Chapter 10

Programming with Laziness

10.1 Haskell

The paradigmatic modern lazy programming language is called Haskell, in honor of Haskell Curry, a pioneering researcher who laid the foundation for a great deal of modern programming language theory. We will study the experience of programming in Haskell (using its own syntax) to get a feel for the benefits of laziness.

What follows is only a brief sampler of Haskell’s many wonders. In particular, it is colored by the fact that this text uses Haskell primarily to illustrate specific linguistic features, as opposed to providing a general introduction to Haskell. The Haskell language Web site\(^1\) has references to several texts and tutorials that describe the language in far greater detail and from several perspectives.

10.1.1 Expressions and Definitions

Like Scheme, simple Haskell programs do not need to be wreathed in scaffolding; and like most Scheme implementations, most Haskell implementations provide an interactive environment. These notes use one called Helium; others have a similar interface.

\begin{verbatim}
Prelude> 3
3

Prelude> True
True
\end{verbatim}

\begin{verbatim}
(Prelude> is a Haskell prompt whose significance will soon become clear.) Haskell employs a traditional algebraic syntax for operations (with the corresponding order of precedence), with parentheses representing only grouping:

Prelude> 2*3+5
11

Prelude> 2+3*5
\end{verbatim}

\(^1\)http://www.haskell.org/
As in most programming languages other than Scheme, some built-in operators are written in infix notation while most others, including user-defined ones, are written in prefix. A prefix operator can always be used in an infix position, however, using a special syntactic convention (note that these are *backquotes*):

```
Prelude> 7 `mod` 4
3
Prelude> (7 `mod` 4) `mod` 2
1
```

and infix operators can, similarly, be treated as a procedural value, even used in a prefix position:

```
Prelude> ((<) 4 ((+) 2 3))
True
```

The latter is syntactically unwieldy; why would one need it?²

We have seen integers (Int) and booleans (Bool). Haskell also has characters (of type Char) that are written inside single-quotes: ‘c’ (the character ‘c’), ‘3’ (the character ‘3’), ‘\n’ (the newline character), and so on.

Based on what we’ve seen so far, we can begin to write Haskell functions. This one proposes a grading scale for a course:

```haskell
scoreToLetter :: Int -> Char
scoreToLetter n
  | n > 90  = 'A'
  | n > 80  = 'B'
  | n > 70  = 'C'
  | otherwise = 'F'
```

The first line of this excerpt tells Haskell the type to expect for the corresponding definition (read :: as “has the type”). The rest of the excerpt defines the actual function using a series of rules, akin to a Scheme conditional. Loading this definition into the Haskell evaluator makes it available to execute³. To test the function, we use it in the evaluator:

---

²Answer: Because we may want to use a traditionally infix operator as an argument to another function. If Haskell lacked this notation, the use of the operator would lead to syntactic, type or semantic errors.

³In Helium, this definition must be saved in a file whose name begins with a capital letter. Helium’s file functions can be accessed from the menus or, as in most other Haskell implementations, from the Haskell command-line: :l followed by a filename loads the definitions in the named file, :r reloads the file loaded most recently, and so on. The implementation manual will describe other short-cuts.
10.1. HASKELL

CS173> scoreToLetter 83
'B'
CS173> scoreToLetter 51
'F'
CS173> scoreToLetter 99
'A'

Note that in typical Haskell implementations, upon loading a file, the prompt changes from Prelude to the name of the file, indicating in which context the expression will be evaluated. The Prelude is the set of definitions built into Haskell.

10.1.2 Lists

Haskell naturally has more sophisticated types as well. As in Scheme, lists are inductively (or recursively) defined data-structures; the empty list is written [] and non-empty list constructor is written :. Haskell also offers a convenient abbreviation for lists. Thus:

Prelude> []
[]
Prelude> 1:[]
[1]
Prelude> 1:2:
[1,2]
Prelude> 1:[2,3,2+2]
[1,2,3,4]

Note, however, that lists must be homogenous: that is, all values in a list must be of the same type.

Prelude> [1,'a']
Type error in element of list
expression : [1, 'a']
term : 'a'
type : Char
does not match : Int

(The exact syntax of the type error will depend on the specific Haskell implementation, but the gist should be the same. Here, Helium tells us that the second element of the list has type Char, whereas Helium was expecting a value of type Int based on the first list element.)

Haskell’s Prelude has many useful functions already built in, including standard list manipulatives:

CS173> filter odd [1, 2, 3, 4, 5]
[1,3,5]
CS173> sum [1, 2, 3, 4, 5]
15
CS173> product [1, 2, 3, 4, 5]
120
We can, of course, use these in the context of our new definition:

```
CS173> map scoreToLetter [83, 51, 99]
'BFA'
CS173> length (map scoreToLetter [83, 51, 99])
3
```

It takes a little practice to know when one can safely leave out the parentheses around an expression. Eliding them in the last interaction above leads to this error:

```
CS173> length map scoreToLetter [83, 51, 99]
Type error in application
  expression : length map scoreToLetter [83, 51, 99]
  term : length
  type : [a] -> Int
  does not match : ((b -> c) -> [b] -> [c]) -> (Int -> Char) -> [Int] -> d
probable fix : remove first and second argument
```

What?!! With practice (and patience), we realize that Haskell is effectively saying that `length` takes only one argument, while the use has three: `length`, `scoreToLetter` and `[83, 51, 99]`. In this case, Helium’s suggestion is misleading: the fix is not to remove the arguments but rather to inform Haskell of our intent (first map the function, the determine the length of the result) with parentheses.

Suppose Haskell didn’t have `length` built in. We could build it easily, using Haskell’s pattern-matching notation:

```
len [] = 0
len (x:s) = 1 + len s
```

The use of pattern-matching (here, the term `(x:s)`) automatically deconstructs the list, though Haskell also provides the operators `head` and `tail` for explicit manipulation). Notice, however, that we haven’t written a type declaration for `length`. This brings us to two interesting aspects of Haskell.

### 10.1.3 Polymorphic Type Inference

We can ask Haskell for the type of any expression using the `:type` or `:t` directive. For instance:

```
CS173> :t 3
3 :: Int
CS173> :t True
True :: Bool
CS173> :t 3 + 4
3 + 4 :: Int
```

What should we expect when we ask Haskell for the type of `len`? Haskell responds with

```
CS173> :t len
len :: [a] -> Int
```
10.1. HASKELL

What does this type mean? It says that len consumes a list and returns a Int, but it says a little more. Specifically, it says that the list consumed by len must have elements (recall that lists are homogenous) of type...a. But a is not a concrete type like Int or Bool; rather, it is a type variable. Mathematically, we would write this as

$$\forall \alpha . \text{len} : [\alpha] \rightarrow \text{Int}$$

That is, $\alpha$ is bound to a concrete type, and remains bound to that type for a particular use of len; but different uses of len can bind $\alpha$ to different concrete types. We call such types polymorphic, and will study them in great detail in Section 26.

We can see the type parameter at work more clearly using the following (trivial) function:

```haskell
listCopy [] = []
listCopy (x:s) = x : listCopy s
```

Haskell reports this type as

```haskell
CS173> :t listCopy
listCopy :: [a] -> [a]
```

which is Haskell’s notation for

$$\forall \alpha . \text{listCopy} : [\alpha] \rightarrow [\alpha]$$

When we apply listCopy to different argument list types, we see that it produces lists of the same type as the input each time:

```haskell
CS173> :t listCopy [1,2,3]
listCopy [1,2,3] :: [Int]
CS173> :t listCopy ['a','b','c']
listCopy ['a','b','c'] :: [Char]
CS173> :t listCopy [[1], [1, 2], []]
listCopy [[1], [1, 2], []] :: [[Int]]
```

In the last instance, notice that we are applying listCopy to—and obtaining as a result—a list of type list of Int (i.e., a nested list of integers).

Why does Haskell assign the type parameter a name (a)? When there is only one parameter the name isn’t necessary, but some functions are parameterized over multiple types. For instance, map is of this form:

```haskell
CS173> :t map
map :: (a -> b) -> [a] -> [b]
```

which we might write with explicit quantifiers as

$$\forall \alpha, \beta . \text{map} : (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$

Just from reading the type we can guess at map’s behavior: it consumes a function that transforms each $\alpha$ into a corresponding $\beta$, so given a list of $\alpha$’s it generates the corresponding list of $\beta$’s.

In the process of studying polymorphism, we may have overlooked something quite remarkable: that Haskell was able to generate types without our ever specifying them! This process is known as type inference. Indeed, not only is Haskell able to infer a type, it infers the most general type it can for each expression. We will study the machinery behind this remarkable power in Section 26.
Exercise 10.1.1 What would you expect is the type of the empty list? Check your guess using a Haskell implementation.

Exercise 10.1.2 Why does Haskell print the type of multi-argument functions with arrows between each adjacent pair of arguments? Experiment with Haskell by providing what appears to be a two-argument function (such as \texttt{map}) with only one argument.

10.1.4 Laziness

Is Haskell eager or lazy? We can test this using a simple interaction:

\begin{verbatim}
CS173> head []
exception: Prelude.head: empty list.
\end{verbatim}

This tells us that attempting to ask for the first element of the empty list will result in a run-time exception. Therefore, if Haskell used eager evaluation, the following expression should also result in an error:

\begin{verbatim}
CS173> (\ x -> 3) (head [])
3
\end{verbatim}

The expression (\ x -> 3) uses Haskell’s notation for defining an anonymous procedure: it is the syntactic analog of Scheme’s (lambda (x) 3). Thus, the whole expression is equivalent to writing

\begin{verbatim}
((lambda (x) 3) (first empty))
\end{verbatim}

which in Scheme would indeed result in an error. Instead, Haskell evaluates it to 3. From this, we can posit that Haskell does not evaluate the argument until it is used, and therefore follows a lazy evaluation regime.

Why is laziness useful? Clearly, we rarely write a function that entirely ignores its argument. On the other hand, functions do frequently use different subsets of their arguments in different circumstances, based on some dynamic condition. Most programming languages offer a form of short-circuited evaluation for the branches of conditional (based on the value of the test expression, only one or the other branch evaluates) and for Boolean connectives (if the first branch of a disjunction yields true the second branch need not evaluate, and dually for conjunction). Haskell simply asks why this capability should not be lifted to function arguments also.

In particular, since Haskell treats all function applications lazily, this also encompasses the use of most built-in constructors, such as the list constructor. As a result, when confronted with a definition such as

\begin{verbatim}
ones = 1 : ones
\end{verbatim}

Haskell does not evaluate the second argument to : until necessary. When it does evaluate it, there is a definition available for ones: namely, a 1 followed by \ldots. The result is therefore an infinite list, but only the act of examining the list actually constructs any prefix of it.

How do we examine an infinite list? Consider a function such as this:

\begin{verbatim}
front :: Int -> [a] -> [a]
front _ [] = []
front 0 (x:s) = []
front n (x:s) = x : front (n-1) s
\end{verbatim}
When used, `front` causes as many list constructions of `ones` as necessary until the recursion terminates—

```
CS173> front 5 ones
[1,1,1,1,1]
CS173> front 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

—but no more. Because the language does not force `front` to evaluate its arguments until necessary, Haskell does not construct any more of `ones` than is needed for `front` to terminate. That is, it is the act of pattern-matching that forces `ones` to grow, since the patternmatcher must determine the form of the list to determine which branch of the function to evaluate.

Obtaining the prefix of a list of ones may not seem especially impressive, but there are many good uses for `front`. Suppose, for instance, we have a function that generates the eigenvalues of a matrix. Natural algorithms for this problem generate the values in decreasing order of magnitude, and in most applications, only the first few are meaningful. In a lazy language, we can pretend we have the entire sequence of eigenvalues, and use `front` to obtain just as many as the actual application needs; this in turn causes only that many to be computed. Indeed, any application can freely generate an infinite list of values, safe in the knowledge that a consumer can use operators such as `front` to inspect the prefix it cares about.

The function `front` is so useful when programming in Haskell that it is actually built into the Prelude, under the name `take`. Performing the same computation in an eager language is considerably more complex, because the computation that generates values and the one that consumes them must explicitly coordinate with each other: in particular, the generator must be programmed to explicitly expect requests from the consumer. This complicates the construction of the generator, which may already have complex domain-specific code; worse, if the generator was not written with such a use in mind, it is not easy to adapt it to behave accordingly.

Where else are infinite lists useful? Consider the process of generating a table of data whose rows cycle between a fixed set of colors. Haskell provides a function `cycle` that consumes a list and generates the corresponding cyclic list:

```
CS173> take 5 (cycle ["blue", "rondo"])
["blue","rondo","blue","rondo","blue"]
```

The procedure for displaying the data can consume the cyclic list and simply extract as many elements from it as necessary. The generator of the cyclic list doesn’t need to know how many rows there will be in the table; laziness ensures that the entire infinite list does not get generated unless necessary. In other words, programmers often find it convenient to create cyclic data structure not so much to build a truly infinite data structure, but rather to produce one that is large enough for all possible consumers (none of whom will ever examine more than a finite prefix, but each of whom may want a different number of prefix elements).

Consider one more example. At the end of some stages of the Tour de France, the top finishers receive a “time bonus”, which we can think of as a certain number of bonus points. Let us suppose that the top three finishers receive 20-, 12- and 8-second bonuses, respectively, while the others receive none. Given a list reflecting the order in which contestants completed a stage, we would like a list that pairs each name with the number of points that person received. That is, we would like a function `timeBonuses` such that
CHAPTER 10. PROGRAMMING WITH LAZINESS

where the notation \((...,\ldots)\) indicates an anonymous tuple of zero or more elements. Note that the result is therefore a list of two-tuples (or pairs), where the heterogeneity of lists forces each tuple to be of the same type (a string in the first projection and a number in the second).

We can write `timeBonuses` by employing the following strategy. Observe that every position gets a fixed bonus (20, 12, and 8, followed by zero for everyone else), but we don’t know how many finishers there will be. In fact, it isn’t even clear there will be three finishers if the organizers run a particularly brutal stage! First let’s create a list of all the bonuses:

\[
[20, 12, 8] ++ \text{cycle \[0\]}
\]

where ++ appends lists. We can check that this list’s content matches our intuition:

\[
\text{Prelude}\text{> take 10 ([20, 12, 8] ++ \text{cycle \[0\]})}
\]

\[
[20,12,8,0,0,0,0,0,0,0]
\]

Now we need a helper function that will match up the list of finishers with the list of scores. Let’s define this function in parts:

\[
tB :: [\text{String}] \rightarrow [\text{Int}] \rightarrow [(\text{String},\text{Int})]
\]

Clearly, if there are no more finishers, the result must also be the empty list; we can ignore the second argument. In contrast, if there is a finisher, we want to assign him the next available time bonus:

\[
tB (f:fs) (b:bs) = (f,b) : tB fs bs
\]

The right-hand side of this definition says that we create an anonymous pair out of the first elements of each list, and construct a list out of this pair and the natural recursion.

At this point our helper function definition is complete. A Haskell implementation ought to complain that we haven’t specified what should happen if the second argument is empty but the first is not:

\[(26,1): \text{Warning: Missing pattern in function bindings:}
\]
\[
tBb (_ : _) [] = ...
\]

This message says that the case where the first list is not empty (indicated by \((_ : _)\)) and the second one is (\([]\)) hasn’t been covered. Since we know the second list is infinitely long, we can ignore this warning.

Given this definition of `tB`, it is now straightforward to define `timeBonuses`:

\[
\text{timeBonuses finishers =}
\]
\[
tB \text{finishers ([20, 12, 8] ++ \text{cycle \[0\]})}
\]

This definition matches the test case above. We should also be sure to test it with fewer than three finishers:

\[
\text{CS173}\text{> timeBonuses ["Lance", "Jan"]}
\]
\[
["Lance",20],("Jan",12)]
\]
Indeed, the helper function \( \text{tB} \) is so helpful, it too (in a slightly different form) is built into the Haskell Prelude. This more general function, which terminates the recursion when the second list is empty, too, is called \( \text{zip} \):

\[
\text{zip} \ [\ ] \ _ = [\ ] \\
\text{zip} \ _ \ [\ ] = [\ ] \\
\text{zip} \ (a:as) \ (b:bs) = (a,b) : \text{zip} \ as \ bs
\]

Notice that the type of \( \text{zip} \) is entirely polymorphic:

\[
\text{CS173} > :\text{type} \ \text{zip} \\
\text{zip} :: [a] \to [b] \to [(a, b)]
\]

Its name is suggestive of its behavior: think of the two lists as the two rows of teeth, and the function as the zipper that pairs them.

Haskell can equally comfortably accommodate non-cyclic infinite lists. To demonstrate this, let’s first define the function \( \text{zipOp} \). It generalizes \( \text{zip} \) by consuming an operator to apply to the pair of first elements:

\[
\text{zipOp} :: (a -> b -> c) -> [a] -> [b] -> [c] \\
\text{zipOp} \ f \ [\ ] \ _ = [\ ] \\
\text{zipOp} \ f \ _ \ [\ ] = [\ ] \\
\text{zipOp} \ f \ (a:as) \ (b:bs) = (f \ a \ b) : \text{zipOp} \ f \ as \ bs
\]

We can recover the \( \text{zip} \) operation from \( \text{zipOp} \) easily:

\[
\text{myZip} = \text{zipOp} \ (\ \lambda \ a \ b \to (a,b))
\]

But we can also pass \( \text{zipOp} \) other operators, such as \((+)\):

\[
\text{CS173} > \text{zipOp} \ (+) \ [1, \ 1, \ 2, \ 3, \ 5] \ [1, \ 2, \ 3, \ 5, \ 8] \\
\ [2, \ 3, \ 5, \ 8, \ 13]
\]

In fact, \( \text{zipOp} \) is also built into the Haskell Prelude, under the name \( \text{zipWith} \).

In the sample interaction above, we are clearly beginning to build up the sequence of Fibonacci numbers. But there is an infinite number of these and, indeed, there is no reason the arguments to \( \text{zipOp} \) must be finite lists. Let us therefore generate the entire sequence. The code above is suggestive: clearly the first and second argument are the same list (the list of all Fibonacci numbers), but the second is the first list “shifted” by one, i.e., the tail of that list. We might therefore try to seed the process with the initial values, then use that seed to construct the remainder of the list:

\[
\text{seed} = [1, \ 1] \\
\text{output} = \text{zipOp} \ (+) \ \text{seed} \ (\text{tail} \ \text{seed})
\]

\[ ^4 \text{Recall that } (\ \lambda \cdots) \text{ is Haskell’s equivalent of } (\text{lambda} \cdots). \]

\[ ^5 \text{We have to enclose + to avoid parsing errors, since + is an infix operator. Without the parentheses, Haskell would try to add the value of zipOp to the list passed as the first argument.} \]
But this produces only one more Fibonacci number before running out of input values, i.e., output is bound to \([2]\). So we have made progress, but need to find a way to keep seed from exhausting itself. Indeed, it would seem that we want a way to make seed and output be the same, so that each new value computed triggers one more computation! Indeed,

\[
fibs = 1 : 1 : \text{zipOp (+)} \text{ fibs (tail fibs)}
\]

We can test this in Haskell:

\[
\text{CS173> take 12 fibs}
\[1,1,2,3,5,8,13,21,34,55,89,144]\]

and indeed fibs represents the entire infinite list of Fibonacci numbers, ready for further use.

**Exercise 10.1.3** Earlier, we saw the following interaction:

\[
\text{Prelude> take 10 ([20, 12, 8] ++ cycle [0])}
\[20,12,8,0,0,0,0,0,0,0]\]

What happens if you instead write take 10 [20, 12, 8] ++ cycle [0]? Does it result in a type error? If not, do you get the expected answer? If so, is it for the right reasons? Try this by hand before entering it into Haskell.

**Exercise 10.1.4** The definition of the Fibonacci sequence raises the question of which “algorithm” Haskell is employing. Is it computing the \(n^{th}\) Fibonacci number in time linear in \(n\) (assuming constant-time arithmetic) or exponential in \(n\)?

1. First, try to determine this experimentally by asking for the \(n^{th}\) term for large values of \(n\) (though you may have trouble with arithmetic overflow).

2. Of course, even if you observe linear behavior, this is not proof; it may simply be that you didn’t use a large enough value of \(n\) to observe the exponential. Therefore, try to reason about this deductively. What about Haskell will determine the computation time of the \(n^{th}\) Fibonacci?

10.1.5 An Interpreter

Finally, we demonstrate an interpreter for WAE written in Haskell. First we define some type aliases,

\[
\text{type Identifier = String}
\text{type Value = Int}
\]

followed by the two important type definitions:

\[
\text{type SubCache = [(Identifier, Value)]}
\text{data WAE = Num Int}
| Add WAE WAE
| Id Identifier
| With Identifier WAE WAE
The core interpreter is defined by cases:

```
interp :: WAE -> SubCache -> Value
interp (Num n) sc = n
interp (Add lhs rhs) sc = interp lhs sc + interp rhs sc
interp (Id i) sc = lookup i sc
interp (With bound_id named_expr bound_body) sc =
    interp bound_body
    (extend sc bound_id (interp named_expr sc))
```

The helper functions are equally straightforward:

```
lookup :: Identifier -> SubCache -> Value
lookup var ((i,v):r)
  | (eqString var i) = v
  | otherwise = lookup var r

extend :: SubCache -> Identifier -> Value -> SubCache
extend sc i v = (i,v):sc
```

This definition of `lookup` uses Haskell’s pattern-matching notation as an alternative to writing an explicit conditional. Finally, testing these yields the expected results:

```
CS173> interp (Add (Num 3) (Num 5)) []
8
CS173> interp (With "x" (Add (Num 3) (Num 5)) (Add (Id "x") (Id "x"))) []
16
```

If we comment out the type declaration for `interp` (a line beginning with two dashes (--) is treated as a comment), Haskell infers this type for it:

```
interp :: WAE -> SubCache -> Int
```

This is the same type as the one we ascribed, differing only in the use of the type alias in one case but not the other.

**Exercise 10.1.5** Extend the Haskell interpreter to implement functions using Haskell functions to represent functions in the interpreted language. Ensure that the interpreted language evaluates under an eager, not lazy, regime.

**Exercise 10.1.6** It is instructive to extend the Haskell interpreter to implement recursion. Use the data structure representation of the environment. In Section 9 this required mutation. Haskell does not support mutation. Do you need it?
10.2 Shell Scripting

While most programmers have never programmed in Haskell before, many have programmed in a lazy language: the language of most Unix shells. In this text we’ll use the language of bash (the Bourne Again Shell), though most of these programs work identically or have very close counterparts in other popular shell languages.

The classical shell model assumes that all programs can potentially generate an infinite stream of output. The simplest such example is the program yes, which generates an infinite stream of y’s:

```bash
> yes
y
y
y
```
and so on, forever. (Don’t try this at home without your fingers poised over the interrupt key!) To make it easier to browse the output of (potentially infinite) stream generators, Unix provides helpful applications such as more to page through the stream. In Haskell, function composition makes the output of one function (the equivalent of a stream-generating application) the input to another. In a shell, the | operator does the same. That is,

```bash
> yes | more
```
generates the same stream, but lets us view finite prefixes of its content in segments followed by a prompt. Quitting from more terminates yes.

What good is yes? Suppose you run the following command:

```bash
> rm -r Programs/Sources/Java
```
Say some of these files are write-protected. For each such file, the shell will generate the query

```
rm: remove write-protected file ‘Programs/Sources/Java/frob.java’?
```
If you know for sure you want to delete all the files in the directory, you probably don’t want to manually type y in response to each such question. How many can there be? Unfortunately, it’s impossible to predict how many write-protected files will be in the directory. This is exactly where yes comes in:

```bash
> yes | rm -r Programs/Sources/Java
```
generates as many y inputs as necessary, satisfying all the queries, thereby deleting all the files.

We’ve seen that more is a useful way of examining part of a stream. But more is not directly analogous to Haskell’s take. In fact, there is a Unix application that is: it’s called head. head prints the first n entries in a stream, where n is given as an argument (defaulting to 10):

```bash
> yes | head -5
y
y
y
y
y
```
These examples already demonstrate the value of thinking of Unix programs as generators and consumers of potentially infinite streams, composed with \(|\). Here are some more examples. The application \texttt{wc} counts the number of characters (\texttt{-c}), words (\texttt{-w}) and lines (\texttt{-l}) in its input stream. Thus, for instance,

\begin{verbatim}
> yes | head -5 | wc -l
  5
\end{verbatim}

(not surprisingly). We can similarly count the number of files with suffix \texttt{.scm} in a directory:

\begin{verbatim}
> ls * .scm | wc -l
  2
\end{verbatim}

We can compose these into longer chains. Say we have a file containing a list of grades, one on each line; say the grades (in any order in the file) are two 10s, one 15, one 17, one 21, three 5s, one 2, and ten 3s. Suppose we want to determine which grades occur most frequently (and how often), in descending order.

The first thing we might do is sort the grades, using \texttt{sort}. This arranges all the grades in order. While sorting is not strictly necessary to solve this problem, it does enable us to use a very useful Unix application called \texttt{uniq}. This application eliminates adjacent lines that are identical. Furthermore, if supplied the \texttt{-c} (“count”) flag, it prepends each line in the output with a count of how many adjacent lines there were. Thus,

\begin{verbatim}
> sort grades | uniq -c
  2 10
  1 15
  1 17
  1 2
  1 21
 10  3
  3  5
\end{verbatim}

This almost accomplishes the task, except we don’t get the frequencies in order. We need to sort one more time. Simply sorting doesn’t do the right thing in two ways:

\begin{verbatim}
> sort grades | uniq -c | sort
  1 15
  1 17
  1  2
  1 21
  2 10
  3  5
 10  3
\end{verbatim}

We want sorting to be numeric, not textual, and we want the sorting done in reverse (decreasing) order. Therefore:
> sort grades | uniq -c | sort -nr
  10 3
  3 5
  2 10
  1 21
  1 2
  1 17
  1 15

There is something fundamentally beautiful—and very powerful!—about the structure of the Unix shell. Virtually all Unix commands respect the stream convention, and so do even some programming languages built atop it: for instance, by default, Awk processes its input one-line-at-a-time, so the Awk program
\{print $1\}
prints the first field of each line, continuing until the input runs out of lines (if ever), at which point the output stream terminates. This great uniformity makes composing programs easy, thereby encouraging programmers to do it.

Alan Perlis recognized the wisdom of such a design in this epigram: “It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures” (the data structure here being the stream). The greatest shortcoming of the Unix shell is that is is so lacking in data-sub-structure, relying purely on strings, that every program has to repeatedly parse, often doing so incorrectly. For example, if a directory holds a filename containing a newline, that newline will appear in the output of \texttt{ls}; a program like \texttt{wc} will then count the two lines as two different files. Unix shell scripts are notoriously fragile in these regards. Perlis recognized this too: “The string is a stark data structure and everywhere it is passed there is much duplication of process.”

The heart of the problem is that the output of Unix shell commands have to do double duty: they must be readable by humans but also ready for processing by other programs. By choosing human readability as the default, the output is sub-optimal, even dangerous, for processing by programs: it’s as if the addition procedure in a normal programming language always returned strings because you might eventually want to print an answer, instead of returning numbers (which are necessary to perform further arithmetic) and leaving conversion of numbers to strings to the appropriate input/output routine.

In short, Unix shell languages are both a zenith and a nadir of programming language design. Please study their design very carefully, but also be sure to learn the right lessons from them!

---

6We might fantasize the following way of making shell scripts more robust: all Unix utilities are forced to support a \texttt{-xmlout} flag that forces them to generate output in a standard XML language that did no more than wrap tags around each record (usually, but not always, line) and each field, and a \texttt{-xmlin} flag that informs them to expect data in the same format. This would eliminate the ambiguity inherent in parsing text.