```
map f (xs ++ ys) = map f xs ++ map f ys
```

11. Prove for all finite lists `xs` and `ys` of the same type and predicate `p` on that type:

```
filter p (xs ++ ys) = filter p xs ++ filter p ys
```

12. Prove for all finite lists `xs` and `ys` of the same type and all predicates `p` on that type:

```
all p (xs ++ ys) = (all p xs) && (all p ys)
```

Note:
```
(&&) :: Bool -> Bool -> Bool
False && x  = False  -- second argument not evaluated
True  && x  = x
```

13. Prove for all finite lists `xs` of some type and predicates `p` and `q` on that type:

```
filter p (filter q xs) = filter q (filter p xs)
```

14. Prove for all finite lists `xs` and `ys` of the same type and for all functions `f` and values `a` of compatible types:

```
foldr f a (xs ++ ys) = foldr f (foldr f a ys) xs
```

15. Prove for all finite lists `xs` of some type and all functions `f` and `g` of conforming types:

```
map (f . g) xs = (map f . map g) xs
```

16. Prove for all finite lists of finite lists `xss` of some base type and function `f` on that type:

```
map f (concat xss) = concat (map (map f) xss)
```

17. Prove for all finite lists `xs` of some type and functions `f` on that type:

```
map f xs = foldr ((:) . f) [] xs
```

18. Prove for all lists `xs` and predicates `p` on the same type:

```
takeWhile p xs ++ dropWhile p xs = xs
```

19. Prove that, if $\oplus$ is a associative binary operation of type `t -> t` with identity element `z` (i.e., a monoid), then

    ```
    foldr (⊕) z xs = foldl (⊕) z xs
    ```

20. Remember the Haskell type for the natural numbers given in an exercise in Section 5.6:

    ```
    data  Nat = Zero | Succ Nat
    ```

    For the functions defined in that exercise, prove the following:

    (a) Prove that `intToNat` and `natToInt` are inverses of each other.

    (b) Prove that `Zero` is the (right and left) identity element for `addNat`.

    (c) Prove for any `Nat`s `x` and `y`: `addNat (Succ x) y = addNat x (Succ y)`.

    (d) Prove associativity of addition on `Nat`s. That is, for any `Nat`s `x`, `y`, and `z`, `addNat x (addNat y z) = addNat (addNat x y) z`.

    (e) Prove commutativity of addition on `Nat`s. That is, for any `Nat`s `x` and `y`, `addNat x y = addNat y x`.

12 PROGRAM SYNTHESIS

This section deals with program synthesis.

In the *proof* of a property, we take an existing program and then demonstrate that it satisfies some property.

In the *synthesis* of a program, we take a property called a *specification* and then synthesize a program that satisfies it [2]. (Program synthesis is called program *derivation* in other contexts.)

Both proof and synthesis require essentially the same reasoning. Often a proof can be turned into a synthesis by simply reversing a few of the steps, and vice versa.

12.1 Fast Fibonacci Function

Reference: This subsection is based on Sections 5.4.5 and 5.5 of the Bird and Wadler textbook and Section 4.5 of Hoogerwoord's dissertation *The Design of Functional Programs: A Calculational Approach* [10].

A (second-order) Fibonacci sequence is the sequence in which the first two elements are 0 and 1 and each successive element is the sum of the two immediately preceding elements: 0, 1, 1, 2, 3, 5, 8, 13, $\cdots$

As we have seen in Section 5.4.6, we can take the above informal description and define a function to compute the `n`th element of the Fibonacci sequence. The definition is straightforward. Unfortunately, this algorithm is quite inefficient, $\mathcal{O}($`fib n`$)$.

```
fib :: Int -> Int
fib 0      = 0                  -- fib.1
fib 1      = 1                  -- fib.2
fib (n+2)  = fib n + fib (n+1) -- fib.3
```

In Section 5.4.6 we also developed a more efficient, but less straightforward, version by using two accumulating parameters. This definition seemed to be "pulled out of thin air". Can we synthesize a definition that uses the more efficient algorithm from the simpler definition above?

Yes, but we use a slightly different approach than we did before. We can improve the performance of the Fibonacci computation by using a technique called *tupling* [10].

The tupling technique can be applied to a set of functions with the same domain and the same recursive pattern. First, we define a new function whose value is a tuple, the components of which are the values of the original functions. Then, we proceed to calculate a recursive definition for the new function.

This technique is similar to the technique of adding accumulating parameters to define a new function.

Given the definition of `fib` above, we begin with the specification [2]

```
twofib n = (fib n, fib (n+1))
```

and synthesize a recursive definition by using induction on the natural number `n`.

Base case `n = 0`.

```
   twofib 0
=     { specification }
   (fib 0, fib (0+1))
=     { arithmetic, fib.1, fib.2 }
   (0,1)
```

This gives us a definition for the base case.

Inductive case `n = m+1`.

Given that there is a definition for `twofib m` that satisfies the specification (i.e., `twofib m = (fib m, fib (m+1))`), calculate a definition for `twofib (m+1)` that satisfies the specification.

```
   twofib (m+1)
=     { specification }
   (fib (m+1), fib ((m+1)+1))
=     { arithmetic, fib.3  }
   (fib (m+1), fib m + fib (m+1))
=     { modularization }
   (b,a+b)
   where (a,b) = (fib m, fib (m+1))
=     { induction hypothesis }
   (b,a+b)
   where (a,b) = twofib m
```

This gives us a definition for the inductive case.

Bringing the cases together, we get the following definition:

```
twofib :: Int -> (Int,Int)
twofib 0     = (0,1)
twofib (n+1) = (b,a+b)
               where (a,b) = twofib n

fastfib :: Int -> Int
fastfib n = fst (twofib n)
```

Above `fst` is the standard prelude function to extract the first component of a pair (i.e., a 2-tuple).

The key to the performance improvement is solving a "harder" problem: computing `fib n` and `fib (n+1)` at the same time. This allows the values needed to be "passed forward" to the "next iteration".

In general, we can approach the synthesis of a function using the following method:

- Devise a specification for the function in terms of defined functions, data, etc.
- Assume the specification holds.
- Using proof techniques (as if proving the specification), calculate an appropriate definition for the function.
- As needed, break the synthesis calculation into cases motivated by the induction "proof" over an appropriate (well-founded) set (e.g., over natural numbers or finite lists). The inductive cases usually correspond to recursive legs of the definition.

12.2 Sequence of Fibonacci Numbers

Now let's consider a function to generate a list of the elements `fib 0` through `fib n` for some natural number `n`. A simple backward recursive definition follows:

```
allfibs :: Int -> [Int]
allfibs 0      = [0]                        -- allfibs.1
allfibs (n+1)  = allfibs n ++ [fib (n+1)] -- allfibs.2
```

Using `fastfib`, each `fib n` calculation is $\mathcal{O}(\mathtt{n})$. Each `++` call is also $\mathcal{O}(\mathtt{n})$. The `fib` and the `++` are "in sequence", so each call of `allfibs` is just $\mathcal{O}(\mathtt{n})$. However, there are $\mathcal{O}(\mathtt{n})$ recursive calls of `allfibs`, so the overall complexity is $\mathcal{O}(\mathtt{n}^2)$.

We again attempt to improve the efficiency by tupling. We begin with the following specification for `fibs`:

```
fibs n = (fib n, fib (n+1), allfibs n)
```

We already have definitions for the functions on the right-hand side, `fib` and `allfibs`. Our task now is to synthesize a definition for the left-hand side, `fibs`.

We proceed by induction on the natural number `n` and consider two cases.

Base case `n = 0`.

```
   fibs 0
=     { fibs specification }
   (fib 0, fib (0+1), allfibs 0)
=     { fib.1, fib.2, allfibs.1 }
   (0,1,[0])
```

This gives us a definition for the base case.

Inductive case `n = m+1`.

Given that there is a definition for `fibs m` that satisfies the specification (i.e., `fibs m = (fib m, fib (m+1), allfibs m)`), calculate a definition for `fibs (m+1)` that satisfies the specification.

```
   fibs (m+1)
=     { fibs specification }
   (fib (m+1), fib (m+2), allfibs (m+1))
=     { fib.3, allfibs.2  }
   (fib (m+1), fib m + fib (m+1), allfibs m ++ [fib (m+1)])
=     { modularization }
   (b,a+b,c++[b])
   where (a,b,c) = (fib m, fib (m+1), allfibs m)
=     { induction hypothesis }
   (b,a+b,c++[b])
   where (a,b,c) = fibs m
```

This gives us a definition for the inductive case.

Bringing the cases together, we get the following definitions:

```
fibs :: Int -> (Int,Int,[Int])
fibs 0     = (0,1,[0])
fibs (n+1) = (b,a+b,c++[b])
             where (a,b,c) = fibs n

allfibs1 :: Int -> [Int]
allfibs1 n = thd3 (fibs n)
```

Above `thd3` is the standard prelude function to extract the third component of a 3-tuple.

We have eliminated the $\mathcal{O}(\mathtt{n})$ `fib` calculations, but still have an $\mathcal{O}(\mathtt{n})$ append (`++`) within each of the $\mathcal{O}(\mathtt{n})$ recursive calls of `fibs`. This program is better, but is still $\mathcal{O}(\mathtt{n}^2)$.

Note that in the `c ++ [b]` expression there is a single element on the right. Perhaps we could build this term backwards using cons, an $\mathcal{O}(1)$ operation, and then reverse the final result.

We again attempt to improve the efficiency by tupling. We begin with the following specification for `fibs`:

```
fibs' n = (fib n, fib (n+1), reverse (allfibs n))
```

For convenience in calculation, we replace `reverse` by its backward recursive equivalent `rev`.

```
rev :: [a] -> [a]
rev []     = []            -- rev.1
rev (x:xs) = rev xs ++ [x] -- rev.2
```

We again proceed by induction on `n` and consider two cases.

Base case `n = 0`.

```
   fibs' 0
=     { fibs' specification }
   (fib 0, fib (0+1), rev (allfibs 0))
=     { fib.1, fib.2, allfibs.1 }
   (0,1, rev [0])
=     { rev.2 }
   (0,1, rev [] ++ [0])
=     { rev.1, append.1 }
   (0,1,[0])
```

This gives us a definition for the base case.

Inductive case `n = m+1`.

Given that there is a definition for `fibs' m` that satisfies the specification (i.e., `fibs' m = (fib m, fib (m+1), allfibs m)`), calculate a definition for `fibs' (m+1)` that satisfies the specification.

```
   fibs' (m+1)
=     { fibs' specification }
   (fib (m+1), fib (m+2), rev (allfibs (m+1)))
=     { fib.3, allfibs.2  }
   (fib (m+1), fib m + fib (m+1), rev (allfibs m ++ [fib (m+1)]))
=     { modularization }
   (b, a+b, rev (allfibs m ++ [b]))
   where (a,b,c) = (fib m, fib (m+1), rev (allfibs m))
```

```
=     { induction hypothesis }
   (b, a+b, rev (allfibs m ++ [b]))
   where (a,b,c) = fibs' m
=     { rev (xs ++ [x]) = x : rev xs, Note 1 }
   (b, a+b, b : rev (allfibs m))
   where (a,b,c) = fibs' m
=     { substitution }
   (b, a+b, b:c)
   where (a,b,c) = fibs' m
```

This gives us a definition for the inductive case.

Note 1: The proof of `rev (xs ++ [x]) = x : rev xs` is left as an exercise.

Bringing the cases together, we get the following definition:

```
fibs' :: Int -> (Int,Int,[Int])
fibs' 0     = (0,1,[0])
fibs' (n+1) = (b,a+b,b:c)
              where (a,b,c) = fibs' n

allfibs2 :: Int -> [Int]
allfibs2 n = reverse (thd3 (fibs' n))
```

Function `fibs'` is $\mathcal{O}(\mathtt{n})$. Hence, `allfibs2` is $\mathcal{O}(\mathtt{n})$.

Are further improvements possible?

Clearly, function `fibs'` must generate an element of the sequence for each integer in the range `[0..n]`. Thus no complexity order improvement is possible.

However, from our previous experience, we know that it should be possible to avoid doing a reverse by using a tail recursive auxiliary function to compute the Fibonacci sequence. The investigation of this possible improvement is left to the reader.

For an $\mathcal{O}(\log_2 \mathtt{n})$ algorithm to compute `fib n`, see Section 5.2 of Kaldewaij's textbook on program derivation [18].

12.3 Synthesis of `drop` from `take`

Suppose that we have the following definition for the list function `take`, but no definition for `drop`.

```
take :: Int -> [a] -> [a]
take 0     _      = []              -- take.1
take _     []     = []              -- take.2
take (n+1) (x:xs) = x : take n xs   -- take.3
```

Further suppose that we wish to synthesize a definition for `drop` that satisfies the following specification for any natural number `n` and finite list `xs`.

```
take n xs ++ drop n xs = xs
```

We proved this as a property earlier, given definitions for both `take` and `drop`. The synthesis uses induction on both `n` and `xs` and the same cases we used in the proof.

Base case `n = 0`.

```
   xs
=     { specification, substitution for this case }
   take 0 xs ++ drop 0 xs
=     { take.1 }
   [] ++ drop 0 xs
=     { ++ identity }
   drop 0 xs
```

This gives the equation `drop 0 xs = xs`.

Base case `n = m+1`, `xs = []`.

```
   []
=     { specification, substitution for this case }
   take (m+1) [] ++ drop (m+1) []
=     { take.2 }
   [] ++ drop (m+1) []
=     { ++ identity }
   drop (m+1) []
```

This gives the defining equation `drop (m+1) [] = []`. Since the value of the argument `(m+1)` is not used in the above calculation, we can generalize the definition to `drop _ [] = []`.

Inductive case `n = m+1, xs = (a:as)`.

Given that there is a definition for `drop m as` that satisfies the specification (i.e., `take m as ++ drop m as = as`), calculate an appropriate definition for `drop (m+1) (a:as)` that satisfies the specification.

```
   (a:as)
=     { specification, substitution for this case }
   take (m+1) (a:as) ++ drop (m+1) (a:as)
=     { take.3 }
   (a:(take m as)) ++ drop (m+1) (a:as)
=     { append.2 }
   a:(take m as ++ drop (m+1) (a:as))
```

Hence, `a:(take m as ++ drop (m+1) (a:as)) = (a:as)`.

```
   a:(take m as ++ drop (m+1) (a:as)) = (a:as)
≡     { axiom of equality of lists (Note 1) }
   take m as ++ drop (m+1) (a:as) = as
≡     { m ≥ 0, specification }
   take m as ++ drop (m+1) (a:as) = take m as ++ drop m as
≡     { equality of lists (Note 2) }
   drop (m+1) (a:as) = drop m as
```

Because of the induction hypothesis, we know that `drop m as` is defined. This gives a definition for this case.

Note 0: The symbol ≡ denotes logical equivalence (i.e., if and only if) and is pronounced "equivales".

Note 1: `(x:xs) = (y:ys)` ≡ `x = y && xs = ys`. In this case `x` and `y` both equal `a`.

Note 2: `xs ++ ys = xs ++ zs` ≡ `ys = zs` can be proved by induction on `xs` using the Note 1 property.

Bringing the cases together, we get the definition that we saw earlier.

```
drop :: Int -> [a] -> [a]
drop 0     xs     = xs         -- drop.1
drop _     []     = []         -- drop.2
drop (n+1) (_:xs) = drop n xs  -- drop.3
```

12.4 Tail Recursion Theorem

In Section 5.4 we looked at two different definitions of a function to reverse the elements of a list. Function `rev` uses a straightforward backward linear recursive technique and `reverse` uses a tail recursive auxiliary function. We proved that these definitions are equivalent.

```
rev :: [a] -> [a]
rev []     = []                                    -- rev.1
rev (x:xs) = rev xs ++ [x]                         -- rev.2

reverse :: [a] -> [a]
reverse xs = rev' xs []                            -- reverse.1
             where rev' [] ys     = ys             -- reverse.2
                   rev' (x:xs) ys = rev' xs (x:ys) -- reverse.3
```

Function `rev'` is a *generalization* of `rev`. Is there a way to calculate `rev'` from `rev`?

Yes, by using the Tail Recursion Theorem for lists. We develop this theorem in a more general setting than `rev`.

Reference: The following is based on Section 4.7 of Hoogerwoord's dissertation [10].

For some types `X` and `Y`, let function `fun` be defined as follows:

```
fun :: X -> Y
fun x | not (b x) = f x              -- fun.1
      | b x       = h x ⊙ fun (g x) -- fun.2
```

- Functions `b`, `f`, `g`, `h`, and $\odot$ are not defined in terms of `fun`.
- `b :: X -> Bool` such that, for any `x`, `b x` is defined whenever `fun x` is defined.
- `g :: X -> X` such that, for any `x`, `g x` is defined whenever `fun x` is defined and `b x` holds.
- `h :: X -> Y` such that, for any `x`, `h x` is defined whenever `fun x` is defined and `b x` holds.
- `(⊙) :: Y -> Y -> Y` such that operation $\odot$ is defined for all elements of `Y` and is an associative operation with left identity `e`.
- `f :: X -> Y` such that, for any `x`, `f x` is defined whenever `fun x` is defined and `not (b x)` holds.
- Type `X` with relation $\prec$ admits induction (i.e., $\langle$`X`, $\prec\rangle$ is a well-founded ordering).
- For any `x`, if `fun x` is defined and `b x` holds, then `g x` $\prec$ `x`.

Note that both `fun x` and the recursive leg `h x ⊙ fun (g x)` have the general structure `y ⊙ fun z` for some expressions `y` and `z` (i.e., `fun x = e ⊙ fun x`). Thus we specify a more general function `fun'` such that

```
fun' :: Y -> X -> Y
fun' y x = y ⊙ fun x
```

and such that `fun'` is defined for any `x` $\in$ `X` for which `fun x` is defined.

Given the above specification, we note that:

```
   fun' e x
=     { fun' specification }
   e ⊙ fun x
=     { e is the left identity for ⊙ }
   fun x
```

We proceed by induction on the type `X` with $\prec$. (We are using a more general form of induction than we have before.)

We have two cases. The base case is when `not (b x)` holds for argument `x` of `fun'`. The inductive case is when `b x` holds (i.e, `g x` $\prec$ `x`).

Base case `not (b x).` (That is, `x` is a minimal element of `X` under $\prec$.)

```
      fun' y x
=        { fun' specification }
      y ⊙ fun x
=        { fun.1 }
      y ⊙ f x
```

Inductive case `b x`**.** (That is, `g x` $\prec$ `x`.)

Given that there is a definition for `fun' y (g x)` that satisfies the specification for any `y` (i.e., `fun' y (g x) = y` $\odot$ `fun (g x)`), calculate a definition for `fun' y x` that satisfies the specification.

```
   fun' y x
=     { fun' specification }
   y ⊙ fun x
=     { fun.2 }
   y ⊙ (h x ⊙ fun (g x))
=     { ⊙ associativity }
   (y ⊙ h x) ⊙ fun (g x)
=     { g x ≺ x, induction hypothesis }
   fun' (y ⊙ h x) (g x)
```

Thus we have synthesized the following tail recursive definition for function `fun'` and essentially proved the Tail Recursion Theorem shown below.

```
fun' :: Y -> X -> Y
fun' y x | not (b x) = y ⊙ f x              -- fun'.1
         | b x       = fun' (y ⊙ h x) (g x) -- fun'.2
```

Note that the first parameter of `fun'` is an accumulating parameter.

Tail Recursion Theorem: If `fun` and `fun'` are defined as given above, then `fun x = fun' e x`.

Now let's consider the `rev` and `rev'` functions again. First, let's rewrite the definitions of `rev` in a form similar to the definition of `fun`.

```
rev :: [a] -> [a]
rev xs | xs == []  = []                          -- rev.1
       | xs /= []  = rev (tail xs) ++ [head xs] -- rev.2
```

For `rev` we substitute the following for the components of the `fun` definition:

- `fun x` $\leftarrow$ `rev xs`
- `b x` $\leftarrow$ `xs /= []`
- `g x` $\leftarrow$ `tail xs`
- `h x` $\leftarrow$ `[head xs]`

- `l` $\odot$ `r` $\leftarrow$ `r ++ l` (Note the flipped operands)
- `f x` $\leftarrow$ `[]`
- `l` $\prec$ `r` $\leftarrow$ `(length l) < (length r)`
- `e` $\leftarrow$ `[]`
- `fun' y x` $\leftarrow$ `rev' xs ys` (Note the flipped arguments)

Thus, by applying the tail recursion theorem, `fun'` becomes the following:

```
rev' :: [a] -> [a] -> [a]
rev' xs ys | xs == []  = ys                                  -- rev'.1
           | xs /= []  = rev' (tail xs) ([head xs]++ys) -- rev'.2
```

From the Tail Recursion Theorem, we conclude that `rev xs = rev' xs []`.

Why would we want to convert a backward linear recursive function to a tail recursive form?

- A tail recursive definition is sometimes more space efficient. (This is especially the case if the strictness of an accumulating parameter can be exploited. See Section 13.5.)
- A tail recursive definition sometimes allows the replacement of an "expensive" operation (requiring many steps) by a less "expensive" one. (For example, `++` is replaced by cons in the transformation from `rev` to `rev'`.)
- A tail recursive definition can be transformed (either by hand or by a compiler) into an efficient loop.
- A tail recursive definition is usually more general than its backward linear recursive counterpart Sometimes we can exploit this generality to synthesize a more efficient definition. (We see an example of this in the next subsection.)

12.5 Finding Better Tail Recursive Algorithms

Reference: This section is adapted from Section 11.3 of Cohen's textbook [4].

Although the Tail Recursion Theorem is important, the technique we used to develop it is perhaps even more important. We can sometimes use the technique to transform one tail recursive definition into another that is more efficient [10].

Consider exponentiation by a natural number power. The operation `**` can be defined recursively in terms of multiplication as follows:

```
infixr 8 **
(**) :: Int -> Int -> Int
m ** 0     = 1            -- **.1
m ** (n+1) = m * (m ** n) -- **.2
```

For `(**)` we substitute the following for the components of the `fun` definition of the previous subsection:

- `fun x` ← `m ** n`
- `b x` ← `n > 0` (Applied only to natural numbers)
- `g x` ← `n-1`
- `h x` ← `m`
- `l` ⊙ `r` ← `l * r`
- `f x` ← `1`
- `l` ≺ `r` ← `l < r`
- `e` ← `1`
- `fun' y x` ← `exp a m n`

Thus, by applying the Tail Recursion Theorem, we define the function `exp` such that

```
exp a m n = a * (m ** n)
```

and, in particular, `exp 1 m n = m ** n`.

The resulting function `exp` is defined as follows:

```
exp :: Int -> Int -> Int -> Int
exp a m 0     = a              -- exp.1
exp a m (n+1) = exp (a*m) m n -- exp.2
```

In terms of time, this function is no more efficient than the original version; both require $\mathcal{O}(\mathtt{n})$ multiplies. (However, by exploiting the strictness of the first parameter, `exp` can be made more space efficient than `**`. See Section 13.5.)

Note that `exp` algorithm converges upon the final result in steps of one. Can we take advantage of the generality of `exp` and the arithmetic properties of exponentiation to find an algorithm that converges in larger steps?

Yes, we can by using the technique that we used to develop the Tail Recursion Theorem. In particular, let's try to synthesize an algorithm that converges logarithmically (in steps of half the distance) instead of linearly.

Speaking operationally, we are looking for a "short cut" to the result. To find this short cut, we use the "maps" that we have of the "terrain". That is, we take advantage of the properties we know about the exponentiation operator.

We thus attempt to find expressions `x` and `y` such that

```
exp x y (n/2) = exp a m n
```

where "`/`" represents division on integers.

For the base case where `n = 0`, this is trivial. We proceed with a calculation to discover values for `x` and `y` that make `exp x y (n/2) = exp a m n` when `n > 0` (i.e., in the inductive case). In doing this we can use the specification for `exp` (i.e., `exp a m n = a * (m ** n)`).

```
   exp x y (n/2)
=     { exp specification }
   x * (y ** (n/2))
=     { Choose y = m ** 2 (Note 1) }
   x * ((m ** 2) ** (n/2))
```

Note 1: The strategy is to make choices for `x` and `y` that make `x * (y ** (n/2))` equal to `a * (m ** n)`. This choice for `y` is toward getting the `m ** n` term.

Because we are dealing with integer division, we need to consider two cases because of truncation.

Subcase `even n` (for `n > 0`).

```
   x * ((m ** 2) ** (n/2))
=     { arithmetic properties of exponentiation, n even }
   x * (m ** n)
=     { Choose x = a, toward getting a * (m ** n) }
   a * (m ** n)
=     { exp specification }
   exp a m n
```

Thus, for even `n`, we derive `exp a m n = exp a (m*m) (n/2)`. We optimize and replace `m ** 2` by `m * m`.

Subcase `odd n` (for `n > 0`). That is, `n/2 = (n-1)/2`.

```
   x * ((m ** 2) ** ((n-1)/2))
=     { arithmetic properties of exponentiation }
   x * (m ** (n-1))
=     { Choose x = a * m, toward getting a * (m ** n) }
   (a * m) * (m ** (n-1))
=     { arithmetic properties of exponentiation }
   a * (m ** n)
=     { exp specification }
   exp a m n
```

Thus, for odd `n`, we derive `exp a m n = exp (a*m) (m*m) (n/2)`.

To differentiate the logarithmic definition for exponentiation from the linear one, we rename the former to `exp'`. We have thus defined `exp'` as follows:

```
exp' :: Int -> Int -> Int -> Int
exp' a m 0                = a                          -- exp'.1
exp' a m n@(p+1) | even n = exp' a     (m*m) (n/2)     -- exp'.2
                 | odd n  = exp' (a*m) (m*m) (p/2)     -- exp'.3
```

Above we showed that `exp a m n = exp' a m n`. However, execution of `exp'` converges faster upon the result: $\mathcal{O}(\log_2 \mathtt{n})$ steps rather than $\mathcal{O}(\mathtt{n})$.

Note: Multiplication and division of integers by natural number powers of 2, particularly 2^1, can be implemented on must current computers by arithmetic left and right shifts, respectively, which are faster than general multiplication and division.

12.6 Text Processing Example

Reference: In this section we develop a text processing package similar to the one in Section 4.3 of the Bird and Wadler textbook [2]. The text processing package in the Haskell standard prelude is slightly different in its treatment of newline characters.

A textual document can be viewed in many different ways. At the lowest level, we can view it as just a character string and define a type synonym as follows:

```
type Text = String
```

However, for other purposes, we may want to consider the document as having more structure, i.e., view it as a sequence of words, lines, paragraphs, pages, etc. We sometimes want to convert the text from one view to another.

12.6.1 Line processing

Consider the problem of converting a `Text` document to the corresponding sequence of lines. Suppose that in the `Text` document, the newline characters (`'\n'`) serve as *separators* of lines, not themselves part of the lines. Since each line is a sequence of characters, we define a type synonym `Line` as follows:

```
type Line = String
```

We want a function `lines'` that will take a `Text` document and return the corresponding sequence of lines in the document. The function has the type signature:

```
lines' :: Text -> [Line]
```

Example: `lines' "This has\nthree   \nlines"`
$\Longrightarrow$ `["This has", "three   ", "lines"]`

Writing function `lines'` is not trivial. However, its inverse `unlines'` is quite easy. Function `unlines'` takes a list of `Line`s, inserts a newline character between each pair of adjacent lines, and returns the `Text` document resulting from the concatenation.

```
unlines' :: [Line] -> Text
```

Let's see if we can develop `lines'` from `unlines'`.

The basic computational pattern for function `unlines'` is a folding operation. Since we are dealing with the construction of a list and the list constructors are nonstrict in their right arguments, a `foldr` operation seems more appropriate than a `foldl` operation. (See Section 13.6.)

To use `foldr`, we need a binary operation that will append two lines with a newline character inserted between them. The following, a bit more general, operation `insert'` will do that for us. The first argument is the element that is to be inserted between the two list arguments.

```
insert' :: a -> [a] -> [a] -> [a]
insert' a xs ys  = xs ++ [a] ++ ys  -- insert.1
```

Informally, it is easy to see that `insert'` is an associative operation but that it has no right (or left) identity element.

Since `insert'` has no identity element, there is no obvious "seed" value to use with `foldr`. Thus we will need to find a different way to express `unlines'`.

If we restrict the domain of `unlines'` to non-nil lists of lines, then we can use `foldr1`, a right-folding operation defined over non-empty lists. This function does not require an identity element for the operation. Function `foldr1` is defined in the standard prelude as follows:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Note: There is a similar function, `foldl1`, that takes a non-nil list and does a left-folding operation.

Thus we can now define `unlines'` as follows:

```
unlines' :: [Line] -> Text
unlines' xss = foldr1 (insert' '\n') xss
```

Given the definition of `unlines'`, we can now specify what we want `lines'` to do. It must satisfy the following specification for any *non-nil* `xss` of type `[Line]`:

```
lines' (unlines' xss) = xss
```

That is, `lines'` is the inverse of `unlines'` for all non-nil arguments.

Our first step in the synthesis of `lines'` is to guess at a possible structure for the `lines'` function definition. Then we will attempt to calculate the unknown pieces of the definition.

Since `unlines'` uses a right-folding operation, it is reasonable to guess that its inverse will also use a right-folding operation. Thus we speculate that `lines'` can be defined as follows, given an appropriately defined operation `op` and "seed value" `a`.

```
lines' :: Text -> [Line]
lines' = foldr op a
```

Because of the definition of `foldr` and type signature of `lines'`, function `op` must have the type signature:

```
op :: Char -> [Line] -> [Line]
```

and `a` must be the right identity of `op` and hence have type `[Line]`. Our task now is to find appropriate definitions for `op` and `a`.

From what we know about `unlines'`, `foldr1`, `lines'`, and `foldr` we see that the following identities hold. (These can be proved, but we do not do so here.)

```
unlines' [xs]        = xs                              -- unlines.1
unlines' ([xs]++xss) = insert' '\n' xs (unlines' xss) -- unlines.2

lines' []            = a                               -- lines.1
lines' ([x]++xs)     = op x (lines' xs)                -- lines.2
```

Note the names we give each of the above identities (e.g., `unlines.1`). We use these equations to justify our steps in the calculations below.

Next, let us calculate the unknown identity element `a`. Our strategy is to transform `a` by use of the definition and derived properties for `unlines'` and the specification and derived properties for `lines'` until we arrive at a constant.

```
   a
=     { lines.1 (right to left) }
   lines' []
=     { unlines'.1 (right to left) with xs = [] }
   lines' (unlines' [[]])
=     { specification of lines' (left to right) }
   [[]]
```

Therefore we define `a` to be `[[]]`. Note that because of `lines.1`, we have also defined `lines'` in the case where its argument is `[]`.

Now we proceed to calculate a definition for `op`. Remember that we assume `xss` $\neq$ `[]`.

As above, our strategy is use what we know about `unlines'` and what we have assumed about `lines'` to calculate appropriate definitions for the unknown parts of the definition of `lines'`. We first expand our expression to bring in `unlines'`.

```
   op x xss
=     { specification for lines' (right to left) }
   op x (lines' (unlines' xss))
=     { lines.2 (right to left) }
   lines' ([x] ++ unlines' xss)
```

Since there seems no other way to proceed with our calculation, we distinguish between cases for the variable `x`. In particular, we consider the case where `x` is the line separator and the case where it is not, i.e., `x = '\n'` and `x ≠ '\n'`.

Case `x = '\n'`

Our strategy is to absorb the `"\n"` into the `unlines'`, then apply the specification of `lines'`.

```
   lines' ("\n" ++ unlines' xss)
=     { [] is the identity for ++ }
   lines' ([] ++ "\n" ++ unlines' xss)
=     { insert.1 (right to left) with a == '\n' }
   lines' (insert' '\n' [] (unlines' xss))
=     { unlines.2 (right to left) }
   lines' (unlines' ([[]] ++ xss))
=     { specification of lines' (left to right) }
   [[]] ++ xss
```

Thus `op '\n' xss = [[]] ++ xss`,

Case `x ≠ '\n'`

Our strategy is to absorb the `[x]` into the `unlines'`, then apply the specification of `lines'`.

```
   lines' ([x] ++ unlines' xss)
=     { Assumption xss ≠ [], let xss = [ys] ++ yss }
   lines' ([x] ++ unlines' ([ys] ++ yss))
=     { unlines.2 (left to right) with a = '\n' }
   lines' ([x] ++ insert' '\n' ys (unlines' yss))
=     { insert.1 (left to right) }
   lines' ([x] ++ (ys ++ "\n" ++ unlines' yss))
=     { ++ associativity }
   lines' (([x] ++ ys) ++ "\n" ++ unlines' yss)
=     { insert.1 (right to left) }
   lines' (insert' '\n' ([x]++ys) (unlines' yss))
=     { unlines.2 (right to left) }
   lines' (unlines' ([[x]++ys] ++ yss))
=     { specification of lines' (left to right) }
   [[x]++ys] ++ yss
```

Thus, for `x ≠ '\n'` and `xss ≠ []`,
`op x xss = [[x] ++ head xss] ++ (tail xss)`.

To generalize `op` like we did `insert'` and give it a more appropriate name, we define `op` to be `breakOn '\n'` as follows:

```
breakOn :: Eq a => a -> a -> [[a]] -> [[a]]
breakOn a x []              = error "breakOn applied to nil"
breakOn a x xss | a == x    = [[]] ++ xss
                | otherwise = [[x] ++ ys] ++ yss
                              where (ys:yss) = xss
```

Thus, we get the following definition for `lines'`:

```
lines' :: Text -> [Line]
lines' xs = foldr (breakOn '\n') [[]] xs
```

Recap: We have synthesized `lines'` from its specification and the definition for `unlines'`, its inverse. Starting from a precise, but non-executable specification, and using only equational reasoning, we have derived an executable definition of the required function. The technique used is a familiar one in many areas of mathematics: first we guessed at a form for the solution, and then we calculated the unknowns.

Note: The definition of `lines` and `unlines` in the standard prelude treat newlines as line *terminators* instead of line separators. Their definitions follow.

```
lines :: String -> [String]
lines "" = []
lines s  = l : (if null s' then [] else lines (tail s'))
           where (l, s') = break ('\n'==) s

unlines :: [String] -> String
unlines  = concat . map (\l -> l ++ "\n")
```

12.6.2 Word processing

Let's continue the text processing example from the previous subsection a bit further. We want to synthesize a function to break a text into a sequence of words.

For the purposes here, we define a word as any nonempty sequence of characters not containing a space or newline character. That is, a group of one or more spaces and newlines separate words. We introduce a type synonym for words.

```
type Word = String
```

We want a function `words'` that breaks a line up into a sequence of words. Function `words'` thus has the following type signature:

```
words' :: Line -> [Word]
```

Example: `words' "Hi there"` $\Longrightarrow$ `["Hi", "there"]`

As in the synthesis of `lines'`, we proceed by defining the "inverse" function first, then we calculate the definition for `words'`.

All `unwords'` needs to do is to insert a space character between adjacent elements of the sequence of words and return the concatenated result. Following the development in the previous subsection, we can thus define `unwords'` as follows.

```
unwords' :: [Word] -> Line
unwords' xs = foldr1 (insert'  ' ') xs
```

Using calculations similar to those for `lines'`, we derive the inverse of `unwords'` to be the following function:

```
foldr (breakOn  ' ') [[]]
```

However, this identifies zero-length words where there are adjacent spaces. We need to filter those out.

```
words' :: Line -> [Word]
words' = filter (/= []) . foldr (breakOn  ' ') [[]]
```

Note that `words' (unwords' xss)` = `xss` for all `xss` of type `[Word]`, but that `unwords' (words' xs)` $\neq$ `xs` for some `xs` of type `Line`. The latter is undefined when `words' xs` returns `[]`. Where it is defined, adjacent spaces in `xs` are replaced by a single space in `unwords' (words' xs)`.

Note: The functions `words` and `unwords` in the standard prelude differ in that `unwords [] = []`, which is more complete.

12.6.3 Paragraph processing

Let's continue the text processing example one step further and synthesize a function to break a sequence of lines into paragraphs.

For the purposes here, we define a paragraph as any nonempty sequence of nonempty lines. That is, a group of one or more empty lines separate paragraphs. As above, we introduce an appropriate type synonym:

```
type Para = [Line]
```

We want a function `paras'` that breaks a sequence of lines into a sequence of paragraphs:

```
paras' :: [Line] -> [Para]
```

Example: `paras' ["Line 1.1","Line 1.2","","Line 2.1"]`
$\Longrightarrow$ `[["Line 1.1","Line 1.2"],["Line 2.1"]]`

As in the synthesis of `lines'` and `words'`, we can start with the inverse and calculate the definition of `paras'`. The inverse function `unparas'` takes a sequence of paragraphs and returns the corresponding sequence of lines with an empty line inserted between adjacent paragraphs.

```
unparas' :: [Para] -> [Line]
unparas' = foldr1 (insert' [])
```

Using calculations similar to those for `lines'` and `words'`, we can derive the following definitions:

```
paras' :: [Line] -> [Para]
paras' = filter (/= []) . foldr (breakOn []) [[]]
```

The `filter (/= [])` operation removes all "empty paragraphs" corresponding to two or more adjacent empty lines.

Note: There are no equivalents of `paras'` and `unparas'` in the standard prelude. As with `unwords`, `unparas'` should be redefined so that `unparas' [] = []`, which is more complete.

12.6.4 Other text processing functions

Using the six functions in our text processing package, we can build other useful functions.

Count the lines in a text.

```
countLines :: Text -> Int
countLines = length . lines'
```

Count the words in a text.

```
countWords :: Text -> Int
countWords = length . concat . (map words') . lines'
```

An alternative using a list comprehension is:

```
countWords xs = length [ w | l <- lines' xs, w <- words' l]
```

Count the paragraphs in a text.

```
countParas :: Text -> Int
countParas = length . paras' . lines'
```

Normalize text by removing redundant empty lines and spaces.

The following functions take advantage of the fact that `paras'` and `words'` discard empty paragraphs and words, respectively.

```
normalize :: Text -> Text
normalize = unparse . parse

parse :: Text -> [[[Word]]]
parse = (map (map words')) . paras' . lines'

unparse :: [[[Word]]] -> Text
unparse = unlines' . unparas' . map (map unwords')
```

We can also state `parse` and `unparse` in terms of list comprehensions.

```
parse xs = [ [words' l | l <- p] | p <- paras' (lines' xs) ]

unparse xssss =
      unlines' (unparas' [ [unwords' l | l<-p] | p<-xssss])
```

Section 4.3.5 of the Bird and Wadler textbook goes on to build functions to fill and left-justify lines of text.

12.7 Exercises

1. The following function computes the integer base 2 logarithm of a positive integer:

```
lg :: Int -> Int
lg x | x == 1  = 0
     | x > 1   = 1 + lg (x/2)
```

   Using the tail recursion theorem, write a definition for `lg` that is tail recursive.

2. Synthesize the recursive definition for `++` from the following specification:

```
xs ++ ys = foldr (:) ys xs
```

3. Using tupling and function `fact5` from Section 3, synthesize an efficient function `allfacts` to generate a list off factorials for natural numbers 0 through parameter `n`, inclusive.

4. Consider the following recursive definition for natural number multiplication:

```
mul :: Int -> Int -> Int
mul m 0     = 0
mul m (n+1) = m + mul m n
```

   This is an $\mathcal{O}(\mathtt{n})$ algorithm for computing `m * n`. Synthesize an alternative operation that is $\mathcal{O}(\log_2 \mathtt{n})$. Doubling (i.e., `n*2`) and halving (i.e, `n/2` with truncation) operations may be used but not multiplication (`*`) in general.

5. Derive a "more general" version of the Tail Recursion Theorem for functions of the shape

```
func :: X -> Y
func x | not (b x) = f x                          -- func.1
       | b x       = h x ⊙ func (g x) ♢ d.x -- func.2
```

   where functions `b`, `f`, `g`, and `h` are constrained as in the definition of `fun` in the Tail Recursion Theorem. Be sure to identify the appropriate constraints on `d`, $\odot$, and $\diamondsuit$ including the necessary properties of $\odot$ and $\diamondsuit$.

13 MODELS OF REDUCTION

Reference: This section is based in part on Sections 6.1–6.3 of the Bird and Wadler textbook [2] and in part on Chapter 6 of Field and Harrison's textbook [8].

13.1 Efficiency

We state efficiency (i.e., time complexity or space complexity) of programs in terms of the "Big-O" notation and asymptotic analysis.

For example, consider the list-reversing functions `rev` and `reverse` that we have looked at several times. We stated that the number of steps required to evaluate `rev xs` is, in the worst case, "on the order of" n^2 where n denotes the length of list `xs`. We let the number of steps be our measure of time and write

$$T_{\texttt{rev}}(\texttt{xs}) = \mathcal{O}(n^2)$$

to mean that the time to evaluate `rev xs` is bounded by some (mathematical) function that is proportional to the square of the length of list `xs`.

Similarly, we write

$$T_{\texttt{reverse}}(\texttt{xs}) = \mathcal{O}(n)$$

to mean that the time (i.e., number of steps) to evaluate `reverse xs` is bounded by some function that is proportional to the length of `xs`.

Note: These expressions are not really equalities. We write the more precise expression, (e.g., $T_{\texttt{reverse}}(\texttt{xs})$) on the left-hand side and the less precise expression $\mathcal{O}(n)$ on the right-hand side.

For short lists, the performance of `rev` and `reverse` are similar. But as the list gets long, `rev` requires considerably more steps than `reverse`.

The Big-O analysis is an asymptotic analysis. That is, it estimates the order of magnitude of the evaluation time as the size of the input approaches infinity (gets large). We often do worst case analyses of time. Such analyses are usually easier to do than average-case analyses.

13.2 Reduction

The terms *reduction*, *simplification*, and *evaluation* all denote the same process: rewriting an expression in a "simpler" equivalent form. That is, they involve two kinds of replacements:

- the replacement of a subterm that satisfies the left-hand side of an equation by the right-hand side with appropriate substitution of arguments for parameters. (This is sometimes called β-reduction.)
- the replacement of a primitive application (e.g., `+` or `*`) by its value. (This is sometimes called δ-reduction.)

The term *redex* refers to a subterm of an expression that can be reduced.

An expression is said to be in *normal form* if it cannot be further reduced.

Some expressions cannot be reduced to a value. For example, `1/0` cannot be reduced; an error message is usually generated if there is an attempt to evaluate (i.e., reduce) such an expression.

For convenience, we sometimes assign the value $\perp$ (pronounced "bottom") to such error cases to denote that their values are undefined. Remember that this value cannot be manipulated within a computer.

Redexes can be selected for reduction in several ways. For instance, the redex can be selected based on its position within the expression:

leftmost redex first, the leftmost reducible subterm in the expression text is reduced before any other subterms are reduced.

rightmost redex first, the rightmost reducible subterm in the expression text is reduced before any other subterms are reduced.

The redex can also be selected based on whether or not it is contained within another redex:

outermost redex first, a reducible subterm that is not contained within any other reducible subterm is reduced before one that is contained within another.

innermost redex first, a reducible subterm that contains no other reducible subterm is reduced before one that contains others.

The two most often used reduction orders are:

applicative order reduction (AOR), where the leftmost innermost redex is reduced first.

normal order reduction (NOR), where the leftmost outermost redex is reduced first.

To see the difference between AOR and NOR consider the following functions:

```
fst :: (a,b) -> a
fst (x,y) = x

sqr :: Int -> Int
sqr x     = x * x
```

Now consider the following reductions.

AOR:

```
   fst (sqr 4, sqr 2)
⟹   { sqr }
   fst (4*4, sqr 2)
⟹   { * }
   fst (16, sqr 2)
⟹   { sqr }
   fst (16, 2*2)
⟹   { * }
   fst (16, 4)
⟹   { fst }
   16
```

AOR requires 5 reductions.

NOR:

```
   fst (sqr 4, sqr 2)
⟹   { fst }
   sqr 4
⟹   { sqr }
   4*4
⟹   { * }
   16
```

NOR requires 3 reductions.

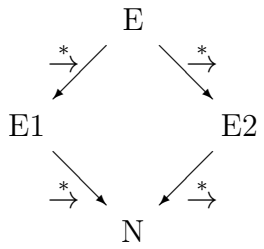
In this example NOR requires fewer steps because it avoids reducing the unneeded second component of the tuple.

The number of reductions is different, but the result is the same for both reduction sequences.

In fact, this is always the case. If any reduction terminates (and not all do), then the resulting value will always be the same.

(Consequence of) Church-Rosser Theorem: If an expression can be reduced in two different ways to two normal forms, then these normal forms are the same (except that variables may need to be renamed).

The *diamond property* for the reduction relation $\rightarrow$ states that if an expression `E` can be reduced to two expressions `E1` and `E2`, then there is an expression `N` which can be reached (by repeatedly applying $\rightarrow$) from both `E1` and `E2`. We use the symbol $\stackrel{*}{\rightarrow}$ to represent the *reflexive transitive closure* of $\rightarrow$. (`E` $\stackrel{*}{\rightarrow}$ `E1` means that `E` can be reduced to `E1` by some finite, possibly zero, number of reductions.)



Some reduction orders may fail to terminate on some expressions. Consider the following functions:

```
answer :: Int -> Int
answer n = fst (n+n, loop n)

loop :: Int -> [a]
loop n   = loop (n+1)
```

AOR:

```
   answer 1
⟹   { answer  }
   fst (1+1,loop 1)
⟹   { + }
   fst (2,loop 1)
⟹   { loop }
   fst (2,loop (1+1))
⟹   { + }
   fst (2,loop 2)
⟹   { loop }
   fst (2,loop (2+1))
⟹   { + }
   fst (2,loop 3)
⟹ ··· Does not terminate normally
```

NOR:

```
   answer 1
⟹   { answer  }
   fst (1+1,loop 1)
⟹   { fst }
   1+1
⟹   { + }
   2
```

NOR requires 3 reductions.

An Important Property: If an expression `E` has a normal form, then a normal order reduction of `E` (i.e., leftmost outermost) is guaranteed to reach the normal form (except that variables may need to be renamed).

A few related concepts:

- **Applicative order reduction.** Reduce leftmost innermost redex first.

  Eager evaluation. Evaluate any expression that can be evaluated regardless of whether the result is ever needed. (For example, arguments of a function are evaluated before the function is called.)

  Strict semantics. A function is only defined if all of its arguments are defined. For example, multiplication is only defined if both of its operands are defined, `5 * ⊥ = ⊥` .

  Call-by-value parameter passing. Evaluate the argument expression and bind its value to the function's parameter.

- **Normal order reduction.** Reduce leftmost outermost redex first.

  Lazy evaluation. Do not evaluate an expression unless its result is needed.

  Nonstrict (lenient) semantics. A function may have a value even if some of its arguments are undefined. For example, tuple construction is not strict in either parameter, `(⊥,x)` $\neq \perp$ and `(x,⊥)` $\neq \perp$

  Call-by-name parameter passing. Pass the unevaluated argument expression to the function; evaluate it upon each reference.

  Note that in the absence of side-effects (e.g., when we have referential transparency), call-by-name gives the same result as call-by-value.

In general, call-by-name parameter passing is inefficient. However, a referentially transparent language can replace call-by-name parameter passing with the equivalent, but more efficient, *call-by-need* method.

In the call-by-need method, the unevaluated argument expression is passed to the function as in call-by-name. The first reference to the corresponding parameter causes the expression to be evaluated; subsequent references just use the value computed by the first reference. Thus the expression is only evaluated when needed and then only once.

Consider the `sqr` program again.

```
sqr x = x * x
```

AOR:

```
   sqr (4+2)
⟹   { + }
   sqr 6
⟹   { sqr }
   6 * 6
⟹   { * }
   36
```

AOR requires 3 reductions.

NOR:

```
   sqr (4+2)
⟹   { sqr }
   (4+2) * (4+2)
⟹   { + }
   6 * (4+2)
⟹   { + }
   6 * 6
⟹   { * }
   36
```

NOR requires 4 reductions.

Here NOR is less efficient than AOR. What is the problem?

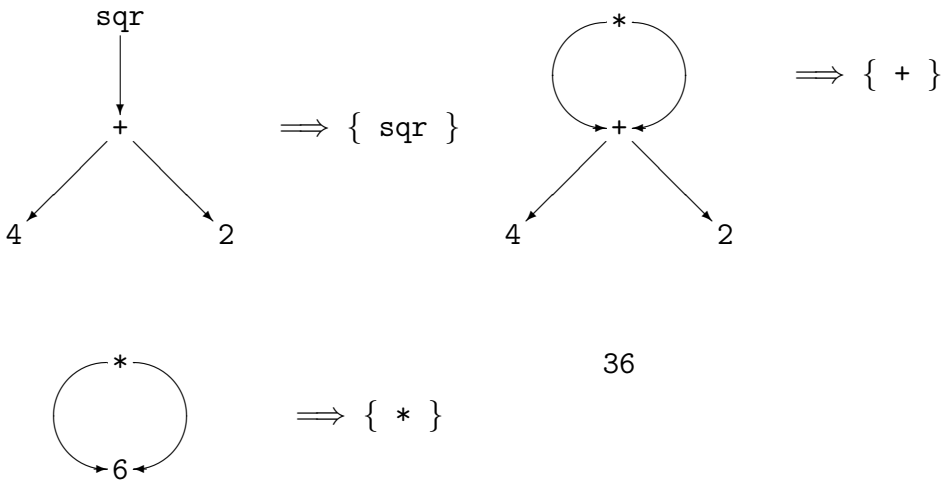
The argument `(4+2)` is reduced twice because the parameter appeared twice on the right-hand side of the definition.

The rewriting strategy we have been using so far can be called *string reduction* because our model involves the textual replacement of one string by an equivalent string.

A more efficient alternative is *graph reduction*. In this technique, the expressions are represented as (directed acyclic) expression graphs rather than text strings. The repeated subterms of an expression are represented as shared components of the expression graph. Once a shared component has been evaluated, it need not be evaluated again. Thus leftmost outermost (i.e., normal order) graph reduction is a technique for implementing call-by-need parameter passing.

The Haskell interpreter uses a graph reduction technique.

Consider the leftmost outermost graph reduction of the expression `sqr (4+2)`.



Note: In a graph reduction model, normal order reduction never performs more reduction steps than applicative order reduction. It may perform fewer. And, like all outermost reduction techniques, it is guaranteed to terminate if any reduction sequence terminates.

As we see above, parameters that repeatedly occur on the right-hand side introduce shared components into the expression graph. A programmer can also introduce shared components into a function's expression graph by using `where` or `let` to define new symbols for subexpressions that occur multiple times in the defining expression. This potentially increases the efficiency of the program .

Consider a program to find the solutions of the following equation:

$$a * x^2 + b * x + c = 0$$

Using the quadratic formula the two solutions are:

$$\frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$$

Expressing this formula as a Haskell program to return the two solutions as a pair, we get:

```
roots :: Float -> Float -> Float -> (Float,Float)
roots a b c = ( (-b-d)/e, (-b+d)/e )
              where d = sqrt (sqr b - 4 * a * c)
                    e = 2 * a
```

Note the explicit definition of local symbols for the subexpressions that occur multiple times.

Function `sqr` is as defined previously and `sqrt` is a primitive function defined in the standard prelude.

In one step, the expression `roots 1 5 3` reduces to the expression graph shown on the following page. For clarity, we use the following in the graph:

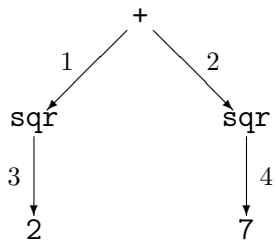
- `tuple-2` denotes the pair forming operator `( , )`.
- `div` denotes division (on `Float`).
- `sub` denotes subtraction.
- `neg` denotes unary negation.

The application `roots 1 5 3` reduces to the following expression graph:

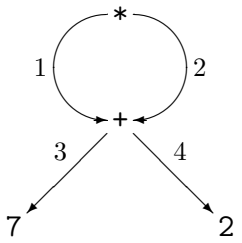
(Drawing Not Currently Available)

We use the total number of *arguments* as the measure of the *size* of a term or graph.

Example: `sqr 2 + sqr 7` has size 4.



Example: `x * x where x = 7 + 2` has size 4.



Note: This size measure is an indication of the size of the unevaluated expression that is held at a particular point in the evaluation process. This is a bit different from the way we normally think of space complexity in an imperative algorithms class, that is, the number of "words" required to store the program's data.

However, this is not as strange as it may first appear. Remember that data structures such as lists and tuples are themselves *expressions* built by applying constructors to simpler data.

13.3 Head Normal Form

Sometimes we need to reduce a term but not all the way to normal form.

Consider the expression `head (map sqr [1..7])` and a normal order reduction.

```
   head (map sqr [1..7])
⟹   { [1..7] }
   head (map sqr (1:[2..7]))
⟹   { map.2 }
   head (sqr 1 : map sqr [2..7])
⟹   { head }
   sqr 1
⟹   { sqr }
   1 * 1
⟹   { * }
   1
```

Note that the expression `map sqr [1..7]` was reduced but not all the way to normal form. However, any term that is reduced must be reduced to *head normal form*.

Definition: A term is in *head normal form* if:

- it is not a redex,
- it cannot become a redex by reducing any of its subterms.

If a term is in normal form, then it is in head normal form, but not vice versa.

Any term of form `(e1:e2)` is in head normal form, because regardless of how far `e1` and `e2` are reduced, no reduction rule applies to `(e1:e2)`. The cons operator is the primitive list constructor; it is not defined in terms of anything else.

However, a term of form `(e1:e2)` is only in normal form if both `e1` and `e2` are in their normal forms.

Similarly, any term of the form `(e1,e2)` is in head normal form. The tuple constructor is a primitive operation; it is not defined in terms of anything else.

However, a term of the form `(e1,e2)` is in normal form only if both `e1` and `e2` are.

Whether a term needs to be reduced further than head normal form depends upon the context.

Example: In the reduction of the expression `head (map sqr [1..7])`, the term `map sqr [1..7]` only needed to be reduced to head normal form, that is, to the expression `sqr 1 : map sqr [2..7]`.

However, `appendChan stdout (show (map sqr [1..7])) exit done` would cause reduction of `map sqr [1..7]` to normal form.

13.4 Pattern Matching

For reduction using equations that involve pattern matching, the leftmost outermost (i.e., normal order) reduction strategy is not, by itself, sufficient to guarantee that a terminating reduction sequence will be found if one exists.

Consider function `zip'`.

```
zip' :: [a] -> [b] -> [(a,b)]
zip' (a:as) (b:bs) = (a,b) : zip' as bs
zip' _      _      = []
```

Now consider a leftmost outermost (i.e., normal order) reduction of the expression `zip' (map sqr []) (loop 0)` where `sqr` and `loop` are as defined previously.

```
   zip' (map sqr []) (loop 0)
⟹   { map.1, to determine if first arg matches (a:as)}
   zip' [] (loop 0)
⟹   { zip'.2 }
   []
```

Alternatively, consider a rightmost outermost reduction of the same expression.

```
   zip' (map sqr []) (loop 0)
⟹   { loop, to determine if second arg matches (b:bs)}
   zip' (map sqr []) (loop (0+1))
⟹   { + }
   zip' (map sqr []) (loop 1)
⟹   { loop }
   zip' (map sqr []) (loop (1+1))
⟹   { + }
   zip' (map sqr []) (loop 2)
⟹   ···    Does not terminate normally
```

Pattern matching should not cause an argument to be reduced unless absolutely necessary; otherwise nontermination could result.

Pattern-matching reduction rule: Match the patterns left to right. Reduce a subterm only if required by the pattern.

In `zip' (map sqr []) (loop 0)` the first argument must be reduced to head normal form to determine whether it matches `(a:as)` for the first leg of the definition. It is not necessary to reduce the second argument unless the first argument match is successful.

Note that the second leg of the definition, which uses two anonymous variables for the patterns, does not require any further reduction to occur in order to match the patterns.

Expressions

```
zip' (map sqr [1,2,3]) (map sqr [1,2,3])
```

and

```
zip' (map sqr [1,2,3]) []
```

both require their second arguments to be reduced to head normal form in order to determine whether the arguments match `(b:bs)`.

Note that the first does match and, hence, enables the first leg of the definition to be used in the reduction. The second expression does not match and, hence, disables the first leg from being used. Since the second leg involves anonymous patterns, it can be used in this case.

Our model of computation:

- Normal order graph reduction $\mathtt{e}_0 \Longrightarrow \mathtt{e}_1 \Longrightarrow \mathtt{e}_2 \Longrightarrow \cdots \Longrightarrow \mathtt{e}_n$
- Time = number of reduction steps (n)
- Space = size of the largest expression graph $\mathtt{e}_i$

Most lazy functional language implementations more-or-less correspond to graph reduction.

13.5 Reduction Order and Space

It is always the case that the number of steps in an *outermost graph reduction* $\leq$ the number of steps in an *innermost reduction* of the same expression.

However, sometimes a combination of innermost and outermost reductions can save on space and, hence, on implementation overhead.

Consider the following definition of the factorial function. (This was called `fact3` earlier in these notes.)

```
fact ::  Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Now consider a normal order reduction of the expression `fact 3`.

```
   fact 3
⟹   { fact.2 }
   3 * fact (3-1)
⟹   { -, to determine pattern match }
   3 * fact 2
⟹   { fact.2 }
   3 * (2 * fact (2-1))
⟹   { -, to determine pattern match }
   3 * (2 * fact 1)
⟹   { fact.2 }
   3 * (2 * (1 * fact (1-1)))        MAX SPACE!
⟹   { -, to determine pattern match }
   3 * (2 * (1 * fact 0))
⟹   { fact.1 }
   3 * (2 * (1 * 1))
⟹   { * }
   3 * (2 * 1)
⟹   { * }
   3 * 2
⟹   { * }
   6
```

Time: Count reduction steps. 10 for this example.

In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3n+1` reductions. $\mathcal{O}(\texttt{n})$.

Space: Count arguments in longest expression. 4 binary operations, 1 unary operation, hence size is 9 for this example.

In general, 1 multiplication for each `n > 0` plus 1 subtraction and one application of `fact`. Thus `2n + 3` arguments. $\mathcal{O}(\texttt{n})$.

Note that function `fact` is strict in its argument. That is, evaluation of `fact` *always* requires the evaluation of its argument.

Since the value of the argument expression `n-1` in the recursive call is eventually needed (by the pattern match), there is no reason to delay evaluation of the expression. That is, the expression could be evaluated eagerly instead of lazily. Thus any work to save this expression for future evaluation would be avoided.

Delaying the computation of an expression incurs overhead in the implementation. The delayed expression and its calling environment (i.e., the values of variables) must be packaged so that evaluation can resume correctly when needed. This packaging—called a *closure*, *suspension*, or *recipe*—requires both space and time to be set up.

Furthermore, delayed expressions can aggravate the problem of *space leaks*.

The implementation of a lazy functional programming language typically allocates space for data dynamically from a memory *heap*. When the heap is exhausted, the implementation searches through its structures to recover space that is no longer in use. This process is usually called *garbage collection*.

However, sometimes it is very difficult for a garbage collector to determine whether or not a particular data structure is still needed. The garbage collector thus retains some unneeded data. These are called space leaks.

Aside: Picture bits of memory oozing out of the program, lost to the program forever. Most of these bits collect in the bit bucket under the computer and are automatically recycled when the interpreter restarts. However, in the past a few of these bits leaked out into the air, gradually polluting the atmosphere of functional programming research centers. Although it has not be scientifically verified, anecdotal evidence suggests that the bits leaked from functional programs, when exposed to open minds, metamorphose into a powerful intellectual stimulant. Many imperative programmers have observed that programmers who spend a few weeks in the vicinity of functional programs seem to develop a permanent distaste for imperative programs and a strange enhancement of their mental capacities.

Aside continued: As environmental awareness has grown in the functional programming community, the implementors of functional languages have begun to develop new leak-avoiding designs for the language processors and garbage collectors. Now the amount of space leakage has been reduced considerably. Although it is still a problem. Of course, in the meantime a large community of programmers have become addicted to the intellectual stimulation of functional programming. The number of addicts in the USA is small, but growing. FP traffickers have found a number of ways to smuggle their illicit materials into the country. Some are brought in via the Internet from clandestine archives in Europe; a number of professors and students are believed to be cultivating a domestic supply. Some are smuggled from Europe

inside strange red-and-white covered books (but that source is somewhat lacking in the continuity of supply). Some are believed hidden in Haskell holes; others in a young nerd named Haskell's pocket protector. (Haskell is Miranda's younger brother; she was the first one who had any comprehension about FP.)

Aside ends: Mercifully.

Now let's look at a tail recursive definition of factorial.

```
fact' :: Int -> Int -> Int
fact' f 0 = f
fact' f n = fact' (f*n) (n-1)
```

Because of the Tail Recursion Theorem, we know that `fact' 1 n = fact n` for any natural `n`.

Now consider a normal order reduction of the expression `fact' 1 3`.

```
   fact' 1 3
⟹   { fact'.2 }
   fact' (1 * 3) (3 - 1)
⟹   { -, to determine pattern match }
   fact' (1 * 3) 2
⟹   { fact'.2 }
   fact' ((1 * 3) * 2) (2 - 1)
⟹   { -, to determine pattern match }
   fact' ((1 * 3) * 2) 1
⟹   { fact'.2 }
   fact' (((1 * 3) * 2) * 1) (1 - 1)        MAX SPACE!
⟹   { -, to determine pattern match }
   fact' (((1 * 3) * 2) * 1) 0
⟹   { fact'.1 }
   ((1 * 3) * 2) * 1
⟹   { * }
   (3 * 2) * 1
⟹   { * }
   6 * 1
⟹   { * }
   6
```

Time: Count reduction steps. 10 for this example, same as for `fact`.

In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3n+1` reductions. $\mathcal{O}(\mathtt{n})$.

Space: Count arguments in longest expression. 4 binary operations, 1 two-argument function, hence size is 10 for this example.

In general, 1 multiplication for each `n > 0` plus 1 subtraction and one application of `fact'`. Thus `2n + 4` arguments. $\mathcal{O}(\texttt{n})$.

Note that function `fact'` is strict in both arguments. The second argument of `fact'` is evaluated immediately because of the pattern matching. The first argument's value is eventually needed, but its evaluation is deferred until after the `fact'` recursion has reached its base case.

Perhaps we can improve the space efficiency by forcing the evaluation of the first argument immediately as well. In particular, we try a combination of outermost and innermost reduction.

```
   fact' 1 3
⟹    { fact'.2 }
   fact' (1 * 3) (3 - 1)
⟹    { *, innermost }
   fact' 3 (3 - 1)
⟹    { -, to determine pattern match }
   fact' 3 2
⟹    { fact'.2 }
   fact' (3 * 2) (2 - 1)
⟹    { *, innermost }
   fact' 6 (2 - 1)
⟹    { -, to determine pattern match }
   fact' 6 1
⟹    { fact'.2 }
   fact' (6 * 1) (1 - 1)
⟹    { *, innermost }
   fact' 6 (1 - 1)
⟹    { -, to determine pattern match }
   fact' 6 0
⟹    { fact'.1 }
   6
```

Time: Count reduction steps. 10 for this example. Same as for previous two reduction sequences.

In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3n+1` reductions. $\mathcal{O}(\texttt{n})$.

Space: Count arguments in longest expression.

For any `n > 0`, the longest expression consists of one multiplication, one subtraction, and one call of `fact'`. Thus the size is constantly 6. $\mathcal{O}(1)$.

Problem: How to decrease space usage and implementation overhead.

Solutions:

1. The compiler could do *strictness analysis* and automatically force eager evaluation of arguments that are always required.

   This is done by many compilers. It is sometimes a complicated procedure.

2. The language could be extended with a feature that allows the programmer to express strictness explicitly.

In Haskell, reduction order can be controlled by use of the special function `strict`.

A term of the form `strict f e` is reduced by first reducing expression `e` to head normal form, and then applying function `f` to the result. The term `e` can be reduced by normal order reduction, unless, of course, it contains another call of `strict`.

The following definition of `fact'` gives the mixed reduction order given in the previous example. That is, it evaluates the first argument eagerly to save space.

```
fact' :: Int -> Int -> Int
fact' f 0 = f
fact' f n = (strict fact' (f*n)) (n-1)
```

13.6 Choosing a Fold

Remember that earlier we defined two folding operations. Function `foldr` is a backward linear recursive function that folds an operation through a list from the tail (i.e., right) toward the head. Function `foldl` is a tail recursive function that folds an operation through a list from the head (i.e., left) toward the tail.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs)   = f x (foldr f z xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs
```

The first duality theorem (as given in the Bird and Wadler textbook) states the circumstances in which one can replace `foldr` by `foldl` and vice versa.

First duality theorem: If $\oplus$ is a associative binary operation of type `t -> t` with identity element `z`, then:

```
foldr (⊕) z xs = foldl (⊕) z xs
```

Thus, often we can use either `foldr` or `foldl` to solve a problem. Which is better?

We discussed this problem before, but now we have the background to understand it a bit better.

Clearly, eager evaluation of the second argument of `foldl`, which is used as an accumulating parameter, can increase the space efficiency of the folding operation. This optimized operation is called `foldl'` in the standard prelude.

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z []     = z
foldl' f z (x:xs) = strict (foldl' f) (f z x) xs
```

Suppose that `op` is *strict in both arguments* and can be computed in $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ space. (For example, `+` and `*` have these characteristics.) If `n = length xs`, then both `foldr op i xs` and `foldl op i xs` can be computed in $\mathcal{O}(\mathtt{n})$ time and $\mathcal{O}(\mathtt{n})$ space.

However, `foldl' op i xs` requires $\mathcal{O}(\mathtt{n})$ time and $\mathcal{O}(1)$ space. The reasoning for this is similar to that given for `fact'`.

Thus, in general, `foldl'` is the better choice for this case.

Alternatively, suppose that `op` is *nonstrict in either argument*. Then `foldr` is usually more efficient than `foldl`.

As an example, consider operation `||` (i.e., logical-or). The `||` operator is strict in the first argument, but not in the second. That is, `True || x = True` without having to evaluate `x`.

Let `xs = [`$\mathtt{x}_1$`, `$\mathtt{x}_2$`, `$\mathtt{x}_3$`, ⋯, `$\mathtt{x}_n$`]` such that

$$(\exists\, i : 1 \leq i \leq n :: \mathtt{x}_i = \mathtt{True} \ \wedge\ (\forall\, j : 1 \leq j < i :: \mathtt{x}_j = \mathtt{False}))$$

Suppose $\mathtt{x}_i$ is the minimum i satisfying the above existential.

```
   foldr (||) False xs
⟹    { many steps }
   x1 || (x2 || ( ⋯ || (xi || ( ⋯ || (xn || False) ⋯ )
```

Because of the nonstrict definition of `||`, the above can stop after the $\mathtt{x}_i$ term is processed. None of the list to the right of $\mathtt{x}_i$ needs to be evaluated.

However, a version which uses `foldl` must process the entire list.

```
   foldl (||) False xs
⟹ { many steps }
   ( ⋯ ( False || x1) || x2) || ⋯) || xi) || ⋯) || xn
```

In this example, `foldr` is clearly more efficient than `foldl`.

14 DIVIDE AND CONQUER ALGORITHMS

Reference: This section is based on section 6.4 of the Bird and Wadler textbook [2] and section 4.2 of Kelly's book *Functional Programming for Loosely-coupled Multiprocessors* [19].

14.1 Overview

General strategy:

1. Decompose problem P into subproblems, each like P but with a smaller input argument.
2. Solve each subproblem, either directly or by recursively applying the strategy.
3. Assemble the solution to P by combining the solutions to its subproblems.

Advantages:

- Can lead to efficient solutions.
- Allows use of a "horizontal" parallelism. Similar problems can be solved simultaneously.

Section 6.4 of the Bird and Wadler textbook discusses several important divide and conquer algorithms: mergesort, quicksort, multiplication, and binary search. In these algorithms the divide and conquer technique leads to more efficient algorithms.

For example, a simple sequential search is $\mathcal{O}(n)$ (where n denotes the length of the input). Application of the divide and conquer strategy leads to the binary search which is a more efficient $\mathcal{O}(\log_2 n)$ search.

As a general pattern of computation, the divide and conquer strategy can be stated with the following higher order function.

```
divideAndConquer :: (a -> Bool)          -- trivial
                    -> (a -> b)          -- simplySolve
                    -> (a -> [a])        -- decompose
                    -> (a -> [b] -> b)   -- combineSolutions
                    -> a                 -- problem
                    -> b

divideAndConquer trivial simplySolve decompose
                 combineSolutions problem
     = solve problem
       where solve p | trivial p = simplySolve p
                     | otherwise = combineSolutions p
                                       (map solve (decompose p))
```

If the problem is trivially simple (i.e., `trivial p`), then it is solved directly using `simplySolve`.

If the problem is not trivially simple, then it is decomposed into a list of subproblems using `decompose` and each subproblem is solved separately using `map solve`. The list of solutions to the subproblems are then assembled into a solution for the problem using `combineSolutions`. (Sometimes `combineSolutions` may require the original problem description in order to put the solutions back together properly. Hence the parameter `p`.)

Note that the solution of each subproblem is completely independent from the solution of all the others. If all the subproblem solutions are needed by `combineSolutions`, then the language implementation could potentially solve the subproblems simultaneously. The implementation could take advantage of the availability of multiple processors and actually evaluate the expressions in parallel. This is "horizontal" parallelism as described above.

Note: If `combineSolutions` does not require all the subproblem solutions, then the subproblems cannot be safely solved in parallel. If they were, the result of `combineSolutions` might be nondeterministic, that is, the result could be dependent upon the relative order in which the subproblem results are completed.

14.2 Divide and Conquer Fibonacci

Now let's use the function `divideAndConquer` to define a few functions.

First, let's define a Fibonacci function. (This is adapted from the function defined on pages 77-8 of Kelly [19]. This function is inefficient. It is given here to illustrate the technique.)

```
fib :: Int -> Int
fib n = divideAndConquer trivial simplySolve decompose
                         combineSolutions problem
        where trivial 0                = True
              trivial 1                = True
              trivial (m+2)            = False
              simplySolve 0            = 0
              simplySolve 1            = 1
              decompose m              = [m-1,m-2]
              combineSolutions _ [x,y] = x + y
```

14.3 Divide and Conquer Folding

Next, let's consider a folding function (similar to `foldr` and `foldl`) that uses the function `divideAndConquer`. (This is adapted from the function defined on pages 79-80 of Kelly [19].)

```
fold :: (a -> a -> a) -> a -> [a] -> a
fold op i = divideAndConquer trivial simplySolve decompose
                             combineSolutions
            where trivial xs      = length xs <= 1
                  simplySolve []  = i
                  simplySolve [x] = x
                  decompose xs    = [take m xs, drop m xs]
                                    where m = length xs / 2
                  combineSolutions _ [x,y] = op x y
```

This function divides the input list into two almost equal parts, folds each part separately, and then applies the operation to the two partial results to get the combined result.

The `fold` function depends upon the operation `op` being *associative*. That is, the result must not be affected by the order in which the operation is applied to adjacent elements of the input list.

In `foldr` and `foldl`, the operations are not required to be associative. Thus the result might depend upon the right-to-left operation order in `foldr` or left-to-right order in `foldl`.

Function `fold` is thus a bit less general. However, since the operation is associative and `combineSolutions` is strict in all elements of its second argument, the operations on pairs of elements from the list can be safely done in parallel.

Another divide-and-conquer definition of a folding function follows. Function `fold'` is an optimized version of `fold`.

```
fold' :: (a -> a -> a) -> a -> [a] -> a
fold' op i xs = foldt (length xs) xs
                where foldt _ []  = i
                      foldt _ [x] = x
                      foldt n ys  = op (foldt m  (take m ys))
                                       (foldt m' (drop m ys))
                                  where m  = n / 2
                                        m' = n - m
```

14.4 Minimum and Maximum of a List

Consider the problem of finding both the minimum and the maximum values in a nonempty list and returning them as a pair.

First let's look at a definition that uses the left-folding operator.

```
sMinMax :: Ord a => [a] -> (a,a)
sMinMax (x:xs) = foldl' newmm (x,x) xs
                 where  newmm (y,z) u = (min y u, max z u)
```

Let's assume that the comparisons of the elements are expensive and base our time measure on the number of comparisons. Let n denote the length of the list argument.

A singleton list requires no comparisons. Each additional element adds two comparisons (one `min` and one `max`).

$$\begin{aligned} T(n) &= \begin{cases} 0, & \text{for } n = 1 \\ T(n-1) + 2, & \text{for } n \geq 2 \end{cases} \\ &= 2 * n - 2 \end{aligned}$$

Now let's look at a divide and conquer solution.

```
minMax :: Ord a => [a] -> (a,a)
minMax [x]   = (x,x)
minMax [x,y] = if x < y then (x,y) else (y,x)
minMax xs    = (min a c, max b d)
               where m     = length xs / 2
                     (a,b) = minMax (take m xs)
                     (c,d) = minMax (drop m xs)
```

Again let's count the number of comparisons for a list of length n.

$$T(n) = \begin{cases} 0, & \text{for } n = 1 \\ 1, & \text{for } n = 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, & \text{for } n > 2 \end{cases}$$

For convenience suppose $n = 2^k$ for some $k \geq 1$.

$$\begin{aligned}
T(n) &= 2 * T(n/2) + 2 \\
&= 2 * (2 * T(n/4) + 2) + 2 \\
&= 4 * T(n/4) + 4 + 2 \\
&= \cdots \\
&= 2^{k-1} * T(2) + (\Sigma\, i : 1 \leq i < k :: 2^i) \\
&= 2^{k-1} + 2 * (\Sigma\, i : 1 \leq i < k :: 2^i) - (\Sigma\, i : 1 \leq i < k :: 2^i) \\
&= 2^{k-1} + 2^k - 2 \\
&= 3 * 2^{k-1} - 2 \\
&= 3 * (n/2) - 2
\end{aligned}$$

Thus the divide and conquer version takes 25% fewer comparisons than the left-folding version.

So, if element comparisons are the expensive in relation to to the `take`, `drop`, and `length` list operations, then the divide and conquer version is better. However, if that is not the case, then the left-folding version is probably better.

Of course, we can also express `minMax` in terms of the function `divideAndConquer`.

```
minMax' :: Ord a => [a] -> (a,a)
minMax' = divideAndConquer trivial simplySolve decompose
                           combineSolutions
          where n                 = length xs
                m                 = n/2
                trivial xs        = n <= 2
                simplySolve [x]   = (x,x)
                simplySolve [x,y] = if x < y then (x,y) else (y,x)
                decompose xs      = [take m xs, drop m xs]
                combineSolutions _ [(a,b),(c,d)]
                                  = (min a c, max b d)
```

15 INFINITE DATA STRUCTURES

One particular benefit of lazy evaluation is that functions in Haskell can manipulate "infinite" data structures. Of course, a program cannot actually generate or store all of an infinite object, but lazy evaluation will allow the object to be built piece-by-piece as needed and the storage occupied by no-longer-needed pieces to be reclaimed.

15.1 Infinite Lists

Reference: This is based in part on section 7.1 of the Bird/Wadler textbook [2].

In Section 7 we looked at generators for infinite arithmetic sequences such as `[1..]` and `[1,3..]`. These infinite lists are encoded in the functions that generate the sequences. The sequences are only evaluated as far as needed, for example,

```
take 5 [1..]  ⟹ [1,2,3,4,5].
```

Haskell also allows infinite lists of infinite lists to be expressed as shown in the following example which generates a table of the multiples of the positive integers.

```
multiples :: [[Int]]
multiples = [ [ m*n | m<-[1..]]  | n <- [1..]  ]

multiples ⟹ [ [1, 2, 3, 4, 5,···],
               [2, 4, 6, 8,10,···],
               [3, 6, 9,12,14,···],
               [4, 8,12,16,20,···],
               ··· ]

take 4 (multiples !! 3) ⟹ [4,8,12,16]
```

Note: The operator `xs !! n` returns element `n` of the list `xs` (where the head is element `0`).

Haskell's infinite lists are not the same as infinite sets or sequences in mathematics. Infinite lists in Haskell correspond to infinite *computations* whereas infinite sets in mathematics are simply definitions.

In mathematics, set $\{x^2 | x \in \{1,2,3\} \wedge x^2 < 10\} = \{1,4,9\}$.

However, in Haskell, `show [ x * x | x <- [1..], x * x < 10 ]` $\Longrightarrow$ `[1,4,9` This is a computation that never returns a result. Often, we assign this computation the value `1:4:9:`$\perp$.

But `takeWhile (<10) [ x * x | x <- [1..]  ]` $\Longrightarrow$ `[1,4,9]`.

15.2 Iterate

Reference: This is based in part on section 7.2 of the Bird/Wadler textbook [2].

In mathematics, the notation f^n denotes the function f composed with itself n times. Thus, $f^0 = id$, $f^1 = f$, $f^2 = f.\ f$, $f^3 = f.\ f.\ f$, $\cdots$.

A useful function is the function `iterate` such that:

```
iterate f x = [x, f x, f^2 x, f^3 x, ··· x ]
```

The Haskell standard prelude defines `iterate` recursively as follows:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Example: Suppose we need the set of all powers of the integers, that is, a functions `powertables` such that:

```
powertables ⟹ [ [1, 2, 4, 8,···],
                 [1, 3, 9, 27,···],
                 [1, 4,16, 64,···],
                 [1, 5,25,125,···],
                 ··· ]
```

Using `iterate` we can define `powertables` compactly as follows:

```
powertables :: [[Int]]
powertables = [ iterate (*n) 1 | n <- [2..]]
```

Example: Suppose we want a function to extract the decimal digits of a positive integer.

```
digits :: Int -> [Int]
digits = reverse . map (‘mod‘ 10) . takeWhile (/= 0) . iterate (/10)
```

```
digits 178   (Not actual reduction steps)
          = reverse . map (‘mod‘ 10) . takeWhile (/= 0)
                                       [178,17,1,0,0,···]
          = reverse . map (‘mod‘ 10) [178,17,1]
          = reverse [8,7,1]
          = [1,7,8]
```

15.3 Prime Numbers: Sieve of Eratosthenes

Reference: This is based in part on section 7.3 of the Bird/Wadler textbook [2].

The Greek mathematician Eratosthenes described essentially the following procedure for generating the list of all prime numbers. This algorithm is called the Sieve of Eratosthenes.

1. Generate the list 2, 3, $4 \cdots$
2. Mark the first element p as prime.
3. Delete all multiples of p from the list.
4. Return to step 2.

Not only is the 2-3-4 loop infinite, but so are steps 1 and 3 themselves.

There is a straightforward translation of this algorithm to Haskell.

```
primes :: [Int]
primes = map head (iterate sieve [2..])

sieve (p:xs) = [x | x <- xs, x `mod` p /= 0 ]
```

Note: This uses an intermediate infinite list of infinite lists; even though it is evaluated lazily, it is still inefficient.

We can use function `primes` in various ways, e.g., to find the first `1000` primes or to find all the primes that are less than 10,000.

```
take 1000 primes
takeWhile (<10000) primes
```

Calculations such as these are not trivial if the computation is attempted using arrays in an "eager" language like Pascal—in particular it is difficult to know beforehand how large an array to declare for the lists.

However, by separating the concerns, that is, by keeping the computation of the primes separate from the application of the boundary conditions, the program becomes quite modular. The same basic computation can support different boundary conditions in different contexts.

Now let's transform the `primes` and `sieve` definitions to eliminate the infinite list of infinite lists. First, let's separate the generation of the infinite list of positive integers from the application of `sieve`.

```
primes        = rsieve [2..]
rsieve (p:ps) = map head (iterate sieve (p:ps))
```

Next, let's try to transform `rsieve` into a more efficient definition.

```
   rsieve (p:ps)
=     { rsieve }
   map head (iterate sieve (p:ps))
=     { iterate }
   map head ((p:ps) : (iterate sieve (sieve (p:ps)) ))
=     { map.2, head }
   p : map head (iterate sieve (sieve (p:ps)) )
=     { sieve }
   p : map head (iterate sieve [x | x <- ps, x `mod` p /= 0 ])
=     { rsieve }
   p : rsieve [x | x <- ps, x `mod` p /= 0 ]
```

This calculation gives us the new definition:

```
rsieve (p:ps) = p : rsieve [x | x <- ps, x `mod` p /= 0 ]
```

This new definition is, of course, equivalent to the original one, but it is slightly more efficient in that it does not use an infinite list of infinite lists.

.

References

[1] J. Backus. Can programming languages be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International, New York, 1988.

[3] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[4] E. Cohen. *Programming in the 1990's: An Introduction to the Calculation of Programs*. Springer-Verlag, New York, 1990.

[5] H. C. Cunningham. Notes on functional programming with Gofer. Technical Report UMCIS-1995-01, University of Mississippi, Department of Computer and Information Science, June 1995.

[6] A. Cuoco. *Investigations in Algebra*. MIT Press, 1990.

[7] A. J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.

[8] A. J. Field and P. G. Harrison. *Functional Programming*. Addison Wesley, Reading, Massachusetts, 1988.

[9] B. Hayes. On the ups and downs of hailstone numbers. *Scientific American*, 250(1):10–16, January 1984.

[10] R. Hoogerwoord. *The Design of Functional Programs: A Calculational Approach*. PhD thesis, Eindhoven Technical University, Eindhoven, The Netherlands, 1989.

[11] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[12] P. Hudak and J. H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN NOTICES*, 27(5), May 1992.

[13] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN NOTICES*, 27(5), May 1992.

[14] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[15] M. P. Jones. *An Introduction to Gofer*, 1991. Tutorial manual distributed as a part of the Gofer system beginning with Version 2.20.

[16] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, New York, 1987.

[17] S. L. Peyton Jones and D. Lester. *Implementing Functional Languages*. Prentice Hall International, New York, 1992.

[18] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall International, New York, 1990.

[19] P. Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. MIT Press, Cambridge, Massachusetts, 1989.

[20] G. Polya. *How to Solve It: A New Aspect of Mathematical Method*. Princeton Unversity Press, second edition, 1957.

[21] G. Polya. *Mathematical Discovery: On Understanding, Learning, and Teaching Problem Solving*. Wiley, combined edition, 1981.

[22] E. P. Wentworth. Introduction to functional programming using RUFL. Technical Report PPG 89/6, Rhodes University, Department of Computer Science, Grahamstown, South Africa, Revised August 1990.

Index