COM2001: Advanced Programming Topics
A Practical Guide using Haskell and Java
(5th Edition)

Mike Stannett
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello, Sheffield S1 4DP, United Kingdom
m.stannett@dcs.shef.ac.uk

February 2013

Contents

1 Basic concepts **1**

1.1 Introduction 1

1.2 Syntax 1

1.3 Types 2

1.4 Semantics 3

1.5 Completeness 5

1.6 Summary 6

1.7 How complete is our stack ADT? 6

1.8 More Examples 6

1.8.1 Booleans 7

1.8.2 Natural Numbers 8

2 ADTs and HASKELL **9**

2.1 Introduction 9

2.1.1 An aside: Haskell's **module** system 10

2.2 Either a b 10

2.3 Implementation using **Either** 11

2.3.1 Types 11

2.3.2 Constructors 11

2.3.3 Non-constructors 12

2.3.4 Problems with this approach 12

2.4 Using an invariant assertion 13

2.4.1 Changing **pop**'s argument type 13

2.5 Removing the need for **Msg** 15

2.6 Exploiting side-effects using **error** 17

2.7 Using Haskell's **class** system 18

2.8 Summary 19

3 Program proof **21**

3.1 Introduction 21

3.2 How can we establish program correctness? 21

3.2.1 Correctness via testing 21

3.2.2 Correctness by construction 22

3.2.3 Correctness by program proof 22

3.3 Floyd-Hoare Logic 22

3.3.1 Transformation rules 23

3.3.2 Using the annotations 24

3.3.3 Example: Euclid's Algorithm 27

3.4 Proof by induction 31

3.4.1 Putting it all together 33

4 ADTs and their applications **35**

4.1 Introduction 35

4.2 Equational Reasoning 35

4.3 Two Induction Principles for Natural Numbers 37

4.3.1 The relationship between induction and structure 38

4.4 Structural Induction 39

4.4.1 Example: `Nat1` 40

4.4.2 Example: `Nat2` 40

4.5 Structural Induction and Lists 40

4.5.1 Example: `length` and `reverse` 41

4.5.2 Why do values have to be finite and defined? 41

4.6 Structural induction on trees 42

4.7 Summary 43

5 Parsing **45**

5.1 Introduction 45

5.2 The growth of functions 48

5.2.1 Example: Growth of polynomials 48

5.2.2 Big-O notation 49

5.2.3 Comparing functions 50

5.3 Counting the number of steps executed 50

5.3.1 The step-counting function T_f and the cost-function T 51

5.4 Two Examples 52

5.4.1 The function `sum` 52

5.4.2 The concatenation function . . . 52
5.4.3 The reverse function . . . 53
5.4.4 Complexity measures for these three examples . . . 53
5.5 Recurrence Rules . . . 54
5.5.1 Closed form complexity function for `sum` . . . 54
5.5.2 Closed form complexity function for `(++)` . . . 55
5.5.3 Closed form complexity function for `rev` . . . 55
5.6 Summary . . . 55

6 Structural Induction 57
6.1 Introduction . . . 57
6.2 Algebraic Data Types . . . 57
6.3 Field Labels . . . 58
6.4 Haskell's class system . . . 59
6.5 Example of classes in action . . . 61
6.6 Constraint propagation and typing . . . 63
6.6.1 Rule 1. Function Application . . . 64
6.6.2 Rule 2. Type Instantiation . . . 64
6.6.3 Rule 3. Abstraction . . . 64
6.7 Summary . . . 65

7 Complexity analysis 67
7.1 Introduction . . . 67
7.1.1 Abstract Data Types . . . 67
7.1.2 ADTS and Haskell . . . 67
7.1.3 Program Proof . . . 67
7.1.4 Haskell's class and type systems . . . 68
7.2 From Specification to Representation . . . 68
7.2.1 What features need to be represented? . . . 69
7.2.2 Choosing a container . . . 70
7.3 From representation to implementation . . . 70
7.3.1 Implementation using arrays . . . 71
7.3.2 Implementation using lists . . . 73
7.4 Summary . . . 75

8 Advanced HASKELL: the class and type systems 77
8.1 Introduction . . . 77

8.1.1 Queues 77
8.1.2 Other ordered structures 77
8.1.3 Unordered structures 78
8.1.4 Trees 79
8.1.5 Unexpected uses! 79
8.2 Binary Trees and Tree-sort 79
8.2.1 Tree syntax 80
8.2.2 Tree semantics 81
8.2.3 Representation of *Tree* 82
8.2.4 Tree implementations in Haskell 82
8.2.5 Tree-sort *vs* insertion-sort 84
8.3 Heaps, heap-sort, and priority queues 84
8.3.1 Heap syntax and semantics 85
8.3.2 Implementation of a priority queue 86
8.3.3 A scheduler 87
8.4 Summary 87

9 Specification → representation → implementation 89

9.1 Introduction 89
9.2 Basic Parsing 89
9.2.1 Basic parsing components 90
9.2.2 Match this or match that 91
9.2.3 Chaining parsers together 92
9.2.4 Simplifying the results of parsing 93
9.2.5 Matching lists 94
9.2.6 Matching a non-empty list 95
9.2.7 Matching a given string 95
9.2.8 Some specific tokens of interest 95
9.3 Parsing Types 97
9.3.1 The datatype `Type` 97
9.3.2 Parsing types 98
9.4 Parsing expressions 100
9.4.1 Parsing expressions 101
9.5 Declaring types of user-specified functions 103
9.6 Type evaluation 104
9.7 Simplifying the testing process 104

9.8 Summary 105

A Sample Problem Sheets **107**

B Past Assignments **121**

C Haskell Typing Rules **131**

D Basic Definitions **133**

E Solutions to Problem Sheets **143**

Chapter 1

Basic concepts

1.1 Introduction

One of the most common structures you'll find in software engineering is the *stack*. But what exactly *is* a stack? Would you recognise one if you saw one? Here's a fairly standard attempt at a definition.

> A stack is a last-in first-out (LIFO) data structure, typically used for storing entries of some kind, equipped with the operations
>
> - **push** - pushes an entry onto the stack
> - **pop** - pops an entry off of the stack

This is the sort of answer you might see given by a student in an exam script. But how good an answer is it? Would it enable a programmer to program a stack? Would it enable a compiler writer to represent the stack's behaviour correctly? Quite obviously, the answer to both questions is *No*, because all we've done is *name* two operations; we haven't said enough about what those operations are supposed to do. We need to explain their *behaviours*. For example, we need to point out that **pop**ping and **push**ing happen at the same 'end' of the stack (*i.e.*, what does LIFO mean?), and that they're 'sort-of inverses'. We do this by defining their *syntax* and *semantics*.

1.2 Syntax

Syntax tells us whether or not an expression is ***well-formed***. In this case, the syntax will tell us the types of object each operator takes for its argument(s), and what type of result it returns. Let's write **Stack a** for the datatype 'stacks whose entries are of type **a**'. I've used this notation because of its familiarity from Haskell, but if you prefer you could write **Entry** instead of **a** and **Stack of Entry** instead of **Stack a**. The notation is up to you - what matters is that you *explain* it. Given my notation, the operations could perhaps be defined to have the following types:

- **push**: **a** $\rightarrow$ **Stack a** $\rightarrow$ **Stack a**
- **pop** : **Stack a** $\rightarrow$ **Stack a**

According to this syntax,

- **push** takes an entry of type **a** and a stack of type **Stack a**, and combines them to yield a stack of type **Stack a**.
- **pop** takes a stack of type **Stack a**, and returns a stack of type **Stack a**.

Notice that this syntax involves the making of choices. I could just as well have defined **pop** so that it returned *two* values (the entry removed, together with the remaining stack). Traditionally, however, we introduce another operation called **top**, whose job it is to tell us what entry is currently at the top of the stack. This is one of the reasons it's so important to write down your syntax - otherwise, you could find two people using the same words but meaning different things. It's only when you write down the precise syntax that other readers can be certain that you're using operations in essentially the same way they are.

It's easy to see that our list of operations is incomplete, because they don't even tell us how to *create* a stack in the first place (all we have are operations for manipulating stacks that already exist). This is a general feature of the systems we'll be looking at - you *always* need to include at least one operation for *creating* an instance of the type, and (for reasons that will become obvious) for *deciding if an instance is empty.*

That is, we need to extend our syntax with another two operations. What's more, we need to think more carefully about the *types* involved in these operations. For example, what happens if we try to find the top element of an empty stack? The simplest answer is to set aside a special value that's returned whenever something unusual happens; in other words, we can respond with a message of some kind. Consequently, I'll write **Msg** to denote some set of messages, including, in particular, the message

- the stack is empty

Given these considerations, we can re-draft the complete syntax of **Stack a**, as shown in Table 1.1.

1.3 Types

If we look at the operations defined above, it's immediately obvious that they refer to a number of auxiliary data types which are currently undefined. These include

- **a**
  The type of data that will be stored in the stack. This can be sorted out later, when we know more about the needs of the programmer. For example, if the programmer eventually decides that a stack of integers is needed, they might take **a** to be **Integer**.
- **Bool**
  A data type representing the Boolean values **True** and **False**. You may be surprised to see me saying that this type is 'undefined', when it's so familiar from JAVA, Haskell and other languages. But in fact there are many well-established programming languages that don't provide Booleans, including, for example, C. These use integers to represent truth values instead, with all non-zero values being treated as **True** and zero as **False**.
- **Msg**
  A data type representing the various messages that might be generated by the program in response to operators being used in situations where their behaviour is undefined. Notice that these aren't necessarily *error* messages. It's not really an error for the programmer to try popping an empty stack; what makes the outcome undefined is *our decision* to leave things this way. We could just as easily have defined the result of popping the empty stack

Operation	*Intent*
createStack: $\rightarrow$ **Stack a**	Takes no parameters, and creates a new stack of type **Stack a** containing no entries.
emptyStack: **Stack a** $\rightarrow$ **Bool**	Takes a stack of type **Stack a** and returns a **Bool**ean value indicating whether the stack is empty.
push: **a** $\rightarrow$ **Stack a** $\rightarrow$ **Stack a**	Takes an entry of type **a** and a stack of type **Stack a**, and returns the new stack of type **Stack a** obtained by pushing the entry onto the old stack
pop: **Stack a** $\rightarrow$ (**Stack a** $\cup$ **Msg**)	Takes a stack of type **Stack a**, and returns the new stack of type **Stack a** that is obtained by removing the top element of the old stack, if it has one. If the original stack is empty, the operator returns a message instead.
top: **Stack a** $\rightarrow$ (**a** $\cup$ **Msg**)	Takes a stack of type **Stack a** and returns the entry of type **a** currently on top of the stack, if there is one. If the stack is empty, the operator returns a message instead.

Table 1.1: A possible syntax for **Stack a**. Notice that these operations are completely general; we haven't assumed anything about the language the stack will eventually be programmed in. The goal is to work out what operations are involved – we can worry about implementing them later.

to be the empty stack itself, in which case the outcome would have been well defined, and no message would have been needed.

- (**a** $\cup$ **Msg**) and (**Stack a** $\cup$ **Msg**)
  We need to know how to represent types of the form 'either **a** or **b**'. Some languages offer standard techniques for generating such types; for example, *C* has a built-in **union** type. In Haskell we could use the type construct **Either a b**. We'll look at this in more detail later.

Before a programmer can make use of our **Stack a** data type, they *must* first define what they mean by each of these auxiliary types.

1.4 Semantics

Now we've explained what parameters each operation expects to be given, and what sort of result it produces, we can start to define their semantics. The goal here is to give as complete a description as possible, without bringing in any irrelevant implementation details. After all, whatever a *stack* actually is, the one thing we can be fairly certain about is that it can be programmed in many different languages. We don't want our description to rely on the features of any one of those languages, as this will cause problems if we decide to use a different language later.

How can we do this? We want to explain the behaviours of the various operations, but we're not allowed to use anything outside what we already know about... the answer is to describe *how the different operators interact with each other*. In this case, for example, we can say

given any entry **x** and any stack **s**, it's always the case that

```
top (push x s) == x
```

which tells us something about **top** and something about **push**. This sort of rule, which is essentially just a logical statement that is required to hold in all situations, is called an ***axiom***, and this style of describing the semantics of an ADT is called ***axiomatic semantics***. A full set of these axioms, each assigned a unique name for later reference, is illustrated in Table 1.2.

Axiom	***Meaning***
[`empty.1`]: `emptyStack (createStack) == True`	A newly created stack is empty
[`empty.2`]: `emptyStack (push x s) == False`	Any stack onto which an entry has been pushed is non-empty
[`top.1`]: `top (createStack) == "the stack is empty"`	An empty stack doesn't have a top entry
[`top.2`]: `top (push x s) == x`	If you push an entry onto a stack, that entry becomes the top element
[`pop.1`]: `pop (createStack) == "the stack is empty"`	You can't pop an empty stack
[`pop.2`]: `pop (push x s) == s`	If you push an entry onto a stack and then pop it off again, you get back the original stack

Table 1.2: Semantics for **Stack a**

Notice how the axioms have been constructed. The various operators can be split into three groups, which we call *constructors*, *observers* and *mutators*.

- A ***constructor*** is an operator that builds new objects. In this case there are two constructors, **createStack** and **push**. The first of these, **createStack**, generates a new empty stack out of nothing, while the second, **push**, allows us to add entries to an empty stack so as to build bigger stacks.
- An ***observer*** is an operator that lets you examine an object without changing it in any way. In this case, we have two observers, **emptyStack** and **top**. The observer, **top**, allows us to examine the top entry in the stack (if one exists), while the observer, **emptyStack**, allows us to check whether or not a stack is empty.
- A ***mutator*** is an operator that changes one object into another, and which has not been classified as a constructor. In general, constructors help you to *build* new objects, whereas mutators either *rearrange* or *deconstruct* objects. In this case we have just one mutator. The operator **pop** deconstructs a stack by removing whichever entry (if any) is currently at the top.

The axioms address each observer and mutator in turn, asking how it interacts with each constructor. So, for example, the first two axioms ask how **emptyStack** behaves when applied after the constructors **createStack** and **push.** The next two ask what happens when the observer **top** is applied after each of the constructors. And the final two axioms ask what happens when the mutator **pop** is applied after **createStack** and **push**.

The reason we do things this way is straightforward. A stack is either empty or it isn't, and the behaviour of mutators and observers typically depends on which of the two situations applies. But what *is* a non-empty stack? The only way we can build a stack with entries in it is by **push**ing

them onto an empty stack. Consequently, any stack we think of will definitely match precisely one of two ***patterns***:

- either a stack is empty, in which case it matches the pattern **createstack**;
- or else it is non-empty, in which case it matches the pattern **push x s'**, for suitable choices of entry, **x**, and sub-stack, **s'**.

You should be familiar with this idea of matching patterns from your work on Haskell last semester.

1.5 Completeness

How complete are the syntax and semantics given above? Do they tell you *everything you need to know* about how to program a stack? The axioms tells us that pushing an entry onto a stack and then popping it off again results in the original stack, but what about the other way around? If the axioms are complete, they should enable to prove such facts about stacks as the following: *if you pop an entry off a non-empty stack, and then push it back on again, you'll get back the original stack.* Before deciding whether this is something we can prove, we need to work out how to write it down formally. Given that

- the condition 's is non-empty' can be written '`emptyStack s == False`';
- the entry we'll be pushing back onto the stack is '`top s`';

the statement we need to prove is

$$\texttt{[emptyStack s == False]} \Rightarrow \texttt{[push (top s) (pop s) == s]} \tag{1.1}$$

But *can* we prove it? According to the axioms, the only way `emptyStack s` can evaluate to `False` is if `s` is of the form `s == push x s'` for some `x` and `s'`. Formally,

$$\texttt{[empty s == False]} \Rightarrow \texttt{[s == push x s']} \quad \text{for some } \mathbf{x}, \mathbf{s'} \tag{1.2}$$

But in this case, we have

$$\begin{aligned} \texttt{top s} &\texttt{== top (push x s')} \\ &\texttt{== x} \qquad \text{by [}\texttt{top.2}\text{]:} \end{aligned} \tag{1.3}$$

and

$$\begin{aligned} \texttt{pop s} &\texttt{== pop (push x s')} \\ &\texttt{== s'} \qquad \text{by [}\texttt{pop.2}\text{]:} \end{aligned} \tag{1.4}$$

Putting all the pieces together, we can say

$$\texttt{[emptystack s == False]} \Rightarrow \texttt{[s == push x s']}$$

for some particular values of **x** and **s'**, whence

$$\begin{aligned} \texttt{push (top s) (pop s)} &\texttt{== push (top (push x s')) (pop (push x s'))} \\ &\texttt{== push x s'} \\ &\texttt{== s} \end{aligned} \tag{1.5}$$

as required. □

1.6 Summary

An ***abstract data type*** (ADT) is a data type which is defined by specifying its

- Types
  What types have to be defined before the ADT is itself fully defined?
- Syntax
  What operations are available, and what are their types?
- Semantics
  How are the operations related to one another?

The goal is to provide a *complete* semantics; a set of rules that is powerful enough to let you prove anything about the data type that ought to be true. In general you do this by identifying which of the operations are *constructors* and which aren't. You then provide one semantic rule for each non-constructor/constructor pair. That is, for each constructor **C** and each non-constructor **F** you write down a rule explaining the overall effect of 'first apply **C**, then apply **F**'.

1.7 How complete is our stack ADT?

I've suggested above that the ADT given above is complete, bgut a little though shows that this isn't quite true. The problem is, we've simply *assumed* that messages are strings — that's why I could write the rule

```
pop (createStack) == "the stack is empty"
```

In general, of course, messages could take many different forms; and we can't assume that the messages generated when **pop** fails are of the same type as the messages generated when **top** fails. After all, one is a message describing our inability to identify what *stack* should be returned by the function, and the other describes our inability to return a *value*. To be on the safe side we could another type into the definition, namely

- **Msg a**
  Messages generated when problems are encountered trying to identify a result of type **a**

together with two new constant functions

- **msgNoValue : Msg a**
- **msgNoStack : Msg (Stack a)**

In exactly the same way, we should identify **True** and **False** as new constant functions of type **Bool**, because otehrwise they're undefined values (remember, we can't assume anything is defined just because it's familiar from the programming languages we use — only the functions defined in the syntax exist as far as the ADT definition is concerned). Later in this section we'll see how to define the booleans as an ADT in their own right. Adopting this more complete approach gives the amended syntax and semantics shown in Table 1.3.

1.8 More Examples

In this section we'll consider another two ADTS, and show how they'd be defined. We'll look in more detail at how ADTs can be *used* later.

createStack	:	**Stack a**
emptyStack	:	**Stack a** $\rightarrow$ **Bool**
push	:	**a** $\rightarrow$ **Stack a** $\rightarrow$ **Stack a**
pop	:	**Stack a** $\rightarrow$ (**Stack a** $\cup$ **Msg** (**Stack a**))
top	:	**Stack a** $\rightarrow$ (**a** $\cup$ **Msg a**)
true	:	**Bool**
false	:	**Bool**
msgNoStack	:	**Msg** (**Stack a**)
msgNoValue	:	**Msg a**

Axiom	*Meaning*
[`empty.1`]: `emptyStack (createStack) == true`	A newly created stack is empty
[`empty.2`]: `emptyStack (push x s) == false`	Any stack onto which an entry has been pushed is non-empty
[`top.1`]: `top (createStack) == msgNoValue`	An empty stack doesn't have a top entry
[`top.2`]: `top (push x s) == x`	If you push an entry onto a stack, that entry becomes the top element
[`pop.1`]: `pop (createStack) == msgNoStack`	You can't pop an empty stack
[`pop.2`]: `pop (push x s) == s`	If you push an entry onto a stack and then pop it off again, you get back the original stack

Table 1.3: Amended syntax and semantics for **Stack a.**

1.8.1 Booleans

Booleans are used to represent notions like *true* and *false*; while some languages have the type built-in, others don't, so we need to specify **Bool** as an ADT. In addition to the two values **true** and **false**, we also need functions like **not**, **and** and **or**, which have the familiar signatures

- **true** : **Bool**
- **false** : **Bool**
- **not** : **Bool** $\rightarrow$ **Bool**
- **and** : **Bool** $\rightarrow$ **Bool** $\rightarrow$ **Bool**
- **or** : **Bool** $\rightarrow$ **Bool** $\rightarrow$ **Bool**

The *constructors* for this type — the functions which construct values out of nothing — are just **true** and **false**, and the semantics are the usual truth table definitions, for example:

- Semantics of **not**
  not true == **false**
  not false == **true**

Notice that this ADT doesn't use any auxiliary types; the only type that's mentioned in the syntax is **Bool** itself.

1.8.2 Natural Numbers

The set ℕ of natural numbers comprises the non-negative integers $0, 1, 2, \ldots,$ and we want to define an ADT **Nat** that ca[ptures their essential properties. If you look back at **Bool**, the reasion we could simply list the possible boolean values (**true** and **false**) is that there were only finitely many of them. The same trick obviously won't work for the natural numbers, because ℕ is infinite. Instead, we have to describe how we could *construct* each of the numbers.

Constructors. To start with, are there any 'special' natural numbers? Well yes — zero is special because it's the only one that dioesn't have a predecessor. So let's define a special value **Zero**. What about the others? Well, we've just identified a key property they all have in common — each of them is the *successor* of some other number. So let's introduce a function **succ** that maps each value n to its successor $n + 1$. Armed with **zero** and **succ** we can define any n we like: 1 is **succ zero**, 2 is **succ succ zero**, and so on.

Other functions. What other functions do we traditionally define on natural numbers? The obvious candidates are functions like **plus**, **times** and so forth. A possible syntax for the type is shown in Table 1.4.

zero	:	**Nat**		
succ	:	**Nat**	→ **Nat**	
plus	:	**Nat**	→ **Nat**	→ **Nat**
times	:	**Nat**	→ **Nat**	→ **Nat**

Table 1.4: A possible syntax for **Nat**. You could add further functions if you wanted to, like **minus** and **mod**.

Semantics. We know that the constructors are **zero** and **succ**, so we need to define what happens when we apply **plus** and **times** to values of the form **zero** and **succ x**. Let's concentrateon **plus** — **times** is virtually identical. Clearly, no matter what value of **x** we start with, we'll always have

plus y zero == y

But what about **plus y (succ x)**? The answer has to be either **zero** or **succ z** for some **z**, and clearly it can't be **zero**. So we need to find some expression **z** which makes the equation

plus y (succ x) == succ z

work; or putting it back into familiar maths, we need to find z such that $y + (x + 1) = z + 1$. The obvious answer is $z = y + x$, so the rule we want has to be

plus y (succ x) == succ (plus y x)

Types. The only type mentioned in this ADT's syntax is the type **Nat** itself. A more complete definition might also include the type **Bool**, for example if we want to have an **isZero** or an **isGreaterThan** function.

Chapter 2

ADTs and HASKELL

2.1 Introduction

Last week: we looked at the *stack*.

- **Types**
  - **a**
  - **Bool**
  - **Msg**
  - (**a** ∪ **Msg**)
  - (**Stack a** ∪ **Msg**)
- **Syntax**
  - **Constructors**
    - **createStack**: → **Stack a**
    - **push**: **a** → **Stack a** → **Stack a**
  - **Others**
    - **emptyStack**: **Stack a** → **Bool**
    - **top**: **Stack a** → (**a** ∪ **Msg**)
    - **pop**: **Stack a** → (**Stack a** ∪ **Msg**)
- **Semantics**
  - [empty.1]: `emptyStack (createStack) == True`
  - [empty.2]: `emptyStack (push x s) == False`
  - [top.1]: `top (createStack) == "the stack is empty"`
  - [top.2]: `top (push x s) == x`
  - [pop.1]: `pop (createStack) == "the stack is empty"`
  - [pop.2]: `pop (push x s) == s`

This week:

- If this really *is* a complete definition of a stack, how do we turn it into code?

- How do we actually *program* an abstract data type?
- The answer will depend on the language you decide to use; we'll be using Haskell. In general, however, the principles are the same no matter what language you use.
- First you need to identify which types in the language will correspond to the ADT's auxiliary types, and then you need to implement each of the ADT's operations.

2.1.1 An aside: Haskell's module system

One way to program an ADT in Haskell is to use the **module** system (covered last semester).

The use of **module**s to program ADTs is explained, with examples, in Simon Thompson's book.

The aim today is to introduce other techniques. Regardless of whether you use **module**s or some other technique, you will still need to address the questions raised below.

2.2 Either a b

Biggest problem: how to represent 'either **a** or **b**'. Many solutions (including *avoid the problem*).

The ***Either*** type is defined in the Haskell Prelude:

```
data Either a b = Left a | Right b
   deriving (Eq, Ord, Read, Show)
```

Either defines a whole family of data types, one for each choice of **a** and **b**. Some examples of types constructed using **Either** are

- **Either Integer Char**
  This is the Haskell type corresponding to (**Integer** $\cup$ **Char**). Typical values of type **Either Integer Char** include **Left 5**, **Right 'x'** and **Left 16**.
- **Either Bool Float**
  This is the Haskell type corresponding to (**Bool** $\cup$ **Float**). Typical values of type **Either Bool Float** include **Left True**, **Right 5.0** and **Left False**.

If we substitute for just one of **Either**'s type variables, we get a family of one-parameter data types. For example, other instances of **Either** include

- **Either a Msg**
  This is the Haskell type corresponding to (**a** $\cup$ **Msg**). Of course, we need to define the **Msg** type in Haskell as well, or this won't be properly defined. Typical values of type **Either a Msg** include **Left x**, for any **x** of type **a** and **Right "the stack is empty"**.

Finally, there's no reason we can't substitute types that are themselves parametric.

- **Either (Stack a) Msg**
  This is the Haskell type corresponding to (**Stack a** $\cup$ **Msg**). Typical values of type **Either (Stack a) Msg** include **Left createStack**, **Right "the stack is empty"** and **Left (push x s)**.

- **Either (Either Bool Integer) Char**
  This is the Haskell type corresponding to ((**Bool** ∪ **Integer**) ∪ **Char**). Typical values of this type include **Left (Left True)**, **Left (Right 5)** and **Right 'x'**.

As these examples show, however, if we decide to use the **Either** type, the values we write down are going to look fairly unintuitive. Ultimately this is once again a matter of personal choice - are we happy to put up with slightly more complicated values in exchange for having a standard way of representing types of the form 'either **a** or **b**'?

2.3 Implementation using Either

Now that we know about **Either**, we can start to implement **Stack a**.

2.3.1 Types

The first thing to notice is that Haskell allows us to leave **a** undefined. This is unusual, and reflects the fact that Haskell is a functional language equipped with the means to represent type variables. Languages like JAVA and C^{++} don't support type variables (except as specific parameters in `templates`), so if we were implementing **Stack a** in one of those languages we would have to decide precisely what type to use to represent entries. As always, the final choice as to which type is which is up to you as the ADT's designer. I choose the following types:

Msg	`String`
Bool	`Bool`
(**Stack a** ∪ **Msg**)	`Either (Stack a) String`
(**a** ∪ **Msg**)	`Either a String`

2.3.2 Constructors

We've identified two of the operations defined in **Stack a**'s syntax as constructors. We can reflect this directly using Haskell's **data** construction.

```
data Stack a = EmptyStack | Push a (Stack a)
```

This definition says that, for any type **a**, **Stack a** is a data type whose elements come in two forms. Either a stack is an **EmptyStack**, or else it can be written in the form **Push x s** for some entry **x** and some existing stack **s**.

The constructors themselves can now be defined directly:

```
-- definition of createStack: -> Stack a
createStack :: Stack a
createStack = EmptyStack

-- definition of push: a -> Stack a -> Stack a
push :: a -> Stack a -> Stack a
push x s = Push x s
```

2.3.3 Non-constructors

The other three operations can be defined just as easily. Notice that the Haskell definitions for these functions are just the rules we worked out for the semantics! Having worked out the ADT definition of a **Stack a**, there's nothing left to do; all the equations are already defined for us.

```
-- definition of emptyStack: Stack a -> Bool
emptyStack :: Stack a -> Bool
emptyStack EmptyStack = True
emptyStack (Push _ _) = False

-- definition of top: Stack a -> Either a String
top :: Stack a -> Either a String
top EmptyStack = Right "the stack is empty"
top (Push x _) = Left x

-- definition of pop: Stack a -> Either (Stack a) String
pop :: Stack a -> Either (Stack a) String
pop EmptyStack = Right "the stack is empty"
pop (Push _ s) = Left s
```

Typing these definitions into Haskell environment shows the following evaluations:

```
push 5 createStack
    --> Push 5 EmptyStack :: Stack Integer

pop (push 5 createStack)
    --> Left EmptyStack :: Either (Stack Integer) String
```

2.3.4 Problems with this approach

Although it looks like we've simply copied out the rules that occurred in the ADT's semantics, this isn't quite true; we've had to add various instances of **Left** and **Right**. These cause very definite problems, as becomes obvious is we try to evaluate

pop (pop (push 5 (push 6 EmptyStack))) .

Consider what the result ought to be: starting with the empty stack, we first push two integers onto the stack and then pop them off again. The result ought to be the empty stack again. In fact, however, the Haskell environment gives the evaluation

```
Main> pop (pop (push 5 (push 6 EmptyStack)))
ERROR - Type error in application
*** Expression     : pop (pop (push 5 (push 6 EmptyStack)))
*** Term           : pop (push 5 (push 6 EmptyStack))
*** Type           : Either (Stack b) String
*** Does not match : Stack a
```

The problem lies in the return type of the first **pop**, which is **Either (Stack a) String.** In order to apply **pop** a second time, we need the argument to be of type **Stack a**. Somehow, we need to get rid of the **Either** types in our definitions. But how can we do this?

2.4 Using an invariant assertion

It looks like using **Either** is causing unnecessary problems, but in fact it's much worse than this. The problem lies in the ADT definition itself. There's no obvious reason why we should rule out applying **pop** twice in a row (as long as the stack contains at least two entries), and yet our syntax won't let us do this. The types simply don't match up:

1st **pop**:	`Stack a` →	(**Stack a** ∪ **Msg**)	
2nd **pop**:		**Stack a**	→ (`Stack a` ∪ `Msg`)

Applying **pop** once generates a result of type (**Stack a** ∪ **Msg**), but the second **pop** requires an argument of type **Stack a**. How can we avoid this problem?

A standard solution is to give what's called an ***invariant assertion***. We simply state, for example, that *whenever a function is applied to a **Msg**, it simply returns its argument unchanged.* In other words, all values of type **Msg** are left ***invariant*** by every operation that acts on them. So, if applying **pop** the first time generates a stack as its result, the second **pop** operates on this stack as usual; if the first **pop** generates the message *the stack is empty*, then the second **pop** *also* generates this result; it passes the message on unchanged. Unfortunately, while this tells us how to *evaluate* what happens when we apply **pop** twice in row, it doesn't actually solve our typing problem!

Clearly, we need the argument type and the return type of **pop** to essentially the same type. So either we need to change **pop** so it can explicitly be applied to arguments of type (**Stack a** ∪ **Msg**), or else we need to remove the need for **Msg** in the return type. Both approaches are viable.

2.4.1 Changing pop's argument type

This solution simply means that **pop**'s signature has to be changed to

pop: (**Stack a** ∪ **Msg**) → (**Stack a** ∪ **Msg**)

but, of course, that's not the only change that's needed. We have to check what happens when *each* of the operations is combined with each of the others. Worse still, as we change the argument types of each operation, we will need to re-address these mismatches, so the entire process is iterative. We keep going until all the signatures are consistent with one another. Starting again with '**pop** then **pop**', the process might begin:

- **pop** then **pop**.
  Got: **pop** returns (**Stack a** ∪ **Msg**)
  Got: **pop** requires **Stack a**
  Change to: **pop**: (**Stack a** ∪ **Msg**) → (**Stack a** ∪ **Msg**)
- **pop** then **push**.
  Got: **pop** returns (**Stack a** ∪ **Msg**)
  Got: **push** requires **a** and **Stack a**
  Change to: **push**: **a** → (**Stack a** ∪ **Msg**) → (**Stack a** ∪ **Msg**)

  and so on . . .

In some ways this is the best solution available to us. Although there are neater answers, they typically rely on making specific compromises or design decisions, whereas the current solutionrem ains truly general. The overall solution isn't particularly complicated, just a little messy:

- **Types**
  - **a**, **Bool**, **Msg**
  - (**a** ∪ **Msg**), (**Stack a** ∪ **Msg**)
- **Syntax**
  - **createStack**: → (**Stack a** ∪ **Msg**)
  - **push**: (**a** ∪ **Msg**) → (**Stack a** ∪ **Msg**) → (**Stack a** ∪ **Msg**)
  - **emptyStack**: (**Stack a** ∪ **Msg**) → (**Bool** ∪ **Msg**)
  - **top**: (**Stack a** ∪ **Msg**) → (**a** ∪ **Msg**)
  - **pop**: (**Stack a** ∪ **Msg**) → (**Stack a** ∪ **Msg**)
- **Semantics**
  - [push.1]: `push msg s == msg` if msg::Msg, s::Stack a
  - [push.2]: `push x msg == msg` if x::a, msg::Msg
  - [push.3]: `push msg msg' == joinMsg(msg,msg')` if msg, msg'::Msg, where `joinMsg` denotes some form of message combination
  - [empty.1]: `emptyStack (createStack) == True`
  - [empty.2]: `emptyStack (push x s) == False` if x::a, s::Stack a
  - [empty.3]: `emptyStack msg == msg` if msg::Msg
  - [top.1]: `top (createStack) == "the stack is empty"`
  - [top.2]: `top (push x s) == x` if x::a, s::Stack a
  - [top.3]: `top msg == msg` if msg::Msg
  - [pop.1]: `pop (createStack) == "the stack is empty"`
  - [pop.2]: `pop (push x s) == s` if x::a, s::Stack a
  - [pop.3]: `pop msg == msg` if msg::Msg

Given this new ADT definition, we can construct a Haskell implementation using **Either**, much as before. The important thing to note is that we can't do tests to find out what type the various arguments are, so we need to use care when converting the semantics into implementation rules.

```
type Msg = String
joinMsg :: Msg -> Msg -> Msg
joinMsg msg _ = msg -- Keep it simple.
                    -- Just use the first of the two messages

data Stack a = EmptyStack | Push a (Stack a)
    deriving (Show, Eq)


createStack :: Either (Stack a) Msg
createStack = Left EmptyStack

-- [push.1] push msg s == msg if msg::Msg, s::Stack a
-- [push.2] push x msg == msg if x::a, msg::Msg
-- [push.3] push msg msg' == joinMsg(msg,msg') if msg, msg'::Msg
```

```
push :: Either a Msg -> Either (Stack a) Msg -> Either (Stack a) Msg
push (Left x)    (Left s)     = Left (Push x s)
push (Right msg) (Left _)     = Right msg                -- [push.1]
push (Left _)    (Right msg)  = Right msg                -- [push.2]
push (Right msg) (Right msg') = Right (joinMsg msg msg') -- [push.3]


-- Easy version so I can just push values without having to "Left"-ify them
pushEasy :: a -> Either (Stack a) Msg -> Either (Stack a) Msg
pushEasy x = push (Left x)

-- IMPLEMENT EMPTYSTACK
-- [empty.1] emptyStack (createStack) == True
-- [empty.2] emptyStack (push x s) == False if x::a, s::Stack a
-- [empty.3] emptyStack msg == msg if msg::Msg

emptyStack :: Either (Stack a) Msg -> Either Bool Msg
emptyStack (Left EmptyStack) = Left True
emptyStack (Left _) = Left False
emptyStack (Right msg) = Right msg

-- IMPLEMENT TOP
-- [top.1] top (createStack) == "the stack is empty"
-- [top.2] top (push x s) == x if x::a, s::Stack a
-- [top.3] top msg == msg if msg::Msg

top :: Either (Stack a) Msg -> Either a Msg
top (Left EmptyStack) = Right "the stack is empty"  -- [top.1]
top (Left (Push x _)) = Left x                      -- [top.2]
top (Right msg)       = Right msg                   -- [top.3]

-- IMPLEMENT POP
-- [pop.1] pop (createStack) == "the stack is empty"
-- [pop.2] pop (push x s) == s} if x::a, s::Stack a
-- [pop.3] pop msg == msg if msg::Msg

pop :: Either (Stack a) Msg -> Either (Stack a) Msg
pop (Left EmptyStack) = Right "the stack is empty"  -- [pop.1]
pop (Left (Push _ s)) = Left s                      -- [pop.2]
pop (Right msg)       = Right msg                   -- [pop.3]

-- Try out a double-pop to see what happens
test = pop (pop (pushEasy 5 (pushEasy 6 (createStack))))
```

which causes `test` to evaluate, as expected, to

```
Left EmptyStack :: Either (Stack Integer) Msg
```

2.5 Removing the need for Msg

A simpler solution would be to remove the need for **Msg** altogether, but to do this we need to remember why **Msg** was introduced in the first place.

The problem was that sometimes it doesn't make sense to apply an operation; for example, it doesn't make sense to apply **top** to the empty stack.

We included **Msg** as a way of letting the user know that they'd done something silly; rather than leaving them in the dark when they try applying **top** after **createStack**, we provide them with the feedback message, 'the stack is empty'.

But in fact there's no need for this. We could set aside a specific value of type **Stack a**, which we return whenever we would previously have returned a message.

- People do this all the time at the concrete (*i.e.*, programming) level; for example, you might define a function that returns integers, but with the understanding that negative results indicate error conditions of some sort.
- Because **top** also returns 'error' messages, we would also need to set aside a special value of type **a**.
- Taking this approach would allow us to simplify the ADT definition quite significantly in some ways, but complicates it in others.
- In particular, we are making *design decisions* when we should be keeping things as abstract as possible.

We should also remember that it's not always *possible* to set aside a special 'error value'. For example, suppose we wanted to program a division operator for integers; we'd need to pick some value to represent *division-by-zero*, but there is no value we can choose that might not actually arise quite naturally as the result of a valid division.

- **Types**
  - **a**, **Bool**
- **Syntax**
  - **aErr**: $\rightarrow$ **a**
  - **stackErr**: $\rightarrow$ **Stack a**
  - **createStack**: $\rightarrow$ **Stack a**
  - **push**: **a** $\rightarrow$ **Stack a** $\rightarrow$ **Stack a**
  - **emptyStack**: **Stack a** $\rightarrow$ **Bool**
  - **top**: **Stack a** $\rightarrow$ **a**
  - **pop**: **Stack a** $\rightarrow$ **Stack a**
- **Semantics**
  - [empty.1]: `emptyStack (createStack) == True`
  - [empty.2]: `emptyStack (push x s) == False`
  - [empty.3]: `emptyStack (stackErr) == False` (it's up to us how we define this)
  - [top.1]: `top (createStack) == aErr`
  - [top.2]: `top (push x s) == x`
  - [top.3]: `top (stackErr) == aErr`
  - [pop.1]: `pop (createStack) == stackErr`
  - [pop.2]: `pop (push x s) == s`

- [pop.3]: `pop (stackErr) == stackErr`
- [push.1]: `push x stackErr == stackErr`
- [push.2]: `push aErr s == stackErr`

where **aErr** and **stackErr** are the specific entities of types **a** and **Stack a**, respectively, used solely to indicate error conditions. Notice, however, that we now need to add several extra rules to the semantics, so we know what happens if people try to operate on these special values. In particular, we've had to add two new rules for **push**.

2.6 Exploiting side-effects using error

That's just about all we can do at the abstract level, but when we try to *implement* the ADT, we can take advantage of specific language features, and in particular we can exploit *side-effects*. In Haskell, we can do this using the **error** function. Recall that this is a hard-wired function, defined in the Haskell Prelude as

$$\texttt{error :: String} \rightarrow \texttt{a}$$

The important point about **error** is that it allows us to *generate* a message to the user, without actually *returning* the message as a return value.

- If we use the error mechanism, we can simplify things somewhat.
- Wherever we previously returned a value of type (**a** ∪ **Msg**), we can now return one of type **a** instead, and
- wherever we returned a value of type (**Stack a** ∪ **Msg**), we can return one of type **Stack a**
- i.e., wherever a message would have been required, we use **error** to generate it instead.
- That is, we use Haskell's **error** system to simplify the syntax for **Stack a**.
- We no longer need to consider the special error values **aErr** and **stackErr**, and so we no longer need to introduce extra rules into the semantics.
- As before, we can use Haskell's **data** construction, and as before the function implementations are simply the semantic rules written using Haskell syntax.

```
data Stack a = EmptyStack | Push a (Stack a)

-- definition of createStack: -> Stack a
createStack :: Stack a
createStack = EmptyStack

-- definition of push: a -> Stack a -> Stack a
push :: a -> Stack a -> Stack a
push x s = Push x s


-- definition of emptyStack: Stack a -> Bool
emptyStack :: Stack a -> Bool
emptyStack EmptyStack = True                -- [empty.1]
emptyStack (Push _ _) = False               -- [empty.2]
```

```
-- definition of top: Stack a -> a
top :: Stack a -> a
top EmptyStack = error "the stack is empty" -- [top.1]
top (Push x _) = x                          -- [top.2]

-- definition of pop: Stack a -> Stack a
pop :: Stack a -> Stack a
pop EmptyStack = error "the stack is empty" -- [pop.1]
pop (Push _ s) = s                          -- [pop.2]
```

As expected, these new type signatures mean that we can apply as many operators as we like in sequence; if the result *ought* to be defined, it will be. For example:

```
    pop (pop (push 5 (push 6 EmptyStack)))
--> pop (pop (push 5 (Push 6 EmptyStack)))
--> pop (pop (Push 5 (Push 6 EmptyStack)))
--> pop (Push 6 EmptyStack)
--> EmptyStack :: Stack Integer
```

2.7 Using Haskell's class system

We'll look at this in more detail later in the course, but for now it's worth noting that Haskell has a **class** system that can be used for implementing ADTs. **Important**: This has *virtually nothing* to do with object-oriented 'classes' in languages like JAVA and C^{++}.

A ***type class*** is a set of types that share the same polymorphic functions. For example, the class **Eq a** comprises those types **a** for which the functions (==) and (/=) are defined. Formally, the class is introduced in the Haskell Prelude by the declaration

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

To define a class you use the **class** keyword, give the name of the class, and then say precisely what functions have to be available if a type is to count as a member of the class. For example, we can say that an arbitrary type **adt** is a member of the **Stack** type class by writing

```
class Stack adt where
  createStack :: adt
  push :: adt -> adt -> adt
  emptyStack :: adt -> Bool
  top :: adt -> adt
  pop :: adt -> adt
```

This tells us that a data type **adt** *can be used as* a stack provided it has some way of representing each of the operators in question. There are many data types can be used in this way. For example, a common technique is to represent a stack as a *list*. In order to fit in with the class definition given above, we'd also need to represent individual entries as lists, and we can do this easily by representing the entry **e** as the singleton-list **[e]**. Given this interpretation, we tell Haskell that lists can be used as stacks by using the ***instance*** keyword, and telling Haskell how to implement each of the ADT's operations.

```
instance Stack [a] where
  createStack   = []
  push [e] s    = e:s
  push  _  _    = error "invalid entry pushed onto stack"
  emptyStack [] = True
  emptyStack _  = False
  top        [] = error "the stack is empty"
  top     (e:_) = [e]
  pop        [] = error "the stack is empty"
  pop     (_:s) = s
```

The first line of this definition says that 'one instance of the class **Stack** is the type **[a]**', provided we define the required operations as shown in the list of implementations that appear within the scope of the ***where*** keyword. In this instance the empty stack is represented as the empty list `[]`, and the **top** element of a list is the list `[e]` containing the top element.

We don't *have* to use existing data types if we don't want to. If we prefer, we can define our own algebraic data types and declare them to be instance of the **Stack** class. For example, the following definitions introduce a new algebraic data type **StackADT a** for storing entries of type **a**, in which the empty stack and the **push** constructor are defined in the usual way. Because of the way I've defined the **Stack** class' operations, entries also need to be expressed as elements of type **StackADT**, so we represent them as stacks containing just the one entry. So, for example, the rule '`top (Push x _) = Push x EmptyStack`' says that the top element of a non-empty stack is represented as the one-element stack whose only entry is the top element in question.

```
data StackADT a = EmptyStack | Push a (StackADT a)
  deriving (Show, Eq)
makeEntry :: a -> StackADT a
makeEntry x = Push x EmptyStack

instance Stack (StackADT a) where
  createStack              = EmptyStack
  push (Push x EmptyStack) s = Push x s
  push       _             _
     = error "invalid entry pushed onto stack"
  emptyStack EmptyStack      = True
  emptyStack _               = False
  top EmptyStack             = error "the stack is empty"
  top (Push x _)             = Push x EmptyStack
  pop EmptyStack             = error "the stack is empty"
  pop (Push _ s)             = s
```

As these examples show, there can be many different instances of any given type class, and there's nothing intrinsically better about one instance rather than another. It is your job as a designer to choose the best instance for your own purposes - it is precisely this advantage, that *ADTs give you this freedom to choose the best implementation for the task at hand*, that justifies the decision to generate the ADT definition in the first place.

2.8 Summary

There are many different ways to implement an ADT in Haskell.

- If we use the **data** construct to declare the ADT as an algebraic data type, the rules we worked out for the ADT's semantics turn out to be exactly the formulae we need to define the ADT's operations.
- An alternative solution is to use Haskell's **class** system. Rather than define *new* data structures to represent the ADT, the **class** system lets us tell Haskell how to use *existing* data types to represent the ADT.
- Whichever approach we use, however, we need to check that the function signatures all make sense - it should be possible to apply the ADT's operations one after another without unexpected type-mismatches occurring.
- This can be done in different ways. We we can examine all of the argument and return types for our operations, and change them wherever necessary to ensure that all the types are consistent with one another.
- Another approach is to remove the need for things like error messages by exploiting Haskell's **error** function, which allows us to generate messages as side-effects.

Chapter 3

Program proof

3.1 Introduction

This week we'll be discussing ***program proof***, and in particular how you prove things about Haskell programs and data structures, like when we proved for all stacks **s** of type **Stack a**, that

```
[emptyStack s == False]
  ⇒ [push (top s) (pop s) == s]
```

As we'll see, the type of semantics you decide to use affects *the way* you prove things about your ADT. But it shouldn't affect *what* you can prove.

3.2 How can we establish program correctness?

It's often claimed that guaranteeing the correctness of a program is literally impossible, and that the best we can do is *test* our code exhaustively. Since it's also claimed that testing can never establish beyond doubt that a program meets its specification, things are looking bad for the software engineering industry! But how valid are these claims?

In fact, there are at least **three different approaches** to ensuring that a given piece of code correctly implements its specification.

3.2.1 Correctness via testing

Books on software testing are keen to point out that the goal of testing is to *locate faults*. If a test fails to do this, it doesn't mean that there are no faults present — simply that the test may not be good enough.

Theory tells us that the question 'does an arbitrary program *prog* happen to satisfy an arbitrary specification *spec*?' is **formally undecidable**. In other words, it is simply not possible to construct an algorithm *SatisfiesSpec* with the property that

$$SatisfiesSpec(prog, spec) =$$

$$\begin{cases} True & \text{if } prog \text{ behaves as described by } spec \\ False & \text{otherwise} \end{cases}$$

for every possible choice of *prog* and *spec*.

Formally, then, it looks like we haven't a hope of establishing the correctness of a program through testing, because any testing algorithm that's guaranteed to work would be an implementation of the 'impossible' algorithm *SatisfiesSpec*.

In fact, however, things aren't that bad. The result says there's no way to implement the program *SatisfiesSpec* if we insist that it works

- for *every prog*, and
- for *every spec*.

But it says nothing about what happens if we restrict attention to *specially-chosen collections* of programs and specifications.

By exploiting this loophole, Holcombe and Ipate [HI98] developed a procedure that *can* ensure the correctness of an implementation by testing. To get round the undecidability problem described above, they insist that, far from being arbitrary, the program being testing has to be developed with the specification very much in mind.

Aside from a few technicalities, they show that

> Provided *prog* satisfies certain ***design-for-test*** conditions derived from careful analysis of *spec*, it is possible to generate a *finite* set T of tests (again derived from careful analysis of *spec*), with the following property: if *prog* passes each of the tests in T, then *prog* is *guaranteed* to be a correct implementation of *spec*.

So, provided you take away the uncertainties that lead to undecidability, testing *can* be good enough to guarantee correctness.

3.2.2 Correctness by construction

Another very well established approach is to guarantee ***correctness by construction***. This *formal methods* technique works by *transforming* the specification, step by step, into a working piece of code. All that's required that each of the transformation steps is logically sound. The idea that a program written in one language can be transformed systematically into a program written in another language is nothing new - compilers do it all the time. The only difference is that, while a compiler might translate a program written in JAVA into a sequence of bytecodes, a formal methods system might transform a program written in Z into a program written in Haskell. The process of converting specifications into code is called ***refinement***, and is the subject of numerous national and international conferences.

3.2.3 Correctness by program proof

Finally, we can always attempt to *prove* that our program is correct, an approach that's been in use since at least the development of *Floyd-Hoare logic* in the 1960s [Flo67, Hoa69]. This is the approach we'll be considering in more detail this week. In addition, we'll consider the way that Haskell definitions allow us to prove the correctness of certain implementations [Tho99].

3.3 Floyd-Hoare Logic

Floyd-Hoare Logic was introduced in the late 1960s as a technique for establishing the correctness of basic imperative programs. Although this means it's not directly applicable to functional

programming and ADTs, the underlying ideas can be applied in other ways, and are worth studying in detail.

Floyd-Hoare logic works by considering *pre-* and *post-conditions.* A ***pre-condition*** is something that *happens* to be true before a staement (or program) is executed, and a ***post-condition*** is something that *must necessarily* be true after it terminates. If P and Q are logical statements, and S is some program code, we write

$$\{P\}S\{Q\}$$

to mean

> If P is true before we run S, then when S finishes we can be certain that Q is true

Seen in this way, the program statement S can be thought of as something that *transforms* the precondition P into the postcondition Q. There are various rules telling us what form this transformation ought to take.

3.3.1 Transformation rules

Changing the precondition

If we can establish that a postcondition Q holds after executing the statement S, given that the preondition P held beforehand, and it turns out that R implies P, then we can justifiably replace P with R. That is,

$$\text{If } \{P\}S\{Q\} \text{ and } R \Rightarrow P, \text{ then } \{R\}S\{Q\}.$$

Changing the postcondition

If we can establish that a postcondition Q holds after executing the statement S, and it turns out that Q implies R, then we can justifiably claim that R holds after executing S. That is,

$$\text{If } \{P\}S\{Q\} \text{ and } Q \Rightarrow R, \text{ then } \{P\}S\{R\}.$$

Rule for assignments

Suppose P is valid before we execute the assignment `x = e`. What will be true afterwards? Those parts of P that told us about x will probably need changing; those parts of P that told us about other variables may not. Either way, we can be certain of one thing: x will be equal to e after the execution. For example, suppose we have the precondition $\{x = ab\}$, and then perform the assignment `x = x - 5`. When we've finished, the value of x will have dropped by 5; since it was equal to ab before the statement was executed, it must now be equal to $ab - 5$. Pictorially, we represent this as the annotated flow-graph in Figure 3.1.

Rule for `if` statements

An `if` statement

`if` C `then` $S1$ `else` $S2$

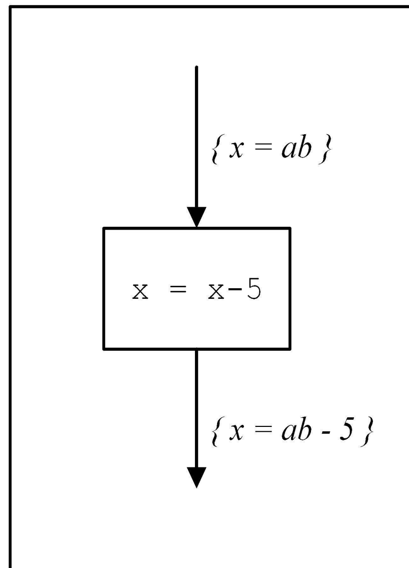


Figure 3.1: Annotated flow-graph showing the effect of an assignment

tests a constraint C, and chooses one of two different sub-programs depending on whether C evaluates to *True* or *False*. Pictorially, we can represent the `if` statement as the following annotated flow-graph in Figure 3.2.

Notice the postconditions on the two routes out of the `if` statement.

Rule for `while` statements

A `while` statement

```
while (C) { S }
```

tests the constraint C one or more times, and each time it eveluates to *True*, the block of code, S, is executed. For reasons that will become clear, the pictorial representation is slightly more complicated than might be expected (see Figure 3.3).

The extra feature here is the logical statement INV that's been added just above the `while` statement. This particular bit of the flow diagram can be traversed in two different ways; we traverse it simply by starting the program; and we also traverse it every time we complete the `while`-loop. The idea is to find some logical description of the program's variables that holds true before we enter the loop for the first time, and which remains true no matter how many times the loop is executed. It's this last feature that gives INV its name – we call it a ***loop invariant***.

3.3.2 Using the annotations

Suppose we've managed to annotate the complete flow-diagram for some program we're interested in, in such a way that all of the annotations make sense. That is, no matter which route we follow through a statement-box, the resulting statement $\{P\}S\{Q\}$ is valid. How does this help us prove that the program is actually *correct*? The proof actually requires two steps.

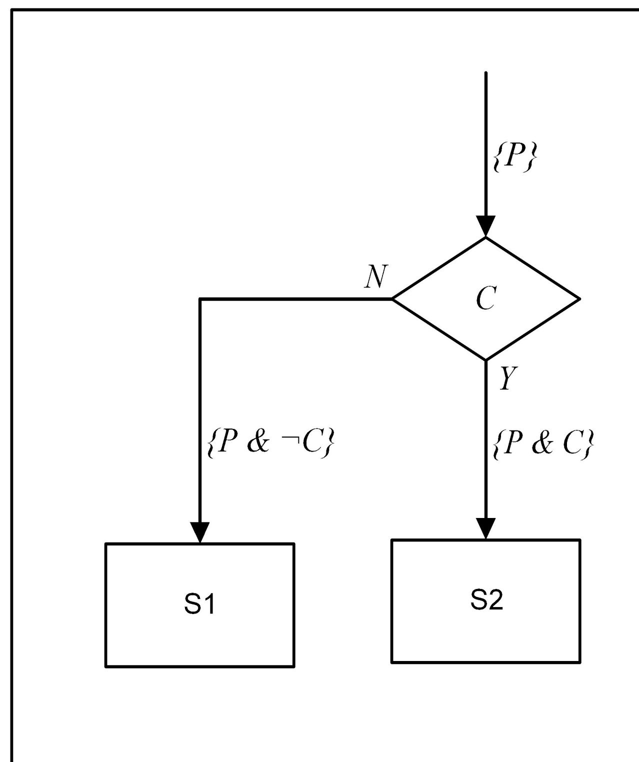


Figure 3.2: Annotated flow-graph showing the effect of an `if` statement

Proving termination

First, we have to show that the program will definitely halt at some point – if it doesn't, we won't get an answer out of it. This is easier than you might think; all we have to do is show that each *loop* will eventually terminate, and to do this we simply find some variable (or combination of variables) whose value necessarily decreases each time we go round the loop, but which is known to be bounded below. Given such a variable, the loop *must* eventually terminate, because the variable will eventually go below its lower bound otherwise. Equivalently, we can find some variable (or combination of variables) which increases each time round the loop, and which is bounded above.

For example, how do we know that the program

```
x = 10;
while ( x > 0 ) {
 x = x - 3;
}
```

will eventually halt? Clearly, the value of x necessarily decreases each time round the loop; but we know the value is bounded below, in the sense that the loop will exit as soon as x falls below 0. Since x definitely starts out being bigger than 0, we know that this bound will eventually be reached, and then the program will halt.

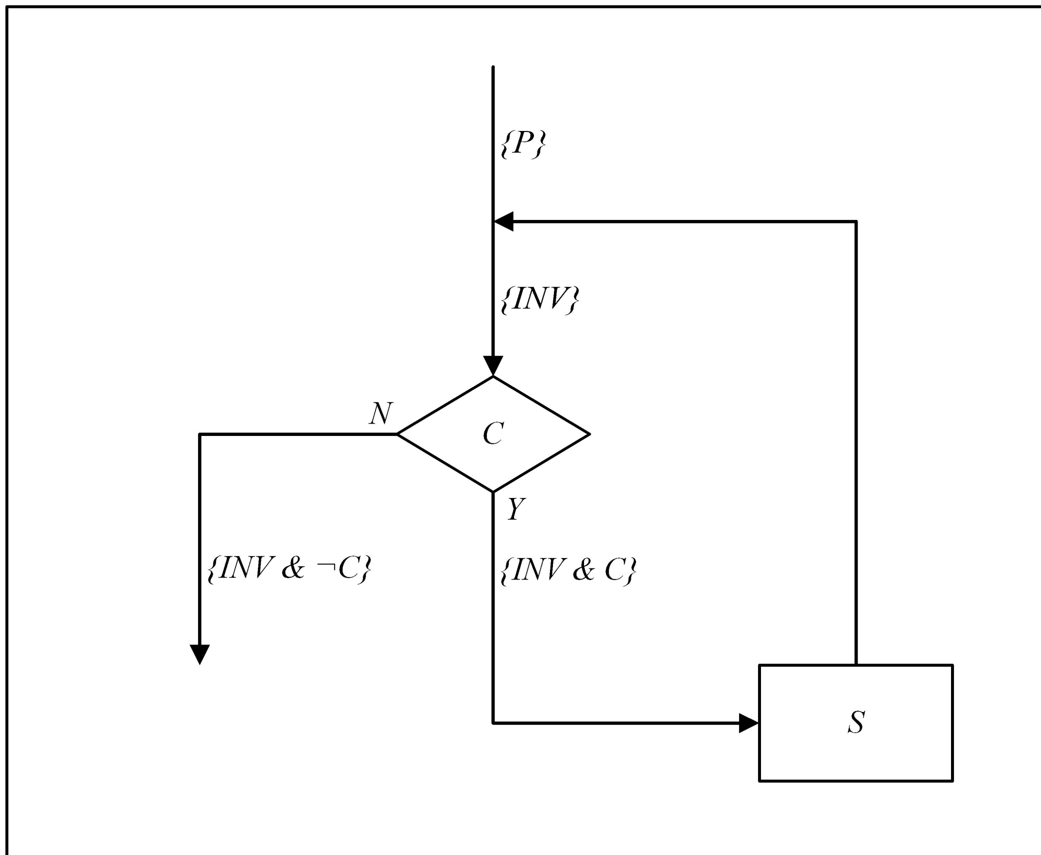


Figure 3.3: Annotated flow-graph showing the effect of a `while` statement

Proving correctness

Secondly, we have to prove that the program has computed the right answers *when* it terminates. This is where the Floyd-Hoare annotations come in. As the program exits, we'll be able to follow the flow of control along some final annotated arrow in the flow diagram, and that arrow will carry some logical statement P. We know that this staement *must* be a valid description of the various variables and other values relevant to the program, so all we have to do is show that P is sufficient to prove correctness.

For example, suppose our program is supposed to be an implementation of the Z-specification

Five

$\Delta[x : \mathbb{N}]$

$0 \leqslant x' \leqslant 5$

which says that whatever value the non-negative integer variable `x` had initially, it should end up with a value between 0 and 5.

Here's a simple imperative program (written in a JAVA- or C^{++}-like language) that does the job:

```
while (x > 5) {
 x = x - 5;
}
```

Figure 3.4 shows the associated flow-diagram.

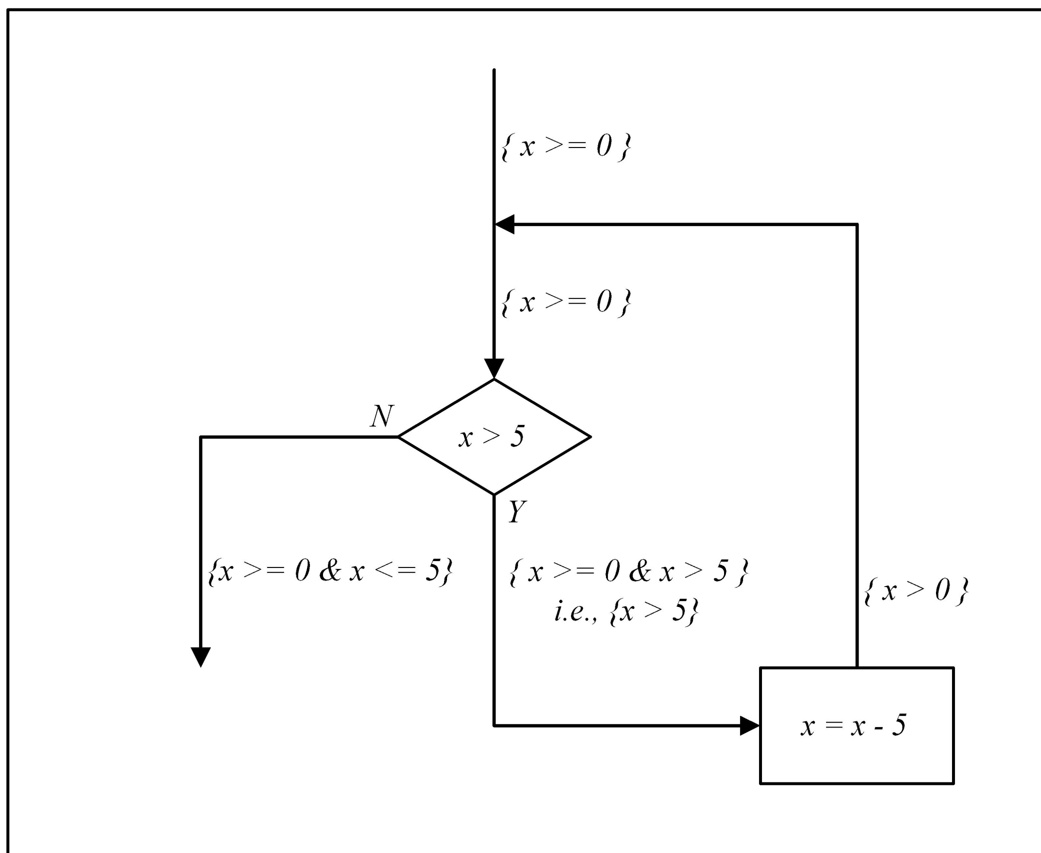


Figure 3.4: Annotated flow-graph for the program `while (x > 5) { x = x - 5 }`

3.3.3 Example: Euclid's Algorithm

Euclid's Algorithm is an ancient technique used to find the highest common factor of two integers. It's a very powerful algorithm, and is still used today. Indeed, the Haskell Preluide uses it in its definition of the `gcd` (grestest common divisor) function. Here's a (slightly modified) version of the definition, for use with non-negative integers.

```
gcd       :: Int -> Int -> Int
gcd x y
 | x <= 0 = error "positive parameters only"
 | y <= 0 = error "positive parameters only"
 | otherwise = gcd' x y
               where gcd' x 0 = x
                     gcd' x y = gcd' y (x `rem` y)
```

Although Haskell is a 'stateless' language, *i.e.*, we don't have any variables whose values change during the execution of a program, we can still represent `gcd` with a flow-diagram. Our aim is to show that the implementation given above works; in other words, we want to show that `gcd` computes the greatest common divisor of x and y. Since the greatest common divisor d of x and y is defined by

$$(x \equiv 0|d|) \wedge (y \equiv 0|d|) \wedge$$
$$(\forall\, n \in \mathbb{N})[(x \equiv 0|n|) \wedge (y \equiv 0|n|) \Rightarrow (d \geqslant n)]$$

our task is to show that `gcd` satisfies the Z-specification

GCD
$x?, y? : \mathbb{Z}$
$d! : \mathbb{N}$
$(x? > 0) \wedge (y? > 0) \wedge$
$(x? \equiv 0|d|!) \wedge (y? \equiv 0|d|!) \wedge$
$(\forall\, n : \mathbb{N})[(x? \equiv 0|n|) \wedge (y? \equiv 0|n|) \Rightarrow (d! \geqslant n)]$

Figure 3.5 shows the flow diagram for `gcd`, where I've taken account of the two `if` statements.

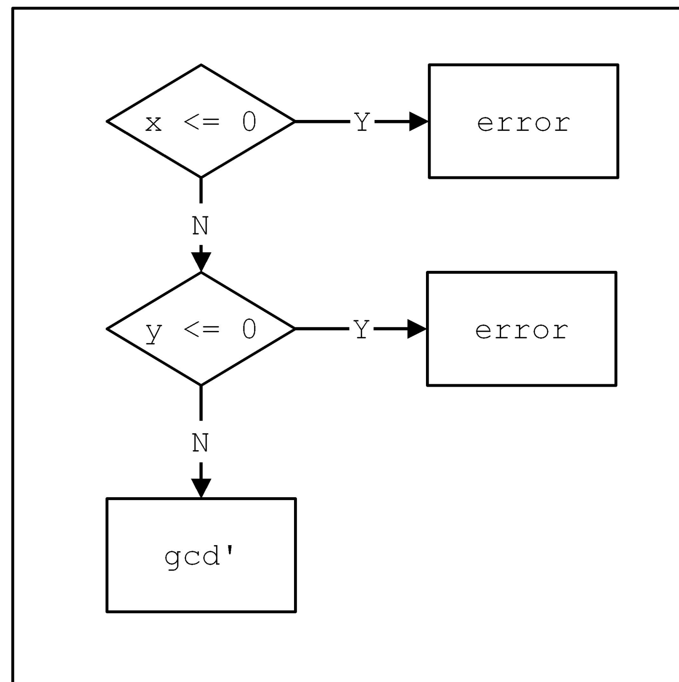


Figure 3.5: Basic flow-diagram for `gcd`

Next, we need to work out the flow diagram for `gcd'` – see Figure 3.6.

We start with an `if` statement, to test whether y is 0.

If $y = 0$, we return the value of x; I've represented this by adding an extra variable called *result*, to which the value x is assigned.

If $y \neq 0$, we need to call `gcd'` again, but with y in place of x, and $(x\,\text{`}rem\text{`}\,y)$ in place of y.

Notice that these two substitutions have to be made simultaneously – if you prefer to have just one statement executing at a time, you can introduce a temporary variable:

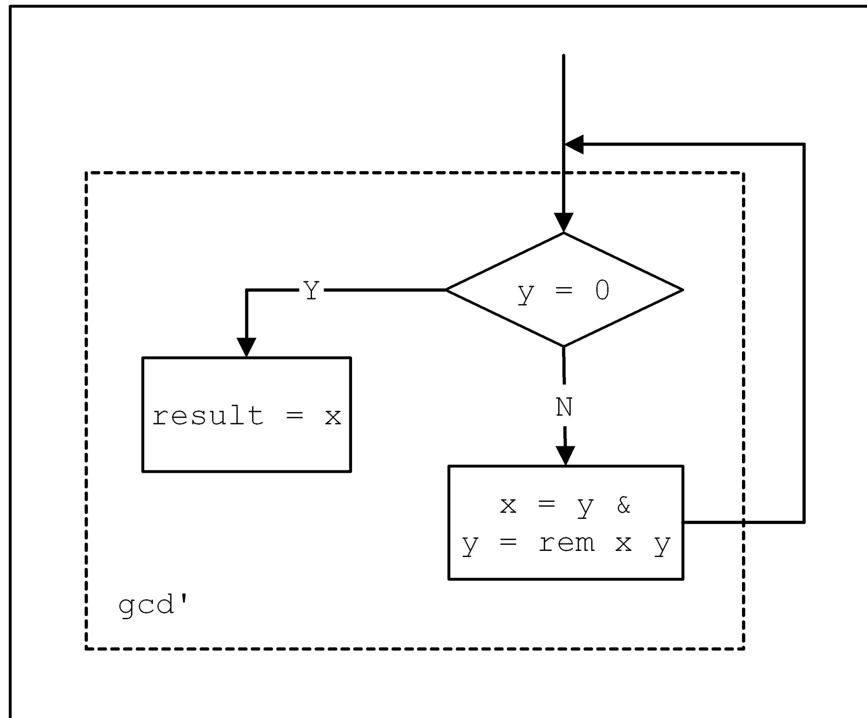


Figure 3.6: Flow-diagram for `gcd'`

```
tmp = x;
x = y;
y = tmp `rem` y
```

Combining the two flow-diagrams gives us a diagram for the entire program. Figure 3.7 shows the flow-diagram, complete with annotations, and one extra feature – because I need to keep track of the original values of x and y, I've added two extra variables a and b to operate upon instead. In constructing the various logical assertions, I've also taken note of the definition of `rem` given in the Haskell Prelude. This function implements the *modulo* function, *i.e.*, `rem x y` is the remainder when x is divided by y.

The loop invariant is valid

The first thing we need to do is show that the loop invariant, $gcd(x, y) = gcd(a, b)$, really is invariant. Since the loop only involves the simultaneous assignment

```
a = b
b = a `rem` b
```

we need to show that the greatest common factor of a and b is the same before and after this assignment is carried out. We can write

$$a = qb + r$$

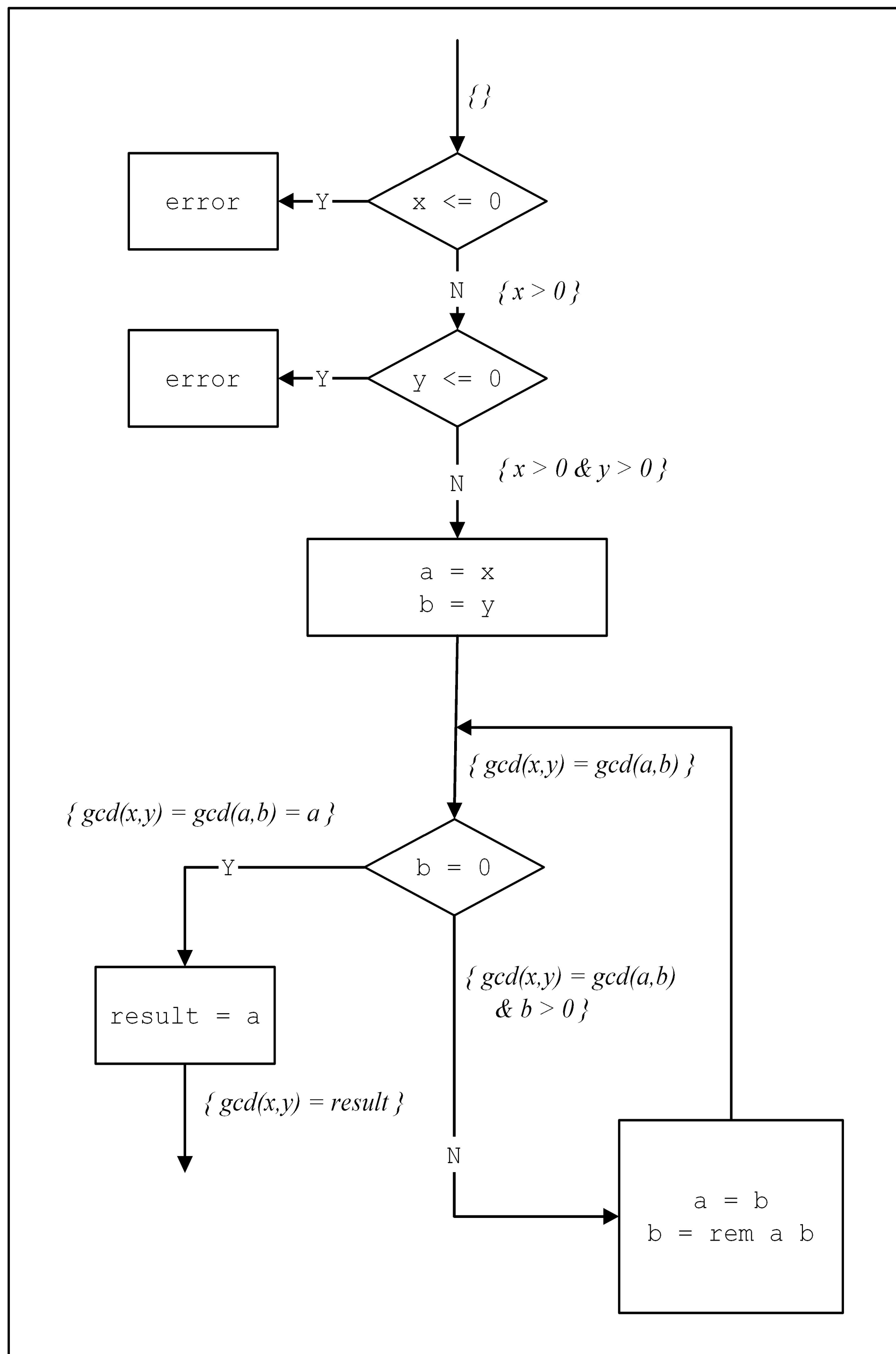


Figure 3.7: Complete annotated flow-diagram for `gcd`

where q is some non-negative integer, and r (equal to `a ‘rem‘ b`) satisfies $0 \leqslant r < b$. Let's write $g = gcd(a, b)$ prior to the simulataneous assignment. By the time it terminates, we have

$a = b$;
$b = r$;

so we need to prove that $gcd(a, b) = gcd(b, r)$.

Writing g for $gcd(a, b)$, we note that (by definition) g divides exactly into both a and b.

- Since we can write $r = a - qb$, g also divides exactly into r.
- So g is definitely a common factor of both b and r.

To see that it's the *greatest* common factor, suppose that g' is a larger common factor.

- Since g' divides both b and r, it also divides $a = qb + r$.
- So g' is also a common factor for a and b.
- But this contradicts the definition of g, which is supposed to be the *greatest* common factor of a and b, because $g' > g$.
- This contradiction completes the argument.

It follows that $gcd(a, b) = gcd(b, r)$, as claimed.

Proof of termination

As we saw in the last section, each time we go round the loop, b is replaced by a value r satisfying $0 \leqslant r < b$ (this actually relies on the fact that $b > 0$ when we enter the loop, which is why I've included it in the diagram. So b always decreases by a non-zero amount as we go round the loop. By definition, b is positive when we start, and can't go non-negative, so eventually the loop has to exit.

Proof of correctness

When the loop terminates, the loop invariant is still true, but we also know that $b = 0$. Since $gcd(a, 0) = a$, it follows from the loop invariant, $gcd(x, y) = gcd(a, b)$ that $gcd(x, y) = a$. So setting `result` equal to a generates the required result.

3.4 Proof by induction

As our proof of Haskell's `gcd` function shows, it is entirely possible to use traditional program-proof techniques to demonstrate the correctness of Haskell programs. It's more usual, however, to use various forms of ***proof by induction***. You're already familiar with induction from your earlier courses (for example, dicrete mathematics), but we'll see as the course goes on that there are *many* different versions of induction, each one tailored to the situation at hand.

The standard *principle of induction for non-negative integers* says that whenever we want to prove that a predicate $P(n)$ holds true for every $n \in \mathbb{N}$, it's enough to prove the *base case* and *induction step* for P.

- **Base case for** P.
  We have to prove that $P(0)$ holds outright.
- **Induction step for** P.
  We have to prove that $P(n) \Rightarrow P(n + 1)$ holds for every $n \in \mathbb{N}$.

For example, consider this question from an assignment a few years ago.

QUESTION

Consider the following algebraic datatype:

```
data Chain a = End | Link a (Chain a)
```

and suppose `joinChains`, `chainToList` and `chainLength` are defined by

```
joinChains :: Chain a -> Chain a -> Chain a
joinChains End c = c                              -- [join.1]
joinChains (Link a c) d = Link a (joinChains c d) -- [join.2]

chainToList :: Chain a -> [a]
chainToList End = []                              -- [tolist.1]
chainToList (Link x c) = x : (chainToList c)      -- [tolist.2]

chainLength :: Chain a -> Int
chainLength End = 0                               -- [length.1]
chainLength (Link x c) = 1 + (chainLength c)      -- [length.2]
```

- Show by induction that

  ```
  chainLength c == length (chainToList c)
  ```

  for all finite chains `c` of type `Chain a`.
- Show by induction that

  ```
  chainLength(joinChains c d) == (chainLength c) + (chainLength d)
  ```

  for all finite chains `c` and `d` of type `Chain a`.

END OF QUESTION

In the assignment, you were asked to invent a form of induction specifically tuned to the `Chain` data type, but we can also answer the question using standard induction. For example, here's an answer to part (a).

We have to prove that

```
chainLength c == length(chainToList c)
```

for all finite chains `c` of type `Chain a`. To use standard induction, we need to re-express what we have to prove as something to do with natural numbers, and the usual technique is to tie-in something like `length`, that links objects to natural numbers. With this in mind, we'll prove that

$P(n) \equiv$

```
chainLength c == n
⇒
chainLength c == length (chainToList c)
```

holds for all $n \in \mathbb{N}$. Clearly, if the result holds for all possible finite chain lengths, then it also holds for all possible finite chains, so proving that $P(n)$ is always true is enough to prove the required result.

Base Case. We have to prove $P(0)$, *i.e.*,

```
chainLength c == 0
⇒
chainLength c == length (chainToList c)
```

If we look at the definition of `chainLength`, it's clear that we can only get `chainLength c == 0` if `c == End`. But now, we have

```
length (chainToList c)  == length (chainToList End)
                        == length []
                        == 0
```

So in this case we have

```
chainLength c == 0 == length (chainToList c)
```

Induction Step. We have to prove $P(n) \Rightarrow P(n+1)$, *i.e.*,

If

```
chainLength c == n
⇒ chainLength c == length (chainToList c)
```

Then

```
chainLength c == (n+1)
⇒ chainLength c == length (chainToList c)
```

So, let's suppose c is a chain for which `chainLength c == (n+1)`, and let's suppose that $P(n)$ holds (this is called the *induction hypothesis*. Looking at the definition of `chainLength`, the only way we can get this result is if c is of the form `Link x c'` for some `x :: a` and `c' :: Chain a`. The evaluation of `chainLength` gives us

```
chainLength c  == chainLength (Link x c')
               == 1 + chainLength c'
```

Since `chainLength c` is $n+1$, it follows that `chainLength c'` must be n. The induction hypothesis, $P(n)$ can therefore be applied, and we can state with confidence that

```
length (chainToList c') == chainLength c' == n
```

3.4.1 Putting it all together

Here's the complete derivation of $P(n+1)$ — remember, the length of the chain c is assumed to be $(n+1)$. The lines marked `***` show where the induction hypothesis $[P(n) \Rightarrow P(n+1)]$ is used. We can use it because we know that `c'` is a chain of length n, so that the assumption $P(n)$ can be applied.

```
length (chainToList c)
== length (chainToList (Link x c'))
== length (x : chainToList c')
== 1 + length (chainToList c')       ***
== 1 + chainLength c'                ***
== chainLength (Link x c')
== chainLength c
```

This completes the proof of part (a), and a similar approach can be used for part (b). You are invited to construct this proof yourself!

Chapter 4

ADTs and their applications

4.1 Introduction

This week we'll be looking at two techniques that are widely used when proving things about Haskell programs. We saw earlier in the course that imperative programs can be proved correct using *Floyd-Hoare* techniques, and that these techniques can also be adapted to work for Haskell programs. This week we'll be looking at two techniques that are especially useful when dealing with functional programs: *equational reasoning* and *structural induction*.

4.2 Equational Reasoning

One way to define the `length` function on lists is like this:

```
length :: [a] -> Int
length []     = 0               -- [length.1]
length (x:xs) = 1 + length xs   -- [length.2]
```

Suppose you're asked to show that `length [1]` evaluates to `1`. You might write down something like this:

```
length [1] ⇝  1 + length []  by [length.2]
           ⇝  1 + 0          by [length.1]
           ⇝  1              properties of (+)
```

Similarly, if someone asked you to prove that `length ["hello"]` evaluates to `1`, you might write this:

```
length ["hello"] ⇝  1 + length []  by [length.2]
                 ⇝  1 + 0          by [length.1]
                 ⇝  1              properties of (+)
```

So far, so good. But now suppose someone asked you to prove that `length [x]` evaluates to `1`, *no matter what value* `x` *has, and no matter what type it has.*

You might be tempted to write the following, but *this is not valid*:

```
This is not valid
length [x] ⇝  1 + length []  by [length.2]
           ⇝  1 + 0          by [length.1]
           ⇝  1              properties of (+)
```

Why not?

The reason becomes clear if you try asking HUGS to evaluate '`length [x]`'. Haskell can only evaluate this expression if `x` has been defined somewhere; you can't leave `x` completely undefined and expect Haskell to do the evaluation for you. Even so, it's quite clear that `length [x]` *does* evaluate to `1`, no matter what value or type `x` takes. Although we can't prove this directly by evaluating the expression, we can use ***equational reasoning*** to do so instead.

The evaluations above implicitly assume something about the rules `[length.1]` and `[length.2]`— they assume that the *purpose* of these rules is to tell us how to evaluate expressions that involve the `length` function. But we can also look at them in another, completely different, way. Consider `[length.2]`, for example:

```
length (x:xs) = 1 + length xs  -- [length.2]
```

Instead of saying that an expression of the form `length (x:xs)` should be *replaced* during evaluation by the corresponding expression `1 + length xs`, we can instead think of the rule as telling us that these two expressions *always evaluate to the same result.*

EVALUATION-BASED MEANING	REASONING-BASED MEANING
Replace $expr_1$ with $expr_2$	$expr_1$ and $expr_2$ evaluate to the same result

Table 4.1: TWO WAYS TO INTERPRET THE STATEMENT $expr_1 = expr_2$

You've already done this sort of reasoning many times; we write '$expr_1 == expr_2$' to mean that $expr_1$ and $expr_2$ have the same value (even if we don't know what it is). Given this notion, we can now prove that `length [x]` always evaluates to `1`; we first show that (whatever the answer is) it has to be the same as the answer you get by evaluating '`1 + 0`'. Since we can evaluate this last expression, we can deduce what answer we get when we evaluate `length [x]`:

```
length [x] ==  1 + length []  by [length.2]
           ==  1 + 0          by [length.1]
           ⇝   1              properties of (+)
```

Because this type of proof relies on interpreting program statements as equations (rather than re-write rules), we call it *equational reasoning.* We've already seen one advantage of equational reasoning – it allows you deduce things about *every possible* value x, rather than having to work things out one case at a time. It also allows us to write rules *backwards*, something we often have to do during induction proofs.

Suppose `int2list` and `int2string` are the functions

```
int2list n
  | n <= 0    = []                      -- [int2list.1]
  | otherwise = [1..n]                  -- [int2list.2]

int2string n
  | n <= 0    = ""                      -- [int2string.1]
  | otherwise = '*' : int2string (n-1)  -- [int2string.2]
```

Clearly, `length (int2list n)` and `length (int2string n)` both give the same result. But how can we prove it?

The obvious answer is to use induction, but this only works for the non-negative integers. What about the negative integers? We use equational reasoning instead. If `n < 0` we have

```
length (int2list n) ==  length []                by [int2list.1]
                    ==  0                        list properties
                    ==  length ""                string properties
                    ==  length (int2string n)    by [int2string.1] backwards
```

Notice that we've used `[int2string.1]` *backwards*. We're implicitly treating the rule as an equation, not a rewrite rule.

4.3 Two Induction Principles for Natural Numbers

In both mathematics and computer science we are often asked to prove statements of the form $(\forall\, n \in \mathbb{N})P(n)$. The basic *principle of induction* with which you are all familiar tells us how to prove such a statement:

Principle of induction
To prove that the statement $(\forall\, n \in \mathbb{N})P(n)$, it is enough to

- Prove that $P(0)$ is valid;
- Prove that $P(n+1)$ is valid whenever $P(n)$ is valid.

Sometimes, however, this principle can't easily be applied without a lot of extra manipulation. For example, consider the well-known function

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

Suppose someone asks you to prove something incredibly basic about this function, like '`fib n` has a definite value, for every `n` $\geqslant 0$'. This is the statement $(\forall\, n \in \mathbb{N})P(n)$, where

$$P(n) = \text{`}\texttt{fib n}\text{ has a definite value'}$$

Proving this by induction requires us to prove that

- `fib 0` has a definite value;
- `fib (n+1)` has a definite value whenever `fib n` has a definite value.

The problem is, we can't do the second part of this proof! Since `fib (n+1) == fib n + fib (n-1)`, the only way we can prove that `fib (n+1)` is defined is by assuming that *both* `fib n` and `fib (n-1)` have definite values; that is, we have to assume that both $P(n)$ *and* $P(n-1)$ are valid; but we're not allowed to – all we can assume is $P(n)$. In such circumstances we can use the *generalised principle of induction* instead.

Generalised principle of induction
To prove that the statement $(\forall\, n \in \mathbb{N})P(n)$ is valid for every $n \in \mathbb{N}$, it is enough to

- Prove that $P(0)$ is valid.
- Prove that $P(n+1)$ is valid whenever ($P(k)$ is valid, for every $k \leqslant n$).

This time, when we come to prove that `fib (n+1)` is defined, we are explicitly allowed to assume both $P(n)$ and $P(n-1)$.

4.3.1 The relationship between induction and structure

Haskell doesn't have a type '`Nat`' representing natural numbers; instead we have to use types like `Int`. To avoid this problem we might decide to define our own ADT representing natural numbers. The key points to bear in mind are that the ADT of natural numbers requires a *zero* element, and a mechanism for finding the *successor* of any previously generated natural number. That is, we want[1]

- TYPES
  Nat
- SYNTAX
  $zero : Nat$
  $succ : Nat \to Nat$
- SEMANTICS
  $x = succ(y) \Rightarrow x \neq zero$
  $succ(x) = succ(y) \Rightarrow x = y$

This says that a natural number is either *zero*, or else it is the successor of another (uniquely identifiable) natural number.

Here's one implementation:

```
data Nat1 = Zero | Succ Nat1

zero :: Nat1
zero = Zero

succ :: Nat1 -> Nat1
succ n = Succ n
```

This uses the custom-built algebraic data type, `Nat1`, to represent natural numbers. Alternatively, here's how a mathematician might define natural numbers, using another custom-built algebraic data type, `Nat2`.

```
data Nat2 = BiggerThan [Nat2]

zero :: Nat2
zero = BiggerThan []

succ :: Nat2 -> Nat2
succ n@(BiggerThan ns) = BiggerThan (ns ++ [n])
```

[1]The full semantics are somewhat more complicated than shown here.

This really *is* how (some) mathematicians define the natural numbers. They say that every natural number n is just the set of its predecessors (the numbers it is *bigger than*), so that 0 is the empty set, while $1 = \{0\}$, $2 = \{0, 1\}, 3 = \{0, 1, 2\}$ and so on. Both implementations are valid, but they suggest very different structures when it comes to proving things.

Induction and structure

The way we've defined `Nat1` has something in common with the way standard induction works. An object of type `Nat1` is either `Zero`, or else is of the form `'n+1' = Succ n`. By analogy, proof by induction requires to prove $P(0)$ and $P(n+1) \Leftarrow P(n)$. In the same way, `Nat2` has something in common with generalised induction. Notice that we can define an enumeration on `Nat2` so that each `n :: Nat2` satisfies

```
n == BiggerThan [zero .. 'n-1']
```

where `'n-1'` denotes the predecessor of `n`. In other words, each value in `Nat2` is constructed using information about all of its predecessors. Likewise, generalised induction tells us to *assume* we have information about n's predecessors, and use it deduce something about n.

Given `n ::`	`Nat1, n` is	STANDARD PRINCIPLE OF INDUCTION
either	`Zero`	Prove $P(0)$
or	`Succ m`	Prove $P(succ(m)) \Leftarrow P(m)$
Given `n ::`	`Nat2, n` is	GENERALISED PRINCIPLE OF INDUCTION
either	`BiggerThan []`	Prove $P(0)$
or	`BiggerThan [zero .. m]`	Prove $P(succ(m)) \Leftarrow (\forall\, k : 0 \leqslant k \leqslant m)P(k)$

(Note. m denotes the natural number '$n - 1$')

Table 4.2: A RELATIONSHIP BETWEEN STRUCTURE AND INDUCTION

In each case, the principle of *structural induction* is at work, as we now explain.

4.4 Structural Induction

The general structure of an algebraic data type definition is

```
data MyType TypeList   = C₁ TypeList₁
                       | C₂ TypeList₂
                       | ...
                       | Cₙ TypeListₙ
```

where each *constructor* $\texttt{C}_k$ is followed by a (possibly empty) list of types. If we want to prove that $P(x)$ is a valid statement for every x of type `MyType`, how might we do it?

For the time being, focus on values that are *finite and defined*. This means that x can be constructed using at most finitely many applications of the constructors $\texttt{C}_1$, ..., $\texttt{C}_n$.

If we want to prove that $P(x)$ is valid, it's enough to prove that

- $P(\texttt{C}_k\ \texttt{arglist}_k)$ holds for every k, and every finite and defined $\texttt{arglist}_k$ entry which is itself constructed using the `MyType` definition.
- If $\texttt{TypeList}_k$ is empty, then we simply have to prove $P(\texttt{C}_k)$ outright.

We call this the ***principle of structural induction***.

4.4.1 Example: `Nat1`

In the definition of `Nat1`, we have two constructors:

- `Zero`
- `Succ Nat1`

So structural induction tells us that if we want to prove that P(`n`) is valid for every finite defined `n::Nat1`, it's enough to

- Prove P(`Zero`) outright;
- Prove that P(`Succ n`) holds whenever P(`n`) holds.

Clearly, the standard principle of induction is just a special case of structural induction.

4.4.2 Example: `Nat2`

The definition of `Nat2` involves just one constructor:

- `BiggerThan [Nat2]`

Structural induction tells us that if we want to prove P(`n`) for every finite defined `n::Nat2`, we have to show that

- P(`BiggerThan [zero .. n]`) is valid whenever we know that P(`k`) is valid for all `k` in `[zero .. n]`.

So the generalised principle of induction is also a special case of structural induction.

4.5 Structural Induction and Lists

What does structural induction tell us about lists? Lists are built using just two constructors, which we can represent as:

- `[ ]`, the empty list;
- `(:) :: a->[a]->[a]`, the *cons* function.

According to structural induction, if we want to prove that a statement P is true for every finite defined list, all we need to do is

- Prove P(`[]`) holds outright;
- Prove P(`(x:xs)`) holds whenever P(`xs`) holds.

4.5.1 Example: `length` and `reverse`

Let's prove that

```
length (reverse xs) == length xs
```

for all lists `xs`. We'll use these definitions to keep things simple:

```
length []      = 0                    -- [len.1]
length (_:xs)  = 1 + length xs        -- [len.2]

reverse []     = 0                    -- [rev.1]
reverse (x:xs) = reverse xs ++ [x]    -- [rev.2]
```

Part 1: Prove P(`[]`) holds outright
We have

```
length (reverse []) ==  length []  by [rev.1]
```

as required.

Part 2: Prove P(`(x:xs)`) holds whenever P(`xs`) holds
We are allowed to assume that

```
length (reverse xs) == length xs   -- [assumed]
```

and have to prove that

```
length (reverse (x:xs)) == length (x:xs)
```

The full proof requires an extra lemma (saying that `length (xs ++ [x]) == 1 + length xs`) but in essence it goes like this:

```
length (reverse (x:xs)) ==  length (reverse xs ++ [x])  by [rev.2]
                        ==  1 + length (reverse xs)     [needs lemma, not shown]
                        ==  1 + length xs               by [assumed]
                        ==  length (x:xs)               by [len.2]
```

4.5.2 Why do values have to be finite and defined?

I've been careful throughout this discussion of structural induction to stress the need for values to be finite and defined. What happens if these conditions aren't satisfied?

Here's an example of an undefined value; it's called `loop` because this is what Haskell will do if you try to evaluate it.

```
loop :: a
loop = loop        -- [loop]
```

We can use `loop` to build a *partially defined* list:

```
partial = 1 : loop -- [partial]
```

What happens if we try reversing `partial`?

```
reverse partial ⇝  reverse (1 :  loop)   by [partial]
                ⇝  reverse loop ++ [1]   by [rev.2]
                ⇝  reverse loop ++ [1]   by [loop]
                ⇝  reverse loop ++ [1]   by [loop]
                ⇝  reverse loop ++ [1]   by [loop]
                ⇝  ...
```

The evaluation never terminates!

Clearly, neither `partial` nor `reverse partial` are properly defined. But are they equal? To demonstrate that they are *not* equal, we need to show that they behave differently, and we can do this by asking what happens when we apply the function `head` to each of them in turn.

We have

```
head :: [a] -> a
head []    = error "head: empty" -- [head.1]
head (x:_) = x                   -- [head.2]
```

and so

```
head (reverse partial) ⇝  head (reverse loop ++ [1])  shown above
                       ⇝  ...                         loops forever

          head partial ⇝  head (1 :  loop)            by [partial]
                       ⇝  1                           by [head.2]
```

Even though neither of them is properly defined, the expressions `partial` and `reverse partial` are definitely different! They behave differently when acted upon by `head`.

4.6 Structural induction on trees

As a final example, let's consider structural induction on trees. Although we *can* use standard induction, structural induction is often much easier. We'll use the following definition of the (binary) `Tree a` data type:

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

This has two constructors

- `Empty`
- `Node (Tree a) a (Tree a)`

so the principle of structural induction tells us that if we want to prove that a property P holds for every finite defined tree `t`, it's enough to prove that

- P(`Empty`) holds outright;
- P(`Node l x r`) holds whenever P(`l`) and P(`r`) *both* hold.

For example, suppose we define

```
flatten :: Tree a -> List a
flatten Empty = []                                  -- [flat.1]
flatten (Node l x r) = flatten l ++ [x] ++ flatten r  -- [flat.2]

treesum :: Tree Int -> Int
treesum Empty = 0                                   -- [tsum.1]
treesum (Node l x r) = treesum l + x + treesum r      -- [tsum.2]

sum :: [Int] -> Int
sum [] = 0                                          -- [sum.1]
sum (x:xs) = x + sum xs                             -- [sum.2]
```

How can we prove that

```
sum (flatten t) == treesum t
```

is a valid equation for all finite defined integer trees, `t`?

Answer: Use structural induction!

Part 1: Prove P(`Empty`) holds outright

```
sum (flatten Empty) ==  sum []          by [flat.1]
                    ==  0               by [sum.1]
                    ==  treesum Empty   by [tsum.1]
```

as required.

Part 2: Prove P(`Node l x r`) holds if P(`l`) and P(`r`) *both* hold
As so often happens, we need to prove some auxiliary results. In this case, we need to prove that, for any integer lists `xs, ys, zs` we have

```
sum (xs ++ ys ++ zs) = sum xs + sum ys + sum zs -- [lemma]
```

Taking this for granted, we have

```
sum (flatten (Node l x r)) ==  sum (flatten l ++ [x] ++ flatten r)        by [flat.2]
                           ==  sum (flatten l) + sum [x] + sum (flatten r)  by [lemma]
                           ==  treesum l + sum [x] + sum (flatten r)      by [P(l)]
                           ==  treesum l + sum [x] + treesum r            by [P(r)]
                           ==  treesum l + (x + sum []) + treesum r       by [sum.2]
                           ==  treesum l + (x + 0) + treesum r            by [sum.1]
                           ==  treesum l + x + treesum r                  by [arithmetic]
                           ==  treesum (Node l x r)                       by [tsum.2]
```

as required.

4.7 Summary

This week we have considered *equational reasoning* and *structural induction.* We've seen in a variety of proofs that the two principles are typically used together, with equational reasoning

being used during induction proofs to show that something is true for *all* instances of a given variable.

We noted that there are two distinct principles of induction for natural numbers. The standard version, with which you're already very familiar, says that to prove $P(n + 1)$, you can only assume $P(n)$. The generalised version is much more powerful; it lets you assume that all of the statements $P(0), P(1), \ldots, P(n)$ are valid simultaneously. Ultimately, however, both of these induction principles are simply special cases of *structural induction*, corresponding to two different ways in which the natural numbers themselves might be defined in a Haskell-type language.

Chapter 5

Parsing Types and Expressions

5.1 Introduction

- Why does throwing extra people at a project that is running late typically make it even later?
- What do these mean?
  - heapsort is **faster** than insertion sort.
  - quicksort is the **fastest** sorting algorithm.
  - there is no known **feasible** travelling salesman algorithm.
  - that problem is **intractable**.
  - the amount of work involved **grew exponentially**.
  - his testing strategy failed because he'd forgotten about the **combinatorial explosion** that occurs when you scale systems up.

An algorithm can be expressed

- in many different languages; and
- in many different *styles* of language

If a JAVA program always outperforms the PROLOG one, is that because it's using a better algorithm, or is it due to differences between the two languages? To avoid this problem, we'll assume that all programs are written in the same language — Haskell.

Our focus this week is on the measurement of ***program complexity***. The first thing to notice is that complexity can be measured in many different ways. In particular, we can measure

- the *time* taken to execute the program;
- the *resources* (typically memory, or *space*) used by the program during computations.

Both measures depend on the input supplied to the program (*e.g.*, searching a large database typically takes longer than searching a small one), so we need to take account of inputs when computing program complexity.

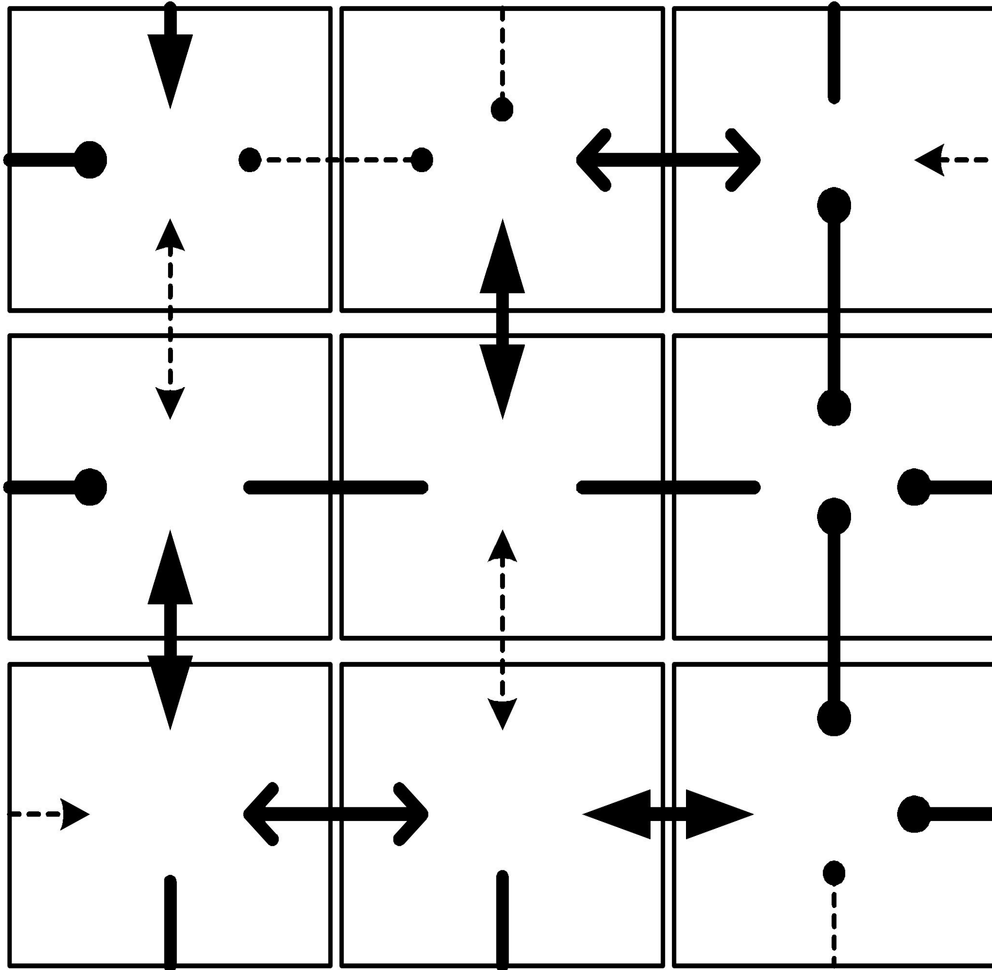


Figure 5.1: A 3×3 monkey puzzle (for illustration only)

n	Number of Arrangements (showing calculation)				
1	4	1!	$\times$	4^1	$= 1 \times 4$
2	6144	4!	$\times$	4^4	$= 24 \times 256$
3	95126814720	9!	$\times$	4^9	$= 362880 \times 262144$
4	89862698310039502848000	16!	$\times$	4^{16}	$= 20922789888000 \times 4294967296$

Table 5.1: The number of arrangements of an $n \times n$ monkey puzzle

Example: Two implementations that are easy to compare

Consider the following two definitions of the identity function:

```
id1 :: Int -> Int
id1 n = n                      -- [id1]

id2 :: Int -> Int
```

```
id2 n
  | n <= 0    = 0                -- [id2.1]
  | otherwise = 1 + id2 (n-1)    -- [id2.2]
```

No matter what value of n we choose, the evaluation of `id1 n` always takes just one step:

```
id1 n  ⇝ n  by [id1]
```

but the number of steps taken by `id2` depends on the value of `n`:

```
id2 1  ⇝ 1 + id2 0       by [id2.2]  3 steps
       ⇝ 1 + 0           by [id2.1]
       ⇝ 1               arithmetic

id2 2  ⇝ 1 + id2 1       by [id2.2]  5 steps
       ⇝ 1 + (1+ id2 0)  by [id2.2]
       ⇝ 1 + (1 + 0)     by [id2.1]
       ⇝ 1 + 1           arithmetic
       ⇝ 2               arithmetic
```

Clearly, `id2` is ‘less efficient’ than `id1` – it has ‘higher complexity’.

Given a function `f::Int->Int`, let `(stepcount f)::Int->Int` be the function which tells us how many steps `f` takes when given a specific input.

```
stepcount id2 0 ⇝ 1
stepcount id2 1 ⇝ 2
stepcount id2 2 ⇝ 3
```

In general

```
stepcount id2 = (\n -> n+1)
```

We say `id2` has *linear* time complexity.

```
stepcount id1 = (\n -> 1)
```

We say `id1` has *constant* time complexity.

Two implementations that are harder to compare

```
delay :: Int -> Int
delay n
  | n <= 1    = 0
  | otherwise = delay (n-1)

id3 :: Int -> Int
id3 n
  | isEven n  = (delay 1000) + n
```

```
    | otherwise = id2 n

id4 :: Int -> Int
id4 n
  | isEven n  = id2 n
  | otherwise = (delay 1000) + n
```

```
stepcount id3 = (\n -> if (isEven n) then   1001   else   (n+1))
stepcount id4 = (\n -> if (isEven n) then  (n+1)   else   1001)
```

What do these examples tell us?

- even very simple problems can take a lot of work to solve $\Longrightarrow$ we need to consider the rate at which workload increases.
- even very simple specifications can have more than one implementation, with different complexities.
- the relationship between solutions to the same problem need not be simple

5.2 The growth of functions

The amount of work required to solve a problem depends on its 'size'; for example,

$$T_{\text{monkey}-\text{puzzle}}(n) = (n^2)! \times 4^{(n^2)}$$

This is too complicated. We need to simplify functions, focussing only on those properties that are directly relevant to program complexity.

> Suppose $n \mapsto f(n)$ and $n \mapsto g(n)$ are functions of n. How can we compare their respective rates of growth?

The mathematical answer is called ***big-O*** notation.

5.2.1 Example: Growth of polynomials

Consider the following polynomials:

$$f(n) = n^3 + n^2 + 10n + 1000000$$
$$g(n) = 10n^2 + 2n + 1000000$$

Which grows faster? Try some evaluations:

Small values give misleading results! We need *asymptotic complexity* (behaviour for very large values):

For *very* large values of n, the only term in f that actually matters is the n^3 term, and the only term in g that matters is the $10n^2$.

The values f takes for very large values of n becomes increasingly indistinguishable from that of n^3 — we say that f is of ***order*** n^3, and write $f = O(n^3)$.

n	$f(n)$	$g(n)$
0	1,000,000	1000000
1	1,000,012	1000012
2	1,000,032	1000044
3	1,000,066	1000096

Table 5.2: For small values of n, $g(n)$ appears to grow faster than $f(n)$.

n	$f(n)$	$g(n)$
10^3	10^9	10^7
10^4	10^{12}	10^9
10^5	10^{15}	10^{11}

Table 5.3: Approximate values of f and g for larger values of n.

n	$f(n) \approx n^3$	$g(n) \approx 10n^2$
10^{10}	10^{30}	10^{21}
10^{20}	10^{60}	10^{41}
10^{30}	10^{90}	10^{61}
10^{40}	10^{120}	10^{81}

Table 5.4: Approximate values of f and g for even larger values of n.

Likewise, g is of order n^2 and we write $g = O(n^2)$.

5.2.2 Big-O notation

Given two functions $f, F \colon \mathbb{N} \to \mathbb{N}$ such that $F(n) \geqslant 0$ for all n, we write

$f = O(F)$ pronounced: f is big oh of F

if there exists constants $M, N > 0$ such that $|f(n)| \leqslant MF(n)$ for all $n > N$.

The function F in this definition can easily be an over-estimate. If we consider the function f defined above, we have

- $f = O(n^3)$
- $f = O(n^4)$
- $f = O(n^5)$
- $f = O(2^n)$
- $f = O(n!)$, and so on.

The ***big-O notation*** implicitly regards the expression e in $O(e)$ as a *function*; in effect, we are using ***lambda notation***.

Standard notation	λ-notation	Haskell notation
$f(x) = x$	$f = \lambda\, x.x$	`f = (\x -> x)`
$f(x) = x + 1$	$f = \lambda\, x.x + 1$	`f = (\x -> x+1)`
$f(x) = x^2$	$f = \lambda\, x.x^2$	`f = (\x -> x*x)`

Table 5.5: Examples of lambda notation.

5.2.3 Comparing functions

$f = O(g)$ means that g ultimately grows *at least as quickly* as f, and may grow much *more* quickly.

On the other hand, two functions might well be of the *same* order, e.g. if $f(n) = n$ and $g(n) = n+1$, then $f = O(g)$ and $g = O(f)$ are both valid.

We can use these concepts to define a basic ordering on functions.

We will write $f \ll g$ to mean that $f = O(g)$ but $g \neq O(f)$ — in other words, g definitely outgrows f in the long run.

The ordering of functions has been studied extensively. We have:

$$n^0 \ll n^1 \ll n^2 \ll \ldots \ll n^k \ll \ldots \ll 2^n \ll 3^n \ll \ldots \ll n! \ll \ldots$$

and there are many functions between each pair of functions in this list. For example,

$$n^0 \ll \log\log n \ll \log n \ll n^1 \ll n \log n \ll \ldots$$

5.3 Counting the number of steps executed

How do measure the amount of work involved in executing a program? The answer depends on several factors. In particular, we need to know whether the language in question is ***lazy*** or ***eager***.

- eager languages do far more work than lazy ones; but
- it is easier to work out how many steps they need to execute

To keep things simple, we will consider *eager* evaluation; the results we obtain will therefore be *upper-limits* to the true workload in Haskell-like languages.

Strict time complexity

- For each function f defined in the program, we derive a related ***step-counting function*** T_f. Given any input x, the value $T_f(x)$ gives the number of steps required to evaluate $f(x)$.
- Next, we define the *size* of an input x. This will allow us to re-express T_f in terms, not of x itself, but of its size.
- The formula we get for T_f is likely to involve recursion. We put it into *closed-form*, so we can use big-O notation.

Other measures of complexity

- The ***worst-case complexity*** is the complexity computed when we just happen to choose inputs of each size which require the greatest workload.

- The ***best-case complexity*** is the complexity computed when we just happen to choose the easiest inputs of each size.
- The ***average-case complexity*** is the average complexity, where the average is taken over the various inputs of each size.
- Others exist as well, e.g. you may find *amortised complexity* an interesting concept!

If the complexity is *uniform*, *i.e.*, the same for all inputs of the same size, then the worst-, best- and average-case complexities are all equal. For example, if we *reverse* lists instead of sorting them, then the amount of work involved depends only on the length of the list, and not on its elements.

5.3.1 The step-counting function T_f and the cost-function T

For any f defined in a Haskell program, its step-counting function is called T_f. In order to compute T_f we need to know how much it costs to evaluate the various types of expression we encounter during execution of the program. We write $T(e)$ for the cost of evaluating the expression e.

cost:case if $f\ a_1\ a_2 \ldots a_n$ is defined directly by an expression of the form $f\ a_1\ a_2 \ldots a_n = e$ then evaluating f requires us first to perform pattern matching to identify the relevant expression as e (assume this takes one step), and then to evaluate it (which takes $T(e)$ steps). So define

$$T_f\ a_1\ a_2 \ldots a_n = 1 + T(e)$$

cost:const if c is a constant, it costs nothing to evaluate c, *i.e.*,

$$T(c) = 0$$

cost:var if v is a variable, it costs nothing to look up the value of v, *i.e.*,

$$T(v) = 0$$

cost:if if e is an expression of the form `if` a `then` b `else` c, then the cost depends on the value of a. In either case we have to evaluate a first (this requires $T(a)$ steps); if a is True, we need to evaluate b (using $T(b)$ steps) and otherwise c (using $T(c)$ steps). So we define

$$T(\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c) = T(a) + (\texttt{if}\ a\ \texttt{then}\ T(b)\ \texttt{else}\ T(c))$$

cost:prim If p is a primitive operation (like addition) which is implemented as part of the language, we can assume that the implementation is efficient enough that the cost of calling p can be ignored. Consequently, the cost of evaluating $p\ a_1 \ldots a_n$ is just the cost of evaluating the n different arguments. So

$$pa_1 \ldots a_n = T(a_1) + \cdots + T(a_n)$$

cost:func If f isn't a primitive function, and the particular evaluation $f\ a_1\ a_2 \ldots a_n$ isn't one of the cases used to define f, the cost of evaluating $f\ a_1 \ldots a_n$ is the cost of evaluating the various arguments, *together with* the cost of applying f to those results, *i.e.*,

$$T(f\ a_1 \ldots a_n) = T(a_1) + \cdots + T(a_n) + (T_f\ a_1 \ldots a_n)$$

5.4 Two Examples

It is probably helpful at this stage to see some examples of step counting in action.

5.4.1 The function `sum`

What is the step-counting function T_{sum} associated with the function `sum`, defined by:

```
sum [] = 0                 -- [sum.1]
sum (x:xs) = x + sum xs    -- [sum.2]
```

Our task is to define a function T_{sum} that takes the same inputs as `sum`, and tells us the number of function calls executed by `sum` when given those same arguments.

Since `sum` is defined in two cases by giving pattern-matching, T_{sum} will be defined the same way.

Case 1: T_{sum} `[]`
We have to determine the value of T_{sum} `[]`. Definition `[sum.1]` tells us that `sum []` is one of the specific cases used to define `sum`. According to `[cost:case]`, we have

$$T_{sum}\,\texttt{[]} = 1 + T(\texttt{0})$$

According to `[cost:const]`, evaluating the constant value `0` has no cost whatsoever. Putting the pieces together gives us:

$$
\begin{aligned}
T_{sum}\ \texttt{[]} &= \texttt{1 + } T(\texttt{0}) && \texttt{by [cost:case]} \\
&= \texttt{1 + 0} && \texttt{by [cost:const]} \\
&= \texttt{1} && \texttt{by arithmetic}
\end{aligned}
$$

Case 2: T_{sum} `(x:xs)`
Following the same reasoning as before, we find that

$$
\begin{aligned}
T_{sum}\ \texttt{(x:xs)} &= \texttt{1 + } T(\texttt{x + sum xs}) && \texttt{by [cost:case]} \\
&= \texttt{1 + } T(\texttt{(+) x (sum xs)}) && \texttt{rewriting} \\
&= \texttt{1 + } T(\texttt{x}) \texttt{ + } T(\texttt{sum xs}) && \texttt{by [cost:prim] ((+) is primitive)} \\
&= \texttt{1 + 0 + } T(\texttt{sum xs}) && \texttt{by [cost:var]} \\
&= \texttt{1 + 0 + } T_{sum}(\texttt{xs}) && \texttt{by [cost:func]} \\
&= \texttt{1 + } T_{sum}(\texttt{xs}) && \texttt{by arithmetic}
\end{aligned}
$$

Consequently, the step-counting function T_{sum} is given by

$$
\begin{aligned}
&T_{sum}\ \texttt{[] = 0} \\
&T_{sum}\ \texttt{(x:xs) = 1 + } T_{sum}\ \texttt{xs}
\end{aligned}
$$

and it's easy to see that T_{sum} is just the `length` function by another name. This makes sense: the number of terms that need to be added together is just the length of the argument list.

5.4.2 The concatenation function

Define

```
 [] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Then

```
T++ [] ys     = 1 + T(ys)                          by [cost:case]
              = 1 + 0                              by [cost:var]
              = 1                                  by arithmetic

T++ (x:xs) ys = 1 + T(x:(xs++ys))                  by [cost:case]
              = 1 + T(x) + T(xs++ys)               by [cost:prim]
              = 1 + 0 + T(xs++ys )                 by [cost:var]
              = 1 + 0 + T(xs) + T(ys) + T++ xs ys  by [cost:func]
              = 1 + 0 + 0 + 0 + T++ xs ys          by [cost:var]
              = 1 + T++ xs ys                      by arithmetic
```

Clearly, the value of `ys` is irrelevant in this calculation, and we have

```
T++ xs ys = 1 + length xs
```

5.4.3 The reverse function

Define `rev` by

```
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

Then

```
Trev []
= 1 + T([])                                by [cost:case]
= 1 + 0                                    by [cost:const]
= 1                                        by arithmetic

Trev (x:xs)
= 1 + T(rev xs ++ [x])                     by [cost:case]
= 1 + T(rev xs) + T([x]) + T++ (rev xs) [x]  by [cost:case]
= 1 + Trev xs + T([x]) + T++ (rev xs) [x]  by [cost:func]
= 1 + Trev xs + 0 + T++ (rev xs) [x]       by [cost:prim, cost:var]
= 1 + Trev xs + T++ (rev xs) [x]           by arithmetic
= 1 + Trev xs + 1 + length xs              by defn of T++
= 2 + Trev xs + length xs                  by arithmetic
```

5.4.4 Complexity measures for these three examples

What size measure makes sense for these three examples?

The main variable is a list, so the obvious way to measure the size of the input is to takes its *length*.

Accordingly, we will re-express the three complexity measures as functions of *size*, where we take size to be length. Call the function C (for *c*ounting the *c*ost) instead of T.

In the definition of T_{++}, I've replaced `ys` with its randomly chosen length k; as the formulae show, the length k is irrelevant to the calculation.

$$\begin{aligned}
T_{sum}\ \texttt{[]} &= 1 \\
T_{sum}\ \texttt{(x:xs)} &= 1 + T_{sum}(\texttt{xs}) \\
\\
T_{++}\ \texttt{[] ys} &= 1 \\
T_{++}\ \texttt{(x:xs) ys} &= 1 + T_{++}\ \texttt{xs ys} \\
\\
T_{rev}\ \texttt{[]} &= 1 \\
T_{rev}\ \texttt{(x:xs)} &= 2 + T_{rev}\ \texttt{xs + length xs}
\end{aligned}$$

$$\begin{aligned}
C_{sum}(0) &= 1 \\
C_{sum}(n+1) &= 1 + C_{sum}(n) \\
\\
C_{++}(0)(k) &= 1 \\
C_{++}(n+1)(k) &= 1 + C_{++}(n)(k) \\
\\
C_{rev}(0) &= 1 \\
C_{rev}(n+1) &= 2 + C_{rev}(n) + n
\end{aligned}$$

5.5 Recurrence Rules

The final step in the calculation of complexity is to express the function C in *closed form*, rather than as a recursive definition.

Recursion starting at 0
If f is defined by

```
f 0 = d
f n = f(n-1) + b.n + c
```

then

$$f(n) = \frac{b}{2}n^2 + \left(c + \frac{b}{2}\right) n + d$$

Recursion starting at 1
If f is defined by

```
f 1 = d
f n = f(n-1) + b.n + c
```

then

$$f(n) = \frac{b}{2}n^2 + \left(c + \frac{b}{2}\right) n + (d - c - b)$$

5.5.1 Closed form complexity function for `sum`

Putting $f = C_{sum}$, we have the definition

```
    f 0     = 1
    f (n+1) = 1 + f n
ie. f n     = 1 + f(n-1)
```

This corresponds to the choices $d = 1, b = 0, c = 1$, whence the closed-form solution is given by

$$f(n) = \frac{0}{2}n^2 + \left(1 + \frac{0}{2}\right) n + 1$$

i.e.,

$$C_{sum}(n) = n + 1$$

So `sum` $= O(n)$.

5.5.2 Closed form complexity function for (++)

Putting $f = C_{++}$, and ignoring k, we have the definition

```
    f 0     = 1
    f (n+1) = 1 + f(n)
ie. f n     = 1 + f(n-1)
```

This is identical to the definition of C_{sum}, so we have

$$C_{++}(n) = n + 1$$

So `(++)` $= O(n)$.

5.5.3 Closed form complexity function for `rev`

Putting $f = C_{rev}$, we have the definition

```
    f 0     = 1
    f (n+1) = 2 + f(n) + n
ie. f n     = 2 + f(n-1) + (n-1)
            = f(n-1) + n + 1
```

This corresponds to the choices $d = 1, b = 1, c = 1$, whence the closed-form solution is given by

$$f(n) = \frac{1}{2}n^2 + \left(1 + \frac{1}{2}\right) n + 1$$

i.e.,

$$C_{rev}(n) = \frac{1}{2}(n + 1)(n + 2)$$

So `rev` $= O(n^2)$.

5.6 Summary

1. This week we've seen how to analyse the *time complexity* of various simple functions.
2. Because we have assumed that programs are evaluated *strictly*, the complexities we compute are likely to be higher than for true Haskell programs (because Haskell is a lazy language).
3. We have not investigated higher-order functions (functions of functions), nor have we considered functions defined on user-specified data types, but the techniques involved can obviously be extended to cover those areas as well.
4. The strategy we use involves three basic steps.
   (a) Work out how many steps the function takes to handle arbitrary inputs. For a function of type `f::a->b`, the result we get is a *step-counting* function $\mathtt{T}_f$`::a->Int`.
   (b) Replace the inputs with some measure of their size; this gives us a complexity measure $\mathtt{C}_f$`::Int->Int`.
   (c) Convert the recursion based definition of C_f into a *closed-form* expression. This makes it easy to estimate the complexity class of the function (*i.e.*, whether it is linear, quadratic, cubic, exponential, etc).

Chapter 6

Equational Reasoning and Structural Induction

6.1 Introduction

Over the past few weeks, we've come across various aspects of the Haskell class system, but we've not really had time to analyse it in detail. We'll be doing that today. At the same time, we'll investigate another feature of Haskell that we've so far only discussed in passing – Haskell's type system. We'll also take this opportunity to introduce some useful Haskell notation that's missing from Thompson's book [Tho99]. Further details of this notational feature – the use of ***field label*** s – can be found in Section 6 of the HUGS tutorial [HPF99].

6.2 Algebraic Data Types

Throughout the course so far, we've been defining algebraic data types using the keyword ***data***. You can use this keyword to define any types you like, as long as you keep to the right syntax. There are many sorts of algebraic data type construction, listed here in order of increasing intuitive complexity. As far as Haskell is concerned, however, there is no difference at all between these different sorts of types.

- `data Spectrum = Red | Orange | Yellow | Green | Blue | Indigo | Violet`
  The members of the type are listed one after the other. Technically, the terms `Red`, `Orange`, . . . , are ***nullary constructors***.
- `data Dimensions = ThreeD Float Float Float`
  The *constructor* `ThreeD` takes three values; in this case, each of those values is of type `Float`.
- `data Encapulate a = Enc a`
  This is a ***polymorphic type***. Whereas `ThreeD` will only accept three floats, `Enc` will accept an argument of any type.
- `data Stack a = Empty | Push a (Stack a)`
  This is a ***recursive type***, as well as being a polymorphic type. The type we're defining appears in the definition itself.

All you have to remember is that the ***constructors*** must start with a capital letter (not a number). If you want to define a polymorphic type, the type variable can be any word beginning with a

lower-case letter. So you could write `Stack entryType` instead of `Stack a` if you find it easier to understand. You can't use the same constructor name more than once in the same scope; for example, you can't use `Empty` to mean both an empty stack and an empty queue in the same bit of code. But you can pick and mix the various types of constructor, and use them all in a single algebraic type, e.g.

```
data MixtureOfCarStuff =
   Car | Van | Paint Spectrum |
   Speed Float | Stack MixtureOfCarStuff
```

Note also that type names and constructor names are regarded as belonging to different name spaces in Haskell, so it's entirely OK to use the word `Person` both as a type name and a constructor in the following definition.

```
type Name   = String
type Phone  = Int
type URL    = String
data Person = Person Name Phone URL
     deriving (Eq, Show)
```

The keyword `deriving` is used to tell Haskell to make the type a member of one or more type classes, and to use default versions of relevant functions. As we'll see shortly, writing

```
deriving (Eq, Show)
```

means that Haskell creates default implementations of the functions

```
(==) :: Person -> Person -> Bool
show :: Person -> String
```

Sometimes, however, the default implementations don't do what you want them to. We'll see below how to define custom implementations instead.

6.3 Field Labels

Everything we've discussed so far should be familiar to you, as should the idea of using selection functions to access the particular values in a tuple type, like this:

```
data Person = Person Name PhoneNumber URL
     deriving (Eq, Show)

name :: Person -> Name
name (Person n _ _) = n

phone :: Person -> Phone
phone (Person _ p _) = p

url :: Person -> URL
url (Person _ _ u) = u
```

Clearly, however, if you had a tuple type with a large number of components, it would take quite a while to define all of the access functions. To make things easier, Haskell allows the use of ***field***

label declarations. Another way of defining the `Person` type with these access functions is to write

```
data Person = Person {
  name  :: String,
  phone :: Int,
  url   :: String
} deriving (Eq, Show)
```

Given such a definition, there is a standard easy way to update values. For example, if you need to update my phone number, you can write

```
-- Old phone number
oldmike = Person "mike" 2221800 "dcs/~mps"

-- New phone number
newmike = oldmike{phone = 2221841}
```

6.4 Haskell's class system

A ***type class*** is a set of types, all of which have instances of certain specific polymorphic functions defined on them. Classes are defined by using the `class` keyword, and listing the relevant polymorphic functions. For example, this is how the Haskell Prelude defines the ***Eq*** class:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimal complete definition: (==) or (/=)
  x == y      = not (x/=y)
  x /= y      = not (x==y)
```

Notice that classes can include default implementations of their functions – but these implementations need not make sense! In this case, for example, the functions `(==)` and `(/=)` are each defined in terms of the other, so if you tried invoking them as they stand you'd get an infinite evaluation loop. To get round this problem, we need to *override* at least one of the two default implementations in each type we decide to include in `Eq`. Each class definition typically includes a comment explaining which functions you need to override to make your instance work properly. In this case, we're told to override either `(==)` or `(/=)` – either will do.

Having decided which functions to override, we declare a type to be a member of a class using the ***instance*** keyword. For example, having defined the `Person` type, I can declare it to be a member of `Eq` as follows:

```
data Person = Person {
  name  :: String,
  phone :: Int,
  url   :: String
}
instance Eq Person where
  -- specific definition of (==)
  (Person n p u) == (Person n' p' u')
      = (n == n') && (p == p') && (u == u')
```

It is precisely because this definition is so simple (and so obvious) that Haskell can work out for itself how you're likely to want to override the `(==)` function; that's what happens when you use the `deriving (Eq)` shortcut to defining these functions.

Now consider the ***Show*** class, which allows HUGS to display values on the screen.

```
class Show a where
  show      :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList  :: [a] -> ShowS

  -- Minimal complete definition: show or showsPrec
  show x          = showsPrec 0 x ""
  showsPrec _ x s = show x ++ s
  showList []     = showString "[]"
  showList (x:xs) = showChar '[' . shows x . showl xs
      where showl []     = showChar ']'
            showl (x:xs) = showChar ',' . shows x . showl xs
```

Once again, we're told that we need to override one of two functions, ***show*** or `showsPrec`, and it's normally the first of these that gets overridden. For example, if we define

```
data List a = EmptyList | Singleton a | Join (List a) (List a)
     deriving (Show)
```

we'll get an implementation of `show` constructed for us, but it's not particularly inspired. What we see on-screen are strings like

```
Join (Singleton "maths") (Join (Singleton "is") (Singleton "fun"))
```

We can get a more intuitive display by redefining `show`, like this:

```
instance (Show a) => Show (List a) where
  show EmptyList     = "E"
  show (Singleton x) = show x
  show (Join xs ys)  = "{" ++ (show xs) ++ "|" ++ (show ys) ++ "}"
```

which makes lists appear on-screen like this:

```
{"maths"|{"is"|"fun"}} :: List [Char]
```

Notice the ***constraint*** in this definition. We're saying that `List a` is a member of `Show`, *provided* `a` is itself a member of `Show`. We need this constraint to be satisfied, because our definition of `show` uses `show x`, which is `a`'s version of `show`. It's worth noting that this is a real constraint, in the sense that many important classes in Haskell are *not* members of `Show`. For example, there is no obvious way to display a value which happens to be a function.

Going in the other direction, we could also declare `List a` to be a member of `Read`, which would allow us to type in strings of the form `{"maths"|{"is"|"fun"}}`, and have Haskell interpret them correctly as values of type `List a`.

6.5 Example of classes in action

We can do some fairly impressive stuff with classes. For example, you have seen many examples of computational models defined in terms of ***state transition systems***, including the finite state machine (FSM), pushdown automaton (PDA) and Turing machine (TM), and there are many others. In each case, the model is assumed to have an intrinsic *configuration* that changes from one move to the next. This configuration normally involves more than just the underlying state-transition system. For example, if the model is intended to act as a recogniser, the configuration would normally include details concerning how much of the input string still needs to be processed. If the model includes outputs, the configuration will include details of the outputs generated so far.

Ignoring these implementation details, we can give a fairly detailed definition of what it means for a type `cfg` to be usable for modelling the configurations of a computational model based on state-transitions; the only assumption I'm making is that transition labels are characters (so that a machine can be used to recognise stings). If you prefer, you could define `Label` differently, and modify the code accordingly.

```
type Label = Char

class (Eq cfg) => Model cfg where
  initialise   :: [Label] -> cfg
  acceptState  :: cfg -> Bool
  doNextMove   :: cfg -> cfg
  runFrom      :: cfg -> cfg
  runModel     :: [Label] -> cfg
  -- Default implementation
  runModel = runFrom . initialise
```

Next, we need to define what we mean by an FSM. The key features of the FSM model are the *transition set*, *initial state* and *terminal states*, so these are the components we'll include. Notice that we're not intereste din the *dynamics* of the FSM model at this stage – we leave that side of things to the `Model` class.

```
type Transitions stateset = [(stateset, Char, stateset)]

class (Eq stateset) => FSM stateset where
  transitions :: Transitions stateset
  initialState :: stateset
  haltStates :: [stateset]
```

Next, we have to say what the configurations of the (FSM + string) system should look like. This is straightforward, because all we need to keep track of are the *current state* and the remaining *string* still to be processed.

```
data FSMConfig stateset = FSMConfig{
  state :: stateset,
  input :: String
} deriving (Eq, Show)
```

To complete the general picture of how an FSM works, we need to explain how these configurations are used and updated as the model runs. Notice the declaration here: we're saying that the type `(FSMConfig stateset)` can be used as the configuration space for a computational model, whenever the `stateset` itself can be used as the stateset of an FSM.

```
instance (FSM stateset) => Model (FSMConfig stateset) where
  initialise  str = FSMConfig initialState str
  acceptState cfg@(FSMConfig s i) = (s `elem` haltStates) && (null i)
  doNextMove  cfg@(FSMConfig s i)
    | null i = cfg
    | otherwise = FSMConfig nextstate (tail i)
        where nextstate = head nextstates
              nextstates = [ s' | (s, c, s') <- transitions, c == head i ]

  runFrom cfg@(FSMConfig s i)
    | null i = cfg
    | acceptState cfg = cfg
    | otherwise = runFrom (doNextMove cfg)
```

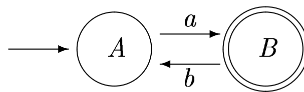


Figure 6.1: A simple FSM

To define a *specific* FSM, we need to describe its specific initial state, halt states and transitions. For the machine in Figure 6.1, for example, we might declare

```
data MyStates = A | B deriving (Eq, Show)

instance FSM MyStates where
  transitions  = [(A, 'a', B), (B, 'b', A)]
  initialState = A
  haltStates   = [B]
```

Finally, it makes sense to define a `recognises` function, that checks whether or not any given string is recognised by the FSM. Unfortunately, this function would have type `(String -> Bool)`, which means it *cannot* be included as part of any class declaration, because it is not polymorphic; the whole point of the '`class ClassName a`' declaration is that it tells us which functions must be defined on the type `a` if it is to be a member of `ClassName`; it doesn't make sense to include functions that don't mention `a` at all. Instead, we have to declare it outside the class, but make it clear which instance of `Model` we're using to handle the recognition process; we can do this by declaring the return type of `(runModel str)` explicitly, as shown here:

```
recognises :: String -> Bool
recognises str = acceptState (runModel str :: FSMConfig MyStates)
```

Collecting all the pieces together gives the following complete implementation of the FSM model of computation:

```
-- GENERAL FSM MODEL
--------------------
class (Eq cfg) => Model cfg where
  initialise   :: String -> cfg
  acceptState  :: cfg -> Bool
  doNextMove   :: cfg -> cfg
  runFrom      :: cfg -> cfg
```

```
  runModel     :: String -> cfg
  -- Default implementation
  runModel = runFrom . initialise

type Transitions stateset = [(stateset, Char, stateset)]

class (Eq stateset) => FSM stateset where
  transitions :: Transitions stateset
  initialState :: stateset
  haltStates :: [stateset]



data FSMConfig stateset = FSMConfig{
  state :: stateset,
  input :: String
} deriving (Eq, Show)

instance (FSM stateset) => Model (FSMConfig stateset) where
  initialise  str = FSMConfig initialState str
  acceptState cfg@(FSMConfig s i) = (s `elem` haltStates) && (null i)
  doNextMove  cfg@(FSMConfig s i)
    | null i = cfg
    | otherwise = FSMConfig nextstate (tail i)
        where nextstate = head nextstates
              nextstates = [ s' | (s, c, s') <- transitions, c == head i ]
  runFrom cfg@(FSMConfig s i)
    | null i = cfg
    | acceptState cfg = cfg
    | otherwise = runFrom (doNextMove cfg)



-- SPECIFIC FSM MODEL
---------------------
data MyStates = A | B deriving (Eq, Show)

instance FSM MyStates where
  transitions  = [(A, 'a', B), (B, 'b', A)]
  initialState = A
  haltStates   = [B]

recognises :: String -> Bool
recognises str = acceptState (runModel str :: FSMConfig MyStates)
```

6.6 Constraint propagation and typing

Consider the following functions:

```
  elem :: Eq a  => a -> [a] -> Bool
  sort :: Ord b => [b] -> [b]
  (.)  :: (d -> e) -> (c -> d) -> (c -> e)
```

Given these definitions, we can soon determine that

```
(elem . sort) [7,1,9] [ [], [7,1,9] ]
```

evaluates to *True*. But what *type* is the function `elem . sort`?

In general, a function signature is of the form "`f :: Constraint => Type`", which says that `f` has type `Type` *provided* the constraint is satisfied. If the constraint isn't satisfied, the function is meaningless. For example, the function `(+3)` has type

```
(+3) :: Num a => a -> a
```

because it's only meaningful to add 3 to a value if that value is a numeric.

Haskell uses just a few basic ***typing rules***.

6.6.1 Rule 1. Function Application

$$\begin{array}{rl} \text{Given} & f :: Cons_f \Rightarrow \sigma \rightarrow \tau \\ & e :: Cons_e \Rightarrow \sigma \\ \text{Deduce} & f\ e :: (Cons_f, Cons_e) \Rightarrow \tau \end{array}$$

For example, here's how we'd derive the type of `(+3) 5`.

$$\begin{array}{rcl} \texttt{(+3)} & \texttt{::} & \texttt{Num a} \Rightarrow \texttt{a} \rightarrow \texttt{a} \\ \texttt{5} & \texttt{::} & \texttt{Num a} \Rightarrow \texttt{a} \\ \hline \texttt{(+3) 5} & \texttt{::} & \texttt{(Num a, Num a)} \Rightarrow \texttt{a} \\ & \equiv & \texttt{Num a} \Rightarrow \texttt{a} \end{array}$$

6.6.2 Rule 2. Type Instantiation

$$\begin{array}{rl} \text{Given} & f :: Cons_f \Rightarrow \tau \\ \text{Deduce} & f :: Cons_f\{\sigma/a\} \Rightarrow \tau\{\sigma/a\} \end{array}$$

For example, here's how we'd show that `5 :: Int`.

$$\begin{array}{rrcl} & \texttt{5} & \texttt{::} & \texttt{Num a} \Rightarrow \texttt{a} \\ \Rightarrow & \texttt{5} & \texttt{::} & \texttt{(Num a)\{Int / a\}} \Rightarrow \texttt{a\{Int / a\}} \\ \Rightarrow & \texttt{5} & \texttt{::} & \texttt{Num Int} \Rightarrow \texttt{Int} \\ \Rightarrow & \texttt{5} & \texttt{::} & \texttt{Int}\ \text{since}\ \texttt{Int} \in \texttt{Num} \end{array}$$

6.6.3 Rule 3. Abstraction

$$\begin{array}{rl} \text{Given} & (x :: \sigma)\ \textit{implies}\ f :: Cons_f \Rightarrow \tau \\ \text{Deduce} & \backslash x \rightarrow f :: Cons_f \Rightarrow \sigma \rightarrow \tau \end{array}$$

For example, what type is `\ x -> (x == x)`? To answer this question, we start by *assuming* that `x ::` σ. Since `(==) :: Eq a => a -> a -> Bool`, we can apply type instantiation to give `(==) :: Eq` σ `=>` σ `->` σ `-> Bool`, whence function application tells us that `(x == x) :: Eq` σ `=> Bool`. Applying the abstraction rule now tells us that `\ x -> (x == x) :: Eq` σ `=>` σ `-> Bool`.

Finally, what about the question that opened this section? What type is `elem . sort`? To answer this question, we need to remember that `elem . sort` is shorthand for `(.) elem sort`, so that evaluating the type of this function requires two stages:

1. Find the type of `(.) elem` by matching up the types of `(.)` and `elem`;
2. Do the same with `(.) elem)` and `sort`, and so find the type of `(.) elem sort`.

Step One: Find the type of `(.) elem`

We know that `(.)` is expecting an argument of type `(d -> e)`, and that the argument `elem` is a function of type `a -> ([a] -> Bool)`. So we need to match up `d` ⇝ `a` and `e` ⇝ `[a]->Bool`. Pictorially, we match up the types in the original declarations:

```
elem ::  Eq a =>  a  ->  ([a] -> Bool)
(.)  ::  (        d  ->        e        ) -> (c -> d) -> (c -> e)
```

and then do a type substitution followed by a function application:

```
     elem ::  Eq a =>  a  ->  ([a] -> Bool)
     (.)  ::  (        a  ->   [a] -> Bool   ) -> (c -> a) -> (c -> [a]->Bool)
-------------------------------------------------------------------------------
 (.) elem ::  Eq a =>                             (c -> a) -> (c -> [a]->Bool)
```

Step Two: Find the type of `(.) elem sort`

We now know that `(.) elem` is expecting an argument of type `(c -> a)`, while `sort` is an argument of type `[d] -> [d]`. So we need to match up `c` ⇝ `[b]` and `a` ⇝ `[b]`. Again, it's useful to look at things pictorially. We match up types in the original declarations:

```
(.) elem ::  Eq a => (    c -> a    ) -> (c -> [a]->Bool)
    sort ::  Ord b =>   [b] -> [b]
```

Performing the type substitution and function application now gives the required result:

```
 (.) elem ::  Eq a => (          [b] -> [b]  ) -> ([b] -> [[b]]->Bool)
     sort ::  Ord b =>           [b] -> [b]
------------------------------------------------------------------------
(.) elem sort ::  (Eq [b], Ord b) =>              ([b] -> [[b]]->Bool)
```

Notice that we're requiring `[b]` to be in `Ord`. Since this is automatically true whenever `b` is in `Ord`, we can replace the constraint `Ord [b]` with the simpler constraint `Ord b`. Finally, as it's more usual to give functions in terms of the type variable `a`, we perform one last type substitution, giving:

```
(elem . sort) :: (Eq a, Ord a) => [a] -> [[a]] -> Bool
```

6.7 Summary

In this section you've learned about the use of field labels in algebraic type definitions, and their usefulness when tuple types contain large numbers of entries. We've explored the Haskell class

system in detail, and seen both how to declare a class, and how to construct instances of one. The functions declared in a class are always polymorphic, and can have default implementations included, but it's normally necessary to override one or more of these defaults in each instance of the class. We've looked at a fairly complex (and useful) example of classes in action: a fully general implentation of the (deterministic) Finite State Machine model of computation.

We've also examined Haskell's typing mechanisms. Despite the apparent complexity of typing decisions, Haskell gets by with just a small number of very simple rules, each of which can easily be machine-implemented.

Chapter 7

Complexity analysis

7.1 Introduction

What have we learned in this course so far?

7.1.1 Abstract Data Types

We started by looking at what it means for something to be stack. Since stacks can be programmed in just about *any* language, the answer has to be independent of *all* languages. We achieve this by defining stacks *abstractly* using an *abstract data type*, or *ADT*. We identify the various types that are relevant to the workings of the ADT, the various functions that characterise its behaviour (*syntax*), and the equations that need to be satisfied by those functions (*semantics*).

7.1.2 ADTS and Haskell

Next, we looked at various techniques for implementing ADTs in Haskell (or any other functional language). This is not entirely straightforward, as we need to decide how error messages are to be handled. Including messages as part of a function's return type can be achieved using Haskell's polymorphic `Either a b` type, but the code gets quite messy. Alternatively, we can set aside specific values in the return type, and interpret them *as if* they were error messages. This works well in some situations (for example, C^{++} programs return 0 on successful completion, and use negative values to indicate error conditions, but sometimes it isn't possible to set aside a special value. For example, if we wanted to set aside a specific stack, *ErrorStack* to indicate an error condition, how would we actually define *ErrorStack*?

Whatever stack we use, how do we ensure that it can never occur naturally in the course of running an error-free program? By far the easiest solution is to use Haskell's polymorphic `error :: String -> a` function, but even this isn't perfect. If an error occurs, the entire program grinds to a halt. If we'd used the `Either a b` solution instead, we'd have been able to decide at run-time whether to halt the program, or take some other action instead. And as you saw last semester, you can also use the `Maybe a` type for exception handling.

7.1.3 Program Proof

No matter which technique you use to implement an ADT, you need some way to show that you've implemented the code *correctly*. Establishing the correctness of a program is notoriously difficult,

but there are at least three approaches that have been considered viable at one time or another. Using *X-machine testing techniques* it is possible to build systems whose correctness can be determined by running a finite set of tests. Alternatively, one can build programs automatically by transforming formal specifications into concrete code, one step at a time; as long as the translation steps are *sound* the final program is guaranteed to meet its specification. This approach, called *refinement*, and is commonly associated with specification languages like Z. The third technique, and the one we considered in some detail, is to *prove* that your program meets its specification. *Floyd-Hoare* logic was developed with imperative languages very much in mind, but as we saw, it can easily be extended to handle functional languages as well. But the technique that comes into its own for providn things about functional programs is *induction*; this reflects the fact that functional programs are typically constructed using recursion.

7.1.4 Haskell's class and type systems

As well as proving that your planned algorithm is correct, you need to be sure you're programming it correctly, and this requires detailed knowledge of the target programming language. We looked at Haskell's class system in detail, and showed that classes offer a convenient way of grouping types together. Classes are also useful for expressing *constraints* on functions, as when we say that the function `show` can only sensibly be defined on the type `Stack a` if `a` is itself a member of the `Show` class. Knowing about the type system is also essential. The examples we considered in our presentation of Floyd-Hoare logic didn't involve constraints; for full generality we'd need to investigate their relevance to the central technique, and doing this will only be possible because we understand how Haskell propagates constraints during type determination.

This week we'll be linking some of these concepts together, and revisiting some of our earlier assumptions. We'll return to our analysis of the `Stack a` type, and ask again how best to implement it. To keep things general, we'll look at the same implementations in both Haskell and C^{++}/JAVA-like terms.

7.2 From Specification to Representation

Let's look again at our ADT of a stack. It defines a stack to be a system involving five functions, which interact in ways specified by various axioms. Notice that I've made `Msg` more sophisticated in this version, as the types of error message associated with not being able to generate an object depend on the type of that object. The `Msg` type is accordingly taken to be polymorphic in the following type signatures.

$$\begin{aligned}
createStack &: Stack\ a \\
push &: a \to Stack\ a \to Stack\ a \\
emptyStack &: Stack\ a \to Bool \\
top &: Stack\ a \to (a \cup Msg\ a) \\
pop &: Stack\ a \to (Stack\ a \cup Msg\ (Stack\ a))
\end{aligned}$$

Although it's tempting to talk about the stack as a data *structure*, this terminology isn't consistent with the ADT description we've given, for the ADT tells us about functions and their interrelationships, but says nothing at all about the physical or logical arrangement of data *within* a stack. When we model an ADT using operations on a data structure, we call the description a ***representation*** of the ADT. It's important to realise that any given ADT may have *many* representations, some more efficient than others. It's also important to realise that a representation is again *abstract*, because data structures make sense in their own right, independently of any programming language.

Because this course focusses on Haskell, I automatically implemented the `Stack a` type as an algebraic data type (a type defined from scratch using the `data` keyword). Some of you will have considered an obvious alternative, namely implementing stacks using Haskell's standard list constructions. Both approaches were intuitively sensible, but therein lies the problem; our goal is to write programs *scientifically*, and there's no reason to believe that intuition is a valid substitute for technical rigour.

Suppose you were asked to *justify* your choice of lists (or algebraic data types) as the underlying representation for stacks – how would you do it? Answering this question is the subject of this week's lecture. We'll look at the stack ADT, and try to work out what features have to be true of a data structure for it to be a valid representation; then we'll investigate whether lists and algebraic data types both satisfy the requirements. Along the way we'll find that the most obvious representations (and hence implementations) aren't always the most efficient.

7.2.1 What features need to be represented?

The most obvious question to address is: *what features need to be included in the representation*? To answer this question we need to look again at the stack functions, and see what they tell us.

- $createStack\colon\ \to Stack\ a$
  This function tells us nothing about the way a stack should be represented as a data structure; all it tells us is that a *createStack* functions needs to be definable.
- $push\colon a \to Stack\ a \to Stack\ a$
  Again, this function tells us nothing about how to represent a stack, except that a *push* function should be definable.
- $emptyStack\colon Stack\ a \to Bool$
  This function *does* tell us something about representing stacks. It tells us that the representation must include information as to whether the stack is empty.
- $top\colon Stack\ a \to (a \cup Msg\ a)$
  This function tells us that it should be possible to identify the top of the stack.
- $pop\colon Stack\ a \to (Stack\ a \cup Msg\ (Stack\ a)$
  This function tells us nothing useful, except that a *pop* function must be definable.

Notice what's going on here. When we implemented `Stack a` as an algebraic data type, it was the *constructors* that played the most obvious role; the constructors are *createStack*, and *push*, and its the result applying these functions that became the constructors in the data declaration. Thus, the result of applying *createStack* is an empty stack, and the result of applying *push* is a stack with an element pushed onto the front, and these two results correspond directly to the `EmptyStack` and `Push` constructor terms:

Haskell declaration	ADT constructor
`data Stack a`	
`= EmptyStack`	$createStack()$
`| Push a (Stack a)`	$push(x, s)$

In contrast, it's now the *observers* that matter. Constructors tell us very little about potential representations, whereas observers tell us very precisely that certain pieces of information must be accessible. In the case of *Stack a*, the observers tell us that , *in addition to representing the stack of entries itself*, the representation should include the following items (the names I've chosen for these items are unimportant)

Component	Type	Represents
stackrep	Container	Represents the stack itself
empty	Boolean flag	Whether or not the stack is empty
top	Pointer to entry	Points to the top entry in the stack

Writing $Ptr\langle Entry\rangle$ to mean *pointer to entry*, and $Container\langle Entry\rangle$ to mean a container for items represented using the data structure *Entry*, these observations tells us that representations of the stack ADT should ultimately be of the form

$$(stackrep, empty, top) \in Container\langle Entry\rangle \times Boolean \times Ptr\langle Entry\rangle$$

The semantics also play an important role in determining which representations are sensible, because they tell us about the internal structure of the container used to represent the stack itself. The fact that entries are ejected from the stack in LIFO order tells us in particular that the container has to be *ordered*. Given any stack of two or more entries, we can tell which one was inserted first.

7.2.2 Choosing a container

What container shall we choose? If our ultimate aim is to build a Haskell program, an obvious choice is to use a list, as this is an ordered container type that is readily available. But what if we don't know what the ultimate language is going to be? In general, the only ordered container type likely to be available in a randomly chosen (but commercially relevant) language is an ***array***. An array is like a list in some ways, but has important differences.

1. The size of an array, *size*, is fixed at construction. A list can expand indefinitely.
2. The array won't allow you to insert more than *size* entries. You can always add another entry to a list.
3. Even if the array contains no entries, it still takes up its full quota of space in memory. Lists can contract as well as expand.
4. It is easy to access any element of the array, because arrays are *random-access* structures (we say that array access takes *constant time*). Lists are less efficient, because you have to work down the list recursively, one entry at a time, until you reach the entry you're after (list access takes *linear time*).

The size restrictions associated with arrays have an important corollary. Clearly, if we try to model our ADT using an array-baesd implementation, we will need to take account of new error conditions that weren't present in the original ADT. The ADT takes it for granted that new values can always be pushed onto the stack, and so no error can occur; the situation for arrays is rather different. What this tells us is that implementations can sometimes fail not because the code has been constructed wrongly, but because they're constructed around a less-than-perfect representation. This doesn't mean that array-based implementations are unacceptable, only that we need to be aware of their pitfalls, and plan ahead accordingly.

7.3 From representation to implementation

Ultimately, it's up to us whether we choose to model stacks using lists or arrays, or some other container type altogether, because the stack is, as was pointed out above, essentially *abstract*. If we decide to use lists and then find that lists aren't available in the target language, we can always

write some code to implement them. Likewise, if we pick arrays, and then find that the target language only supports lists, we can write some code to implement arrays in terms of lists. Either way, we will eventually be able to use our representation to implement our original stack ADT.

7.3.1 Implementation using arrays

To specify an array, we need to say what size it is, so for the sake of argument we will assume that the array we're going to use is called *Array*, and contains precisely N cells, each of which can contain precisely one entry. Array indices will start at 1, not 0. There are many representations we can build, all based on arrays. For example:

Component	Represented by	Of type
stackrep	$data[1..N]$	$Array\langle Entry\rangle$
empty	*flag*	*Boolean*
top	*topIndex*	$Integer \in [1..N]$

represents a stack as an array, together with a Boolean *flag*, and an integer giving the index in the array of the top element. With a little thought, however, we can simplify this representation, by setting $topEntry = 0$ to represent $flag = True$. This allows us to streamline the representation, giving

Component	Represented by	Of type
stackrep	$data[1..N]$	$Array\langle Entry\rangle$
empty	$topIndex == 0$	*Boolean*
top	*topIndex*	$Integer \in [1..N]$

According to this representation, a sensible JAVA implementation of a stack of doubles might be

```
class DoubleStack {
  double[] data;
  int topIndex;
  DoubleStack() { data = new double[N]; topIndex = 0; }
}
```

However, we can't stop there – we need to represent the stack functions as well. Let's represent each function *foo* in the ADT as a function *fooRep* in the representation. Then the functions need to be represented as functions of the following types:

$$\begin{aligned}
createStackRep &: Array\langle Entry\rangle \\
pushRep &: Entry \to Array\langle Entry\rangle \to Array\langle Entry\rangle \\
emptyStackRep &: Array\langle Entry\rangle \to Boolean \\
topRep &: Array\langle Entry\rangle \to (Entry \cup Msg\langle Entry\rangle) \\
popRep &: Array\langle Entry\rangle \to (Array\langle Entry\rangle \cup Msg\langle Array\langle Entry\rangle\rangle)
\end{aligned}$$

Should we implement these functions as ***method*** s in the `IntStack` class, or as functions in separate class? This is not as simple as it first seems, because the whole point of an ADT is that we want the implementation to be hidden from the user. If we implement the functions as methods in `IntStack`, we break our ***information hiding*** obligations. This implementation uses the fact that by default classes aren't accessible outside their package. Users need to import `com2020.*`.

```
package com2020;
public class StackEmptyException extends Exception {
```

```
  public StackFullException() { super("stack is empty"); }
}
public class StackFullException extends Exception {
  public StackFullException() { super("stack is full"); }
}

class DoubleStack {
  private double[] data;
  private int topIndex;
  private DoubleStack() {data = new double[StackRep.MAXSIZE]; topIndex = 0;}

  static DoubleStack createStack() {return new DoubleStack();}
  boolean emptyStack () {return (topIndex == 0);}

  DoubleStack push(double val) throws StackFullException {
    if (topIndex < StackRep.MAXSIZE) {
      topIndex++;
      data[topIndex] = val;
      return this;
    } else throw new StackFullException();
  }

  double top() throws StackEmptyException {
    if (topIndex > 0) {return data[topIndex];}
    else throw new StackEmptyException();
  }

  DoubleStack pop() throws StackEmptyException {
    if (topIndex > 0) {topIndex--; return this;}
    else throw new StackEmptyException();
  }
}

public class StackRep {
  public static int MAXSIZE = 50;
  public static DoubleStack createStack() {
    return DoubleStack.createStack();
  }

  public static DoubleStack push (double val, DoubleStack stack)
    throws StackFullException
  {
    return stack.push(val);
  }

  public static boolean emptyStack (DoubleStack stack) {
    return stack.emptyStack();
  }

  public static double top (DoubleStack stack)
    throws StackEmptyException
  {
    return stack.top();
```

```
    }

    public static DoubleStack pop (DoubleStack stack)
      throws StackEmptyException
    {
      return stack.pop();
    }
  }
```

Notice the implementation of `pop`. If we were using lists in Haskell, we would have to 'throw away' the popped value, and shrink the list by one entry, all of which takes effort and slows the program down. Using an array, all we have to do is decrement the value of `topIndex`; the fact that the popped value is still present in its old cell is irrelevant. If the stack ever expands again, we'll overwrite the value; and if it doesn't expand again we'll never be in a position to access that particular cell in any case.

7.3.2 Implementation using lists

Now let's consider a representation based on lists. As before, we need a container to represent the stack itself, as well as two other terms of expressions.

Component	Represented by	Of type
stackrep	$[a]$	$List\langle Entry\rangle$
empty	*empty*	*Boolean*
top	*head*	$List\langle Entry\rangle \rightarrow Entry$

In this suggested representation, I've taken *empty* to be a Boolean flag that is set to *True* if and only if the list is empty, while *head* is the standard function for lists that returns the front entry.

What happens next depends on the target language. For JAVA, we might well declare a stack of strings something like this:

```
  class StringStack {
    private Vector data;
    private boolean empty;
    private StringStack() { data = new Vector(); empty = true; }

    static StringStack createStack() {return new StringStack ();}

    StringStack push(String str) {
      data.addElement(str);
      empty = false;
      return this;
    }
    boolean emptyStack () {
      return empty;
    }

    String top() throws StackEmptyException {
      if (!empty) {
        return (String)(data.lastElement());
      }
      else throw new StackEmptyException();
```

```
    }

    StringStack pop() throws StackEmptyException {
      if (!empty) {
        int pos = data.size() - 1;
        data.removeElementAt(pos);
        empty = (pos == 0);
        return this;
      }
      else throw new StackEmptyException();
    }
  }
```

If the target language is functional, like Haskell, we need to be a bit more inventive, because Haskell is a ***stateless*** language. That is, once we've defined a value, we can't change it. In particular, once we've set the value of *empty*, we'll never be able to change it again, no matter what happens to the stack. The only way around this is to make *empty* a *function* instead, so that it can decide what value to return depending on the circumstances at the time. The implementation therefore *adapts* the representation like this:

Component	Represented by	Of type
stackrep	$[a]$	$List\langle Entry\rangle$
empty	*empty*	$List\langle Entry\rangle \to Boolean$
top	*head*	$List\langle Entry\rangle \to Entry$

Given this re-interpretation, we can define stacks in Haskell. Just as we used packages to hide information in JAVA, we'll use modules to hide information in Haskell; and we'll use the `error` function to avoid the need to implement the `Msg` type. Notice that the `module` declaration specifies precisely which functions can, and hene which function can't, be used outside the module; notice in particular that we've exported the type name itself (but not the fact that it's a synonym for lists) as this may also be required.

```
module StackADT (Stack, createStack, push, emptyStack, top, pop) where

data Stack a = Stack [a]
  deriving (Eq, Show)

createStack :: Stack a
createStack = Stack []

push :: a -> Stack a -> Stack a
push x (Stack s) = Stack (x:s)

emptyStack :: Stack a -> Bool
emptyStack (Stack s) = null s

top :: Stack a -> a
top (Stack s)
  | null s    = error "stack is empty"
  | otherwise = head s

pop :: Stack a -> Stack a
pop (Stack s)
```

```
    | null s     = error "stack is empty"
    | otherwise = Stack (tail s)
```

Why couldn't we simply have declared `type Stack a = [a]`? The problem arises in HUGS when you try to determine the type of a stack. If you type in `push 5 createStack`, it replies with the evaluation `[5] :: [Integer]`, rather than `[5] :: Stack Integer`. You can get round this by using the `data` declaration (or a `newtype` declaration if you prefer), but you should avoid using field labels, lest HUGS generate descriptions of the form `Stacklist=[5] :: Stack Integer` which reveal the internal structure of the implementation.

7.4 Summary

In this section of the course we've examined the process by which an ADT specification is converted into an implementation, showing the relevance of *representation*. We can think of representation as tanding midway between the abstraction of an ADT and the concreteness of code, but the exact position of a representation along this spectrum depends on the target language. If the representation happens to use the same basic constructions as are present in the language, we can regard the representation as a *bona fide* implementation. If it is relatively unrelated, it makes more sense to think of it as a slightly more concrete, though still essentially abstract, specification.

Chapter 8

Advanced HASKELL: the class and type systems

8.1 Introduction

This is the first of two lectures looking at specific ADTs and their applications.

- Looked at stacks so far: important for computational modelling (the push-down automaton in particular relies on the stack).
- But there are many other **important** data structures.

Just about every data structure you'll find in C-type languages like C^{++} and JAVA is based on one or other of these ADTs [Nel95]. List structures in particular are important in functional languages like Haskell [HPF99, Tho99]. Here's a quick summary of the various ADTs' basic properties.

8.1.1 Queues

- A ***queue*** is like a stack, except FIFO. Cannot identify or interrogate an entry in the middle of the queue without first removing all of the intervening entries.
- A ***priority queue*** is a queue in which the elements have numbers (called *priorities*) associated with them. If two entries have the same priority the standard FIFO rule applies, but entries with higher priorities are removed from the queue before those with lower priorities.
- A ***deque*** is a *double-ended queue*, in which data can be inserted at, and removed from, either end of the structure. A deque is, therefore, both a stack and a queue, and like all stacks and queues it is not possible to identify or interrogate entries in the middle of a deque directly.

8.1.2 Other ordered structures

Lists

- An ordered structure like the stack and queue, but unlike queues and stacks it is possible to ***traverse*** a list, and so determine what entries it contains, without actually removing any of them.

- However, we cannot simply access the entries at will.
- For example, in a standard Haskell list, the only way to find out what the nth is, is to work down the list one entry at a time, visiting the 1st, 2nd, 3rd, ..., $(n-1)^{th}$ elements in turn, and only then reaching the element we're actually interested in.
- In general, lists can be extended indefinitely. All the elements of a list are of the same type.

Arrays

- An ordered structure like a list, but the similarities end there.
- Arrays are ***random access*** structures, meaning that it takes no longer to find the nth entry than the first.
- On the other hand, arrays are *bounded*, and have a definite *size*.
- An array need not be full, but can never contain more entries than are allowed by its *size*.
- All the elements of an array are of the same type. An array with size N can be represented as a function $\mathit{array}\colon [1..N] \to \mathit{Entry}_\perp$, where $\mathit{Entry}_\perp = \mathit{Entry} \cup \{\perp\}$, and $\mathit{array}(n) = \perp$ if and only if the nth cell is empty.
- Some languages allow $\mathit{array}(n+1)$ to be defined even if $\mathit{array}(n)$ isn't; others insist that this should never be the case.

Tuples

- Like an array, except that different entries can have different types.
- Moreover, a tuple is always 'full', in the sense that a tuple defined to contain N entries cannot contain fewer than N entries.
- Like arrays, tuples are random-access; the function accessing the nth element of the tuple is often called the nth ***projection***.
- A tuple of length N can be thought of as a function $\mathit{tuple}\colon [1..N] \to \mathit{Type}$, where *Type* is the set of types.

8.1.3 Unordered structures

Sets

- An *unordered* structure: there is no sense in which any one element comes before any other.
- The defining feature of a set is that no entry can occur more than once.
- Sets are rarely programmed directly, but have to be represented using other data structures; this makes it easier to decide whether a given value is or is not a member of the set.
- We can represent a set S as a ***characteristic function*** $\chi_S\colon \mathit{Entry} \to \{0,1\}$, where $\chi_S(x) = 1$ if and only if $x \in S$.

Bags

- Exactly like a set, except that entries can occur more than once.
- The number of occurrences is called the entry's ***multiplicity***.
- We can represent a bag as a function mapping each entry onto its multiplicity, $\mathit{bag}\colon \mathit{Entry} \to \mathbb{N}$.

8.1.4 Trees

Trees

- A branching structure in which each entry can have more than one immediate successor.
- It is possible for the positions of two entries (called *nodes*) to be incomparable, but there is always some ***branch-point*** that is a predecessor to them both.
- The common predecessor of all the nodes in the tree is called the tree's ***root***.
- Notice that the entries themselves need not respect the structural ordering of the tree itself (the value at the root need not be the smallest entry in the tree)
- Any node's successors determine *subtree*s with the successors at their roots. If each node has at most two subtrees, the tree is a ***binary tree***.

Ordered Trees

- A binary tree whose entries satisfy the following condition.
- Given any node n in the tree, all of the entries in one of the subtrees (we'll call this subtree *left*) are smaller than or equal to the entry at the node, and all of the entries in the other subtree (*right*) are greater.

Heaps

- Like an ordered tree, except that the condition is less stringent.
- Given any node n, the value at n is greater than or equal to all of the values in either subtree.

8.1.5 Unexpected uses!

Tree-sort and *heap*-sort speed up sorting by apparently doing *extra* work! You insert data into the tree, then extract it again.

$$\textit{list of data} \quad \xrightarrow{insert} \quad ADT \quad \xrightarrow{extract} \quad \textit{sorted list of data}$$

Amazingly, the resulting list is not only sorted, but sorted much more quickly than if the 'extra' intermediate work had not been undertaken. We'll see more of this later.

8.2 Binary Trees and Tree-sort

You first met binary trees last semester, when they were defined something like this:

```
data Tree a = EmptyT | Node (Tree a) a (Tree a)
```

This says that a tree is either empty, or else is a node with an attached value and two subtrees.

- Sometimes people also define a tree to have a constructor `Leaf a`, but this is unnecessary, because `Leaf x` can already be represented as the tree `Tree EmptyT x EmptyT`.

- In any tree `t = Node left x right`, we call `left` the *left* subtree, and `right` the *right* subtree.
- The tree t is *ordered* if every entry in *left* is smaller than or equal to x, every entry in *right* is greater than x, and the same holds for every subtree of t.

But what does this like as an ADT?

8.2.1 Tree syntax

The basic functions required of a tree depend on who's doing the specifying, but a reasonable syntax might be:

$$\begin{aligned}
createT &: \to Tree\langle Entry\rangle \\
insertT &: Entry \to Tree\langle Entry\rangle \to Tree\langle Entry\rangle \\
mergeT &: Tree\langle Entry\rangle \to Tree\langle Entry\rangle \to Tree\langle Entry\rangle \\
removeT &: Entry \to Tree\langle Entry\rangle \to Tree\langle Entry\rangle \\
emptyT &: Tree\langle Entry\rangle \to Boolean \\
leftT &: Tree\langle Entry\rangle \to Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle \\
rightT &: Tree\langle Entry\rangle \to Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle \\
rootT &: Tree\langle Entry\rangle \to Entry \cup Msg\langle Entry\rangle \\
flattenT &: Tree\langle Entry\rangle \to List\langle Entry\rangle
\end{aligned}$$

- $createT\colon\ \to Tree\langle Entry\rangle$
  creates a new, empty, tree
- $insertT\colon Entry \to Tree\langle Entry\rangle \to Tree\langle Entry\rangle$
  inserts an entry into a tree, and returns the new tree.
- $mergeT\colon Tree\langle Entry\rangle \to Tree\langle Entry\rangle \to Tree\langle Entry\rangle$
  merghes two trees and returns the result.
- $removeT\colon Entry \to Tree\langle Entry\rangle \to Tree\langle Entry\rangle$
  removes an entry from a tree and returns the resulting tree; if the entry isn't present in the tree to start with, returns the original tree unchanged.
- $emptyT\colon Tree\langle Entry\rangle \to Boolean$
  returns *True* if the tree is empty, and *False* otherwise.
- $leftT\colon Tree\langle Entry\rangle \to Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle$
  returns the left subtree of the tree; if the tree is empty, returns the empty tree.
- $rightT\colon Tree\langle Entry\rangle \to Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle$
  returns the right subtree of the tree; if the tree is empty, returns the empty tree.
- $rootT\colon Tree\langle Entry\rangle \to Entry \cup Msg\langle Entry\rangle$
  returns the value at the node of the tree; if the tree is empty, returns an error message.
- $flattenT\colon Tree\langle Entry\rangle \to List\langle Entry\rangle$
  returns the entries from the tree in a list.

Notice that none of the information above tells you anything about the tree-ordering. This information is provided in the semantics (different tree orderings correspond to different axioms). Notice also that the syntax above assumes that $List\langle Entry\rangle$ is also defined. If we change the syntax of $List\langle Entry\rangle$, we will have to change the semantics of $Tree\langle Entry\rangle$ to match.

8.2.2 Tree semantics

The constructors in the list above are *createT*, *insertT* and *mergeT*, so the semantic axioms should tell us what happens when each of the other functions is applied following one of these two. Note however that *mergeT s t* can be defined by recursively calling *insertT* on the elements of s, so no separate semantics are required.

$$\begin{array}{lcl}
mergeT\ (createT)\ t & = & t \\
mergeT\ (insertT\ x\ s)\ t & = & insertT\ x\ (mergeT\ s\ t) \\
mergeT\ (mergeT\ s\ t)\ u & = & mergeT\ s\ (mergeT\ t\ u)
\end{array}$$

$$\begin{array}{lcll}
removeT\ y\ (createT) & = & createT & \\
removeT\ y\ (insertT\ x\ t) & = & \text{if} & x == y \\
 & & \text{then} & t \\
 & & \text{else} & insertT\ x\ (removeT\ y\ t)
\end{array}$$

$$\begin{array}{lcl}
emptyT\ (createT) & = & True \\
emptyT\ (insertT\ x\ t) & = & False
\end{array}$$

$$\begin{array}{lcll}
leftT\ (createT) & = & createT & \\
leftT\ (insertT\ x\ t) & = & \text{if} & x \leqslant rootT\ t \\
 & & \text{then} & insertT\ x\ (leftT\ t) \\
 & & \text{else} & leftT\ t
\end{array}$$

$$\begin{array}{lcll}
rightT\ (createT) & = & createT & \\
rightT\ (insertT\ x\ t) & = & \text{if} & x \leqslant rootT\ t \\
 & & \text{then} & rightT\ t \\
 & & \text{else} & insertT\ x\ (rightT\ t)
\end{array}$$

$$\begin{array}{lcl}
rootT\ (createT) & = & Msg\ \text{"Tree.rootT: tree is empty"} \\
rootT\ (insertT\ x\ createT) & = & x \\
rootT\ (insertT\ x\ t) & = & rootT\ t
\end{array}$$

$$\begin{array}{lcl}
flattenT\ (createT) & = & createL \\
flattenT\ (insertT\ x\ t) & = & sortL\ (pushL\ x\ flattenL)
\end{array}$$

- *removeT*
  If $t \equiv createT$, then t is empty, and we return the empty tree. If $t \equiv insertT\ x\ t'$, we check whether the value y to be removed is equal to x. If so, we simply remove it again and return t. Otherwise, we remove y from t', not forgetting that x needs to be inserted into the result.

- *emptyT*
  A tree is empty if and only if it has the form *createT*.

- *leftT*, *rightT*
  These are the axioms that characterise the tree ordering. They tell us that when *insertT x t* is evaluated, x is inserted into either the left or the right subtree, depending on whether x is smaller than, equal to, or greater than the current root.

- *rootT*
  If the tree is empty, it has no root. If the tree has only one entry, that entry is the root. When values are inserted into an existing nonempty tree, the value is always inserted into one of the two subtrees, so the root value never changes.

- *flattenT*
  When we flatten an empty tree, we get an empty list. When we flatten a tree *insertT* x t, the result is the same as flattening t, adding x, and then sorting the result.

8.2.3 Representation of *Tree*

The observers defined above are *emptyT*, *leftT*, *rightT*, *rootT* and (arguably) *flattenT*. So the items that need to be included in any representation of *Tree* include

Observable	Meaning	Observer	Type
treeRep	The tree itself	n/a	$Container\langle Entry\rangle$
emptyRep	Empty tree	*emptyT*	*Boolean*
leftRep	Left subtree	*leftT*	$Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle$
rightRep	Right subtree	*rightT*	$Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle$
rootRep	Value at root	*rootT*	$Entry \cup Msg\langle Entry\rangle$
flattenRep	List of entries	*flattenT*	$List\langle Entry\rangle$

8.2.4 Tree implementations in Haskell

The only container readily available in Haskell is the list, and there is indeed an easy way to represent binary trees as lists: we put the root of the tree at position 0, and immediate children of the value at position n are placed at $2n + 1$ and $2n + 2$. But implementing functions like *left* and *right*, while not particularly difficult, can nonetheless be messy. A more common solution is to invent our own container type, `TreeImp a`.

A direct implementation of the representation might look like this

```
-- Data structure
data TreeImp a = Tree{
  emptyImp   :: Bool,
  leftImp    :: TreeImp a,
  rightImp   :: TreeImp a,
  rootImp    :: a,
  flattenImp :: [a]
} deriving (Eq, Show)

createImp :: TreeImp a
createImp =  Tree{
  emptyImp   = True,
  leftImp    = createImp,
  rightImp   = createImp,
  rootImp    = error "Tree.rootRep: tree is empty",
  flattenImp = []
}
... and so on ...
```

but this contains far more information than is required.

Rather than make observers like `emptyRep` and `flattenRep` *members* of the type, we can define them as functions instead. Moreover, we can represent the *TreeRep* constructors as constructors in the sense of algebraic data types. This gives us the following definition of a binary tree:

```
data BTree a = EmptyBT
```

```
              | InsertBT a (BTree a)
              | MergeBT (BTree a) (BTree a)
   deriving (Eq, Show)
```

Alternatively, we can take those elements of *TreeImp* which are polymorphic in `a` to be part of the data structure. Since these elements are based on the representations of *left* and *right*, this gives us the standard representation of the sorted binary tree:

```
data Tree a = EmptyT | Node (Tree a) a (Tree a)

instance (Eq a) => Eq (Tree a) where
  s == t  =  (flattenT s) == (flattenT t)

instance (Show a) => Show (Tree a) where
  show EmptyT = "*"
  show (Node EmptyT x EmptyT) = show x
  show (Node l x r) =  "{"  ++ show l ++ "} "
                    ++         show x
                    ++ " {" ++ show r ++ "}"

-- constructors
createT :: Tree a
createT = EmptyT

insertT :: Ord a => a -> Tree a -> Tree a
insertT x EmptyT = Node EmptyT x EmptyT
insertT x (Node l y r)
  | x > y    = Node l y (insertT x r)
  | x <= y   = Node (insertT x l) y r

mergeT :: Ord a => Tree a -> Tree a -> Tree a
mergeT EmptyT t = t
mergeT s t = foldr insertT t (flattenT s)

--observers
emptyT :: Tree a -> Bool
emptyT EmptyT = True
emptyT _      = False

leftT :: Tree a -> Tree a
leftT EmptyT = EmptyT
leftT (Node l _ _) = l

rightT :: Tree a -> Tree a
rightT EmptyT = EmptyT
rightT (Node _ _ r) = r

rootT :: Tree a -> a
rootT EmptyT = error "Tree.rootT: tree is empty"
rootT (Node _ x _) = x

flattenT :: Tree a -> [a]
flattenT EmptyT = []
flattenT (Node l x r) = (flattenT l) ++ [x] ++ (flattenT r)
```

```
-- mutator
removeT :: Ord a => a -> Tree a -> Tree a
removeT _ EmptyT = EmptyT
removeT x (Node l y r)
  | x < y     = Node (removeT x l) y r
  | x > y     = Node l y (removeT x r)
  | x == y    = mergeT l r
```

8.2.5 Tree-sort *vs* insertion-sort

Given this implementation of *Tree*, we can construct a (somewhat indirect) tree-based algorithm for sorting lists. First we define a function that converts a list into a tree, and then we flatten that tree to generate an ordered list.

Because this algorithm does the sorting as it *inserts* data into the tree, it is natural to compare it with a similar algorithm defined directly on lists. One such algorithm is *insertion sort*:

```
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insertL x (isort xs)

insertL :: Ord a => a -> [a] -> [a]
insertL x [] = [x]
insertL x l@(y:ys)
  | x <= y    = x:l
  | otherwise = y: insertL x ys
```

How do the two algorithms compare? Sorting the array $[10, 9, 8, \cdots, 1]$ into ascending order is noticeably faster using `treesort`, and the difference becomes substantial as the list grows longer:

```
Main>     treesort [10,9..1] ---> (    642 reductions,    1160 cells)
Main>        isort [10,9..1] ---> (    723 reductions,    1135 cells)

Main>   treesort [100,99..1] ---> (  37407 reductions,   69426 cells)
Main>      isort [100,99..1] ---> (  51753 reductions,   78221 cells)

Main> treesort [1000,999..1] ---> (3523557 reductions, 6544477 cells)
Main>    isort [1000,999..1] ---> (5017053 reductions, 7532472 cells)
```

8.3 Heaps, heap-sort, and priority queues

Basic queues are simply FIFO buffers, which emit values in the same order they're received. Queues are used a great deal in computer science, but sometimes we need a bit more power.

- An operating system often maintains a queue of processes to help its scheduling, taking account of *priority.* Those with very low priorities may be regarded as background processes, while those with high priorities are given majority access to resources. A structure which acts like a queue, except that priorities are taken account, is called a ***priority queue***, and can be surprisingly tricky to implement.
- To help with the implementation, we use a structure known as a ***heap***.

A heap is like an ordered subtree, except that the ordering constraint is less stringent.

- Ordered trees require the root of each subtree to lie strictly between the values in the left and right subtrees.
- In a heap, we only require the value at the root to be bigger than or equal to every entry in either subtree.

Important. The root of the tree is *always* the largest value. Consequently, we can always access the largest value in just one operation, and do not need to traverse the tree to do so.

Note. If we want to order entries in ascending order (so that those with lower priority are accessed first), we

- either use a heap with the condition the other way up, so that the root is always the smallest entry in the tree.
- or reverse the ordering on the *Entry* data type

8.3.1 Heap syntax and semantics

The basic functions required of a heap depend on who's doing the specifying, but a reasonable syntax might be:

$$\begin{aligned}
\mathit{createH} &: \ \to \mathit{Heap}\langle \mathit{Entry}\rangle \\
\mathit{insertH} &: \mathit{Entry} \to \mathit{Heap}\langle \mathit{Entry}\rangle \to \mathit{Heap}\langle \mathit{Entry}\rangle \\
\mathit{removeH} &: \mathit{Entry} \to \mathit{Heap}\langle \mathit{Entry}\rangle \to \mathit{Heap}\langle \mathit{Entry}\rangle \\
\mathit{mergeH} &: \mathit{Heap}\langle \mathit{Entry}\rangle \to \mathit{Heap}\langle \mathit{Entry}\rangle \to \mathit{Heap}\langle \mathit{Entry}\rangle \\
\mathit{rootH} &: \mathit{Heap}\langle \mathit{Entry}\rangle \to \mathit{Entry} \cup \mathit{Msg}\langle \mathit{Entry}\rangle \\
\mathit{flattenH} &: \mathit{Heap}\langle \mathit{Entry}\rangle \to \mathit{Heap}\langle \mathit{Entry}\rangle
\end{aligned}$$

This simplified syntax enables us to use the following simplified implementation, where (as usual) we've used the *error* function as a means to take $Msg\langle a\rangle \equiv a$, and hence $a \cup Msg\langle a\rangle \equiv (a \cup a) \equiv a$.

```
data Heap a = EmptyH | NodeH (Heap a) a (Heap a)
  deriving (Eq, Show)

createH :: Heap a
createH = EmptyH

insertH :: a -> Heap a -> Heap a
insertH x EmptyH = NodeH EmptyH x EmptyH
insertH x h@(NodeH l y r)
  | x > y     = NodeH h x EmptyH
  | otherwise = NodeH (insert x l) y r

removeH :: a -> Heap a -> Heap a
removeH _ EmptyH = EmptyH
removeH x h@(NodeH l y r)
  | x == y    = mergeH l r
  | otherwise = h
```

```
mergeH :: Heap a -> Heap a -> Heap a
mergeH EmptyH h = h
mergeH h EmptyH = h
mergeH h@(NodeH l x r) h'@(NodeH l' x' r')
  | x <= x'   = NodeH (mergeH h l') x' r'
  | otherwise = NodeH l x (mergeH r h')

rootH :: Heap a -> a
rootH EmptyH = error "Heap.rootH: empty heap"
rootH (NodeH _ x _) = x

flattenH :: Heap a -> Tree a
flattenH EmptyH = []
flattenH (Node l x r) = x : flattenH (mergeH l r)
```

We saw above that *tree sort* is better than *insertion sort*, and now we can see that *heap sort* is even better:

```
Main> heapsort     [10,9..1] ---> (    351 reductions,     608 cells)
Main> treesort     [10,9..1] ---> (    642 reductions,    1160 cells)
Main>    isort     [10,9..1] ---> (    723 reductions,    1135 cells)

Main> heapsort   [100,99..1] ---> (   3051 reductions,    5469 cells)
Main> treesort   [100,99..1] ---> (  37407 reductions,   69426 cells)
Main>    isort   [100,99..1] ---> (  51753 reductions,   78221 cells)

Main> heapsort [1000,999..1] ---> (  30051 reductions,   54970 cells)
Main> treesort [1000,999..1] ---> (3523557 reductions, 6544477 cells)
Main>    isort [1000,999..1] ---> (5017053 reductions, 7532472 cells)
```

8.3.2 Implementation of a priority queue

The `Heap a` type uses the natural ordering on `a` in the definitions of `insertH` and `flattenH`.

If we want to use heaps to implement a priority queue, we need to define an ordering on the queue's elements.

Writing e for the entry and p for its priority, we can define

```
data PQEntry e p = PQEntry {
  entry :: e,
  priority :: p
} deriving (Show)

instance (Eq p) => Eq (PQEntry e p) where
  (PQEntry _ y) == (PQEntry _ y')  =  (y == y')

instance (Ord p) => Ord (PQEntry e p) where
  (PQEntry _ y) <= (PQEntry _ y')  =  (y <= y')

type PQueue e p = Heap (PQEntry e p)
```

8.3.3 A scheduler

```
type ProcName = String
type Priority = Int
type Process  = PQEntry ProcName Priority
type Schedule = PQueue ProcName Priority
```

To see that this works, suppose we have three processes (*Explorer*, *Windows* and *Notepad*) with associated priorities 9, 4 and 9. If we place them into the scheduler, the order in which they are executed is given by flattening the resulting heap.

On defining

```
proc1 = PQEntry "Explorer"  9 :: Process
proc2 = PQEntry "Windows"   4 :: Process
proc3 = PQEntry "Notepad"   9 :: Process
schedule :: Schedule
schedule = insertH proc3 (insertH proc2 (insertH proc1 createH))
```

we get

```
Main> flattenH schedule  --->
     [PQEntry{entry="Explorer",priority=9},
      PQEntry{entry="Notepad",priority=9},
      PQEntry{entry="Windows",priority=4}]
```

as required.

8.4 Summary

This week we have studied two related ADTs in detail, the ADTs of *ordered trees* and *heaps*.

- While adding an entry to a tree can require the entire tree to be re-sorted, the less stringent requirements on heaps mean that less work is involved.
- Consequently, it takes less work, on average, to convert a list into a heap than into a tree.
- Surprisingly, using these ADTs as intermediate structures enables us to define relatively fast sorting algorithms.
- In particular that *heap sort* is much faster than *tree sort*, which is much faster than the list-based *insertion sort*.
- We also noted the role of heaps in implementing *priority queues*, where the heap structure helps to ensure that accessing the highest-priority entry can always be achieved in just one operation.
- Using priority queues can help operating systems to schedule processes more efficiently.

Chapter 9

Specification, representation and implementation

9.1 Introduction

The code I've supplied is based in part on code described in Thompson's text book [Tho99, §17.5]. Advanced programmers may like to know that it is also possible to write parsers using ***monad*** constructions; monads are mathematical structures whose properties make them excellent tools for functional programming; they are used in Haskell to handle input/output operations. See the Haskell tutorial [HPF99], Hutton & Meijer's papers [HM96, HM98], or Thompson's book [Tho99] for more details.

9.2 Basic Parsing

A simple ***parser*** examines a list of symbols, and attempts to interpret as much of the list as possible as a meaningful 'chunk', usually called a ***token***. The rest of the symbols in the list can then be supplied as input to further parsers; by working together, one after another, the various parsers will (if the symbol stream is valid) eventually converts the entire list of symbols into a list of tokens. If we're lucky (in other words, if the language the tokens belong to is defined sensibly) there will be one and only one way to split the input stream into a list of tokens, but we need to take account of situations where this isn't the case. Consequently, a typical basic parser has the following type

```
type Parse s t = [s] -> [(t,[s])]
```

where `s` is the symbol type and `t` is the token type. Given a string of symbols, the parser returns a list of possible tokens, and in each case it also returns the unprocessed section of the input stream. If there are *no* valid parsings of the input stream, it simply returns the empty list `[]`.

Example: Parsing integers from a string of characters

Suppose we want to parse a stream of characters in order to extract an integer. In this case the symbols are of type `Char` and the tokens are of type `Int`, so the parser is of type `Parse Char Int`.

Example: Recognising words and sentences

Suppose we want to break a block of text into sentences. First we group the characters of the text into words and punctuation. Then we group the words and punctuation into sentences. This involves two stages of parsing; the first uses a parser of type `Parse Char String`, the second a parser of type `Parse String [String]`.

9.2.1 Basic parsing components

As in so much of computer science, the strategy we use when parsing input streams is straightforward: if we need to parse a difficult symbol stream, we do it by joining together lots of simpler parsers. There are several very basic parsers that feature strongly in what follows.

Match nothing

The `matchNone` parser is designed to fail no matter what input it encounters.

```
matchNone :: Parse s t
matchNone inp = []
```

Insert a token into the token stream

The `matchAndAdd` parser simply adds a token to the token stream without doing anything to the input stream. This parser plays very little part in our solution; it is used routinely in the 'monadic' approach to parsing, where it is more usually known by the name `result` [HM96].

```
matchAndAdd :: t -> Parse s t
matchAndAdd val inp = [(val,inp)]
```

Match the next symbol if it satisfies a certain property

The `matchProperty` parser is particularly useful. It declares a symbol to be a token in its own right, provided it satisfies a given property.

```
matchProperty :: (s -> Bool) -> Parse s s
matchProperty p (x:xs)
  | p x           = [(x,xs)]
  | otherwise     = []
matchProperty _ [] = []
```

For example, the Prelude function `isDigit :: Char -> Bool` recognises whether a character is in the list `['0'..'9']`. Consequently, the parser

```
matchProperty isDigit :: Parse Char Char
```

recognises whether or not the next character in the input stream is a digit. Related functions like `isAlpha`, `isUpper` and `isLower` can be used the same way, allowing us to define various useful recognisers:

```
matchAny      = matchProperty (\_ -> True) -- matches any symbol
matchAlphaNum = matchProperty isAlphaNum   -- matches alphanumeric characters
matchUpper    = matchProperty isUpper      -- matches uppercase letters
matchLower    = matchProperty isLower      -- matches lowercase letters
matchDigit    = matchProperty isDigit      -- matches individual digits
```

For example,

```
matchProperty isDigit "Hello" ⇝ []
matchProperty isDigit "12345" ⇝ [('1',"2345")]
matchProperty isUpper "Hello" ⇝ [('H',"ello")]
matchProperty isLower "Hello" ⇝ []
matchproperty isLower "hello" ⇝ ['h',"ello"]
```

In each of these examples, at most one character is matched, and in each case, the unprocessed section of the input string is returned as the second component of the pair.

Matching a specific symbol

A special case of the `matchProperty` parser idea is `matchInput`. This matches a specific input symbol.

```
matchInput :: Eq a => a -> Parse a a
matchInput t = matchProperty (t==)
```

We can use this to recognise punctuation, for example

```
matchLP = matchInput '('   -- left parenthesis
matchRP = matchInput ')'   -- right parenthesis
matchCO = matchInput ','   -- comma
matchSP = matchInput ' '   -- space
matchLB = matchInput '['   -- left bracket
matchRB = matchInput ']'   -- right bracket
matchPT = matchInput '.'   -- point (dot)
matchSQ = matchInput '\''  -- single quote
matchDQ = matchInput '"'   -- double quote
matchSL = matchInput '\\'  -- slash
```

9.2.2 Match this or match that

Suppose we want the next symbol to be a letter, but we don't mind if it is upper or lower case. One solution would be to apply `matchProperty` to a suitable test-function, `isAlpha`. Another approach is to apply both `matchUpper` and `matchLower`, and to combine the results. This behaviour is achieved using the `alt` function.

```
alt :: (Eq s, Eq t) => Parse s t -> Parse s t -> Parse s t
alt p1 p2 inp = list2set (p1 inp ++ p2 inp)
```

Notice the constraint `(Eq s, Eq t)`. This isn't normally necessary – I've only had to introduce it because I'm converting the resulting list into a *set* by throwing out repeated solutions, in a bid to increase efficiency. The function `list2set` is defined as

```
list2set :: Eq a => [a] -> [a]
list2set (x:xs)
  | x `elem` xs = list2set xs
  | otherwise   = x : list2set xs
list2set []     = []
```

The most important thing to note about `alt` is that the results of two parsers can only be combined if the parsers share the same type. For example, we can combine `matchLP `alt` matchCO `alt` matchRP` to generate a parser for parentheses and commas, because all three of the components have the same type, `Parse Char Char`. But if one parser returns a string and the other a character, we cannot combine their results directly.

9.2.3 Chaining parsers together

Suppose we run a parser `p1` on some text to obtain a list of possible parsings. Each of these will include a string representing the unprocessed section of the input. We now envisage running a second parser `p2` on each of these unprocessed sections of text. The outcome should tell us whether the input string can satisfy both parsers, one after the other. If it can, we return *both* matches (the match made by `p1` and the one made by `p2`) as a *pair* of matches; as always, we also return the remaining unprocessed part of the input stream. We can achieve this *chaining* of parsers by defining a new operator:

```
infixr 5 >*>

(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp,
                    (z,rem2) <- p2 rem1 ]
```

The first line of this declaration (`infixr 5 >*>`) tells us that the operator `>*>` is to be assigned a *precedence* of 5, and that it is *right-associative*. This means that the expression `p1 >*> p2 >*> p3` is bracketed like this: `p1 >*> (p2 >*> p3)`.

Example: Matching floating point numbers

Suppose `matchInt :: Parse Char String` can recognise character strings like `"123"` that represent non-negative integers (where such a string is defined to be any string of digits). We've already seen that `matchPT` recognises decimal points, so we can recognise floating point numbers by defining

```
matchFPNumber = matchInt >*> matchPT >*> matchInt
```

In other words, we look for a string of digits, then a decimal point, and then another string of digits. How does this actually work?

- First we apply `matchInt` to `"12.34"`. This gives us *two* possible parsings.

  `matchInt "12.34"` $\rightsquigarrow$ `[ ("1", "2.34"), ("12", ".34") ]`

  In each of the two pairs, the parsers has extracted a valid integer (string of digits) into the first component of the pair, and has put the remaining input stream into the second component.

- Next we apply `matchPT >*> matchInt` to each of the unprocessed sections in turn to see what results we get. To do this, we first apply `matchPT`...

```
matchPT "2.34" ⇝ []               -- so ("1","2.34") is rejected
matchPT ".34" ⇝ [('.', "34")]   -- so ("12",".34") is viable
```

- ...and then `matchInt`...

```
matchPT "34" ⇝ [("3","4"), ("34", "")]
```

- Pulling together all the results gives us the following two matches:

```
matchFPNum "12.34" ⇝ [ (("12",('.',"3")), "4"), (("12",('.',"34")), "") ]
```

Let's take a look at the type of this answer. We have

```
                              matchPT :: Parse Char Char
                             matchInt :: Parse Char String
so               matchPT >*> matchInt :: Parse Char (Char, String)
so  matchInt >*> (matchPT >*> matchInt) :: Parse Char (String,(Char,String))
```

and this is reflected in the answers we computed. For example, the result `(("12",('.',"3")), "4")` tells us that the remaining substring `"4"` is as yet unprocessed, while the earlier section of the input stream, `"12.3"`, has been parsed into a string `"12"`, followed by a character `'.'`, followed by another string `"3"`.

9.2.4 Simplifying the results of parsing

We've just seen that simple parsers like `matchInt` and `matchPT` can be chained together to return more complicated structures, like floating point numbers, but the results they return are in a fairly unpleasant format. We can simplify matters by applying the `build` function.

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x, rem) <- p inp ]
```

We use the `build` function to turn the complicated returned matches into something more sensible. For example, if we define

```
buildFloat :: (String,(Char,String)) -> String
buildFloat (s1,(_,s2)) = s1 ++ "." ++ s2
```

then `matchFPNumber` can be simplified dramatically. We can redefine it as extracting tokens of type `String` rather than (as at present) of type `(String,(Char,String))`, by writing

```
matchFloat :: Parse Char String
matchFloat
  = (matchInt >*> matchPT >*> matchInt) `build` buildFloat
      where buildFloat (x,(_,y)) = (x ++ "." ++ y)
```

Notice, however, that there is no restriction on the return type of the 'build' function we decide to use. Rather than return a string representation of a floating point number, we could just as easily return *the floating point number itself* by defining

```
matchFloatAsFloat :: Parse Char Float
matchFloatAsFloat = (matchInt >*> matchPT >*> matchInt)
                    `build` (read . buildFPNumber)
```

This time, we use `matchInt >*> matchPT >*> matchInt` to find two strings of digits separated by a point; then we apply `buildFPNumber` to glue the results together into a single string; and finally, we use the Prelude function `read` to read the string and interpret the result as a value of type `Float`.

9.2.5 Matching lists

We used the function `matchInt` above, but we haven't yet defined it. Essentially, `matchInt` has to move through a *list* of inputs, and check each one against the property `isDigit`. More generally, we can check any list of symbols against a given property by using the `matchList` function:

```
matchList :: (Eq s, Eq t) => Parse s t -> Parse s [t]
matchList p = (matchAndAdd []) `alt`
              ((p >*> matchList p) `build` (uncurry (:)))
```

The resulting parser merges the results from two subparsers, using the `alt` function; remember, this requires constraints on both the symbol and the token types to be in place; both have to be instances of `Eq`. The first subparser, `matchAndAdd []`, says that the empty token-list should always be regarded as a valid parsing of a symbol-stream. The second subparser is more interesting; it says that a *non-empty* list of valid results is found by first identifying one symbol that matches, and then re-applying `matchList` to the remaining unprocessed input stream.

Notice the use of the `build` function. The type signature tells us that `p` returns tokens of type `t`, while `matchList` returns tokens of type `[t]`. As a result, `(p >*> matchList p)` returns matches of type `(t,[t])`. For our definitions to be consistent, we need to 'build' these back into results of type `[t]`. The function is question simply takes the first symbol matched and appends it to the front of the subsequent matches. In other words, we need to use the 'build' function

```
buildList :: (t, [t]) -> [t]
buildList (x, xs) = x : xs
```

Currying and uncurrying

In the definition of `matchList`, I've actually written `uncurry (:)` instead of `buildList`. Why? The reason goes back to the very beginnings of functional programming. We're used to seeing functions like `(+)` and `(*)` defined to have type `Int -> Int -> Int`, but mathematicians more usually think of them as acting on *pairs* of numbers; they would declare these functions to have type `(Int, Int) -> Int` instead. Clearly, the two interpretations are *isomorphic*, in that we can always redefine a function written one way as a function written the other. The procedures for translating from one type signature to the other are called ***currying*** and ***uncurrying***, and are named after *Haskell Curry*, an early pioneer of the *λ-calculus*, on which all functional programming languages are based. In Haskell, these functions are defined in the Prelude as:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p = f (fst p) (snd p)
```

Since `(:)` is a function of type `t -> [t] -> [t]`, we see that its equivalent 'uncurried' form is of type `(t, [t]) -> [t]`, as required.

9.2.6 Matching a non-empty list

We've just seen that `matchList` merges the results of two subparsers, one of which allows for empty token lists, the other of which searches for non-empty lists of tokens. If we specifically want to match against a *non-empty* list of tokens, all satisfying some given property, we simply throw away the first subparser. This gives us the function `matchNEList`:

```
matchNEList :: (Eq a, Eq b) => Parse a b -> Parse a [b]
matchNEList p = (p >*> matchList p) `build` (uncurry (:))
```

`matchList` and `matchNEList` are important functions, and occur frequently in our code. For example, we can match a non-empty string of alphanumeric characters (`matchAlphaNums`), or a non-empty string of digits (`matchInt`):

```
matchAlphaNums :: Parse Char String
matchAlphaNums = matchList matchAlphaNum

matchInt :: Parse Char String
matchInt = matchNEList matchDigit
```

9.2.7 Matching a given string

Instead of matching all the symbols in a list against the *same* property, we can match them against a *list* of properties, one for each symbol. In particular, we can use this idea to look for specific non-empty strings of symbols, by defining the `matchInputs` function:

```
matchInputs :: Eq a => [a] -> Parse a [a]
matchInputs []     = error "matchInputs: Empty test string"
matchInputs [x]    = matchInput x `build` buildMatch1
                     where buildMatch1 x = [x]
matchInputs (x:xs) = (matchInput x >*> matchInputs xs) `build` (uncurry (:))
                     where buildMatch2 (x,y) = [x,y]
```

9.2.8 Some specific tokens of interest

Whitespace

We define whitespace to be any string of spaces. We can match these by defining:

```
matchWhiteSpace :: Parse Char String
matchWhiteSpace = matchList matchSP
```

Function arrows

The function arrow '`->`' occurs frequently in Haskell type definitions, and is usually surrounded by whitespace. We therefore define:

```
matchArrow = (matchWhiteSpace >*> matchInputs "->" >*> matchWhiteSpace)
             `build` buildArrow
  where buildArrow _ = "->"
```

Matching characters (identified by single quotes)

In Haskell, a *character* can be recognised by the presence of surrounding single quotes. In general we'll ignore the possibility of control characters and escape codes; recognising a character simply involves matching an opening single quote, the character itself, and the closing sungle quote. There is, however, one exception to this rule. Suppose the character in question is itself a single quote. To represent a single quote as a character in Haskell we have to write `\'` inside single quotes, and in order to express *this* pair of characters as a string (which we need to do if we're to use `matchInputs`), we have to write `"\\'"`. Consequently, the function `matchChar` has the following definition:

```
matchChar = (matchSQ >*>
             ((matchInputs "\\'" ) `alt`
              ((matchProperty ('\'' /=)) `build` char2str))
             >*> matchSQ)
            `build` buildChar
  where buildChar (_,(s,_)) = "'" ++ s ++ "'"
```

The function `char2str` converts any character, e.g. `'a'`, into a one-character string, e.g. `"a"`. It is defined as

```
char2str :: Char -> String
char2str x = [x]
```

Matching strings (identified by double quotes)

In Haskell, a *string* is identified by enclosing double quotes. As when matching characters, the only complication arises when the string contains an instance of its own delimiter, in this case a double quote. To represent a double quote as a character, we write \". To represent this pair of characters as a string, we have to 'escape' both the backslash and the quote; this gives us the rather awkward string \\\" to match against.

```
matchString :: Parse Char String
matchString = (matchDQ >*>
               matchList ((matchInputs "\\\"" ) `alt`
               ((matchProperty ('"' /=)) `build` char2str))
               >*> matchDQ)
              `build` buildString
  where buildString (_,(ss,_)) = "\"" ++ (foldr (++) "" ss) ++ "\""
```

Matching booleans

Since there are only two Boolean values in Haskell, we can easily parse for them by defining:

```
matchBool :: Parse Char String
matchBool = (matchInputs "True") `alt` (matchInputs "False")
```

9.3 Parsing Types

Now that we know *how* to parse input streams, we have to consider *what* we'll be parsing. In this case, we want to parse an input string and interpret it as including various 'meaningful chunks' representing *types* and *expressions*. The sorts of token we'll be looking for could include

- Types: `Int`, `Bool`, `Int -> Bool`, `(Int,Bool)`
- Expressions: `5`, `True`, `isZero`, `(5, False)`
- Declarations: `5::Int`, `isZero::Int -> Bool`, `(5,False)::6(Int,Bool)`

We'll consider the parsing of expressions and declarations below; for now we'll concentrate on parsing types.

9.3.1 The datatype `Type`

The first thing we need to do is define a data type representing the sort of structure we expect types to possess. I've defined

```
type TypeName = String
data Type     = NullT
              | UnaryT TypeName
              | ListT Type
              | TupleT [Type]
              | FuncT Type Type
```

The various constructors used in this type are intended to have the following meanings.

NullT

The *null type* is used to represent the result of invalid parsing. Rather than return an error message, I prefer to return a valid result. I'll represent the null type as an empty tuple-type, `( )`.

UnaryT

A *unary type* is a basic type invoked simply by giving its name. Examples include `Int` and `Bool`, which would be represented in my program as `UnaryT "Int"` and `UnaryT "Bool"`, respectively.

ListT

A *list type* is any type of the form `[t]` where `t` is a type. For example, `ListT (UnaryT "Char")` represents the familiar Haskell type `[Char]`.

TupleT

A *tuple type* is any type that is expressed as a tuple `(t1, ..., tn)` of other types. For example, the Haskell type `(Int,Bool)` is represented in my program as `TupleT [UnaryT "Int", UnaryT "Bool"]`.

FuncT

A *function type* is any type of the form `t1 -> t2`, where `t1` and `t2` are types. For example, `FuncT (UnaryT "Int") (UnaryT "Bool")` represents the Haskell type `Int -> Bool`.

In Haskell, there are two other sorts of type; the assignment asks you to implement them. These are *polymorphic* and *constrained* types.

Polymorphic types

These are types like `Either a b` and `Queue a` that include both a type name and a list of type variables.

Constrained types

These are types like `(Eq a, Ord a) => a -> Bool` which include a list of classes to which one or more specified type variables must belong.

9.3.2 Parsing types

Our aim in this section is to parse strings that represent types into the types themselves. Our strategy is straightforward. First we will define parsers for each of the different sorts of type, and then merge them using the `alt` function:

```
type TypeParser = Parse Char Type

matchType :: TypeParser
matchType = matchUnaryT `alt` matchListT `alt` matchTupleT
                        `alt` matchFuncT `alt` matchNullT
```

Don't forget – for this strategy to work, we need to ensure that each of the component parsers (`matchUnaryT`, `matchTupleT`, and so on) all share the same type, `TypeParser`. We will use suitable 'build' functions to ensure that this is the case.

Ultimately, all (useful) types in my program are constructed from unary types; since these take the form `UnaryT typename` where `typename` is of type `TypeName`, the first thing we need is a parser for typenames. Recall that valid typenames comprise an uppercase letter followed by a string of alphanumeric characters. Accordingly, we can define

```
type TypeNameParser = Parse Char TypeName

matchTypeName :: TypeNameParser
matchTypeName = (matchUpper >*> matchAlphaNums) `build` (uncurry (:))
```

Having defined a parser for typenames we can now define a parser for unary types.

```
matchUnaryT :: TypeParser
matchUnaryT str = [ (UnaryT s, rem) | (s,rem) <- matchTypeName str ]
```

The other sorts of type can be parsed just as easily:

```
matchNullT :: TypeParser
matchNullT = (matchLP >*> matchWhiteSpace >*> matchRP)
             `build` (\_ -> NullT)

matchListT = (matchLB >*> matchWhiteSpace >*>
              matchType >*> matchWhiteSpace >*> matchRB)
             `build` buildListT
  where buildListT (_,(_,(t,(_,_)))) = ListT t

matchTupleT = (matchLP >*> matchType >*>
               matchRepeatingBlock >*> matchRP)
            `build` buildTupleT
 where
  matchBlock = matchCO >*> matchWhiteSpace >*> matchType
  matchRepeatingBlock = matchList matchBlock
  buildTupleT (_,(t,(blocks,_))) = normalise typ where
    typ = TupleT (t : (map getTypes blocks))
    getTypes (_,(_,x)) = x

matchFuncT :: TypeParser
matchFuncT = matchFuncNoBracketsT `alt` matchFuncInBracketsT

matchFuncInBracketsT :: TypeParser
matchFuncInBracketsT = (matchLP >*> matchFuncT >*> matchRP)
                       `build` buildFuncIBT
  where buildFuncIBT (_,(t,_)) = normalise t

matchFuncNoBracketsT :: TypeParser
matchFuncNoBracketsT = (
  (matchUnaryT `alt` matchTupleT `alt`
             matchFuncInBracketsT `alt` matchNullT)
  >*> matchArrow >*> matchType
  ) `build` buildFuncNBT
 where buildFuncNBT (t1,(_,t2)) = normalise (FuncT t1 t2)
```

Notice that `matchTupleT`, `matchFuncInBracketsT` and `matchFuncNoBracketsT` all involve the function `normalise`. This is a function that recognises that the same type can be written in various different ways. For example, `NullT->t` and `TupleT [t]` are both the same type as `t`, no matter what type `t` happens to be. The `normalise` function simplifies types to give the simplest equivalent representation. It is defined as

```
normalise :: Type -> Type
normalise (TupleT []) = NullT
normalise (TupleT [t]) = normalise t
normalise (FuncT _ NullT) = NullT
normalise (FuncT NullT t) = normalise t
normalise (ListT t) = ListT (normalise t)
normalise t = t
```

9.4 Parsing expressions

Exactly the same procedures apply to the parsing of expressions. First we define a datatype representing the different sorts of expression that can be written in Haskell, and then we write parsers for each sort. My code defines a type called `Expr`, and its associated parser type, `ExprParser`, as follows:

```
type FuncName = String
data Expr = NullE
          | FuncE FuncName
          | IntE Int
          | FloatE Float
          | CharE Char
          | StringE String
          | BoolE Bool
          | ListE [Expr]
          | TupleE [Expr]
          | ApplyE [Expr]
type ExprParser = Parse Char Expr
```

NullE

The *null expression* doesn't play an active role in my code; it's used to represent errors in the expression-parsing process.

FuncE

A *function expression* is simply an expression comprising a function name. For example, `FuncE "map"` represents an occurrence of Haskell function `map` as an expression.

IntE

An *integer expression* is an expression representing an integer. The Haskell integer `5` is represented in my program as `IntE 5`.

FloatE

A *float expression* is an expression representing a floating point number. The Haskell value `1.23` is represented in my program as `FloatE 1.23`.

CharE

A *character expression* is an expression representing a character. The Haskell character `'a'` is represented in my program as `CharE 'a'`.

StringE

A *string expression* is an expression representing a string. The Haskell string `"Hello"` is represented in my program as `StringE "Hello"`.

BoolE

A *boolean expression* is an expression representing a Boolean value. The Haskell values `True` and `False` are represented in my program as `BoolE True` and `BoolE False`, respectively.

ListE

A *list expression* is an expression representing a list. The Haskell list `[1,2]` is represented in my program as `ListE [IntE 1,IntE 2]`. My code does *not* currently check that all entries in the list are of the same type.

TupleE

A *tuple expression* is an expression representing a tuple. The Haskell tuple `(1,True)` is represented in my program as `TupleE [IntE 1,BoolE True]`.

ApplyE

An *application expression* is an expression representing the results of applying one expression to another. For example, suppose we apply the function `double` to the value `5`. In Haskell this is represented as the expression `double 5`, and in my program it appears as `ApplyE [FuncE "double", IntE 5]`. Likewise, the Haskell expression `map f [1,2]` would be written `ApplyE [FuncE "map",FuncE "f", ListE [IntE 1,IntE 2]]` in my system.

Haskell supports one more type of expression, the *abstraction* or *anonymous function*, which is not included in my program. The assignment asks you to add the relevant code.

Abstraction expressions

Anonymous functions (aka abstractions) are expressions like `(\x -> double x)`, which depend upon a variable (in this case `x`) and an expression (in this case `double x`).

9.4.1 Parsing expressions

As with types, we parse expressions by merging the results of many individual parsers, one for each sort of expression.

```
matchExpr :: ExprParser
matchExpr = matchNotNullE `alt` matchNullE

matchNotNullE :: ExprParser
matchNotNullE = matchNotNullOrApplyE `alt` matchApplyE

matchNotNullOrApplyE :: ExprParser
matchNotNullOrApplyE
  = matchFuncE `alt` matchIntE `alt` matchFloatE `alt`
    matchCharE `alt` matchStringE `alt` matchBoolE `alt`
    matchListE `alt` matchTupleE

matchNullE :: ExprParser
```

```
matchNullE = (matchLP >*> matchWhiteSpace >*> matchRP)
           `build` (\_ -> NullE)

matchFuncE :: ExprParser
matchFuncE = ((matchLower >*> matchAlphaNums)
           `build` (FuncE . (uncurry (:))))

matchIntE :: ExprParser
matchIntE = matchInt `build` (IntE . read)

matchFloatE :: ExprParser
matchFloatE = matchFloat `build` (FloatE . read)

matchCharE :: ExprParser
matchCharE = matchChar `build` (CharE . read)

matchStringE :: ExprParser
matchStringE = matchString `build` (StringE . read)

matchBoolE :: ExprParser
matchBoolE = matchBool `build` (BoolE . read)

matchListE = (matchLB    >*> matchWhiteSpace
          >*> matchExpr  >*> matchWhiteSpace
          >*> matchBlock >*> matchRB)
          `build` buildListE
 where
  buildListE :: (Char,([Char],(Expr,([Char],([Expr],Char))))) -> Expr
  buildListE (_,(_,(e,(_,(es,_))))) = ListE (e:es)
  matchBlock = matchList matchComponent
  matchComponent = (matchCO   >*> matchWhiteSpace
                >*> matchExpr >*> matchWhiteSpace)
     `build` buildComponent
  buildComponent (_,(_,(e,_))) = e

matchTupleE = (matchLP    >*> matchWhiteSpace
          >*> matchExpr  >*> matchWhiteSpace
          >*> matchBlock >*> matchRP)
          `build` buildTupleE
 where
  buildTupleE (_,(_,(e,(_,(es,_))))) = TupleE (e:es)
  matchBlock = matchList matchComponent
  matchComponent = (matchCO   >*> matchWhiteSpace
                >*> matchExpr >*> matchWhiteSpace) `build` buildComponent
  buildComponent ::  (Char,(String,(Expr,String))) -> Expr
  buildComponent (_,(_,(e,_))) = e

matchApplyE = (matchNotNullOrApplyE >*> matchRepeatingBlock)
  `build` buildApplyE
 where
  matchRepeatingBlock
    = matchList (matchSP >*> matchWhiteSpace >*> matchExpr)
  buildApplyE (e,xs) = ApplyE (e : map third xs)
  third :: (Char,(String,Expr)) -> Expr
```

```
third (_,(_,e)) = e
```

9.5 Declaring types of user-specified functions

Some of the expressions described above (for example, integers and Boolean values) are built-in to Haskell, but in many cases the user has to *declare* the types of functions. For example, if we introduce a function called `double`, it is reasonable to require us to declare that `double` has type `Int->Int`. I achieve the same result in my program by requiring the user to declare a *type library*, containing information about the types of every non-standard function used in their code. For the purposes of this exercise, the functions declared in the Prelude should also be included; here's the simple version of the type library I've been using during development:

```
type TypeLib = [Declaration]
typeLib :: TypeLib

-- EXAMPLE: A system with 3 user-defined functions
typeLib = map declare [
  "double :: Int -> Int",
  "fst :: (Int, Int) -> Int",
  "snd :: (Int, Int) -> Int"
  ]
```

The type `Declaration` used in this definition represents declarations of the form `e::t`. As always, we include a *null declaration* to allow for invalid attempts at parsing. To help with testing, I've also introduced a function called `declare` that takes a string and tries to parse it as a declaration, as well as two functions for extracting the expression part, and the type part, from the declaration.

```
data Declaration = NullD | Declare Expr Type

type DeclarationParser = Parse Char Declaration
matchDeclaration = (matchExpr >*> matchWhiteSpace >*>
                    matchInputs "::" >*> matchWhiteSpace >*>
                    matchType)
                  `build` buildDeclaration
  where buildDeclaration (e,(_,(_,(_,t)))) = Declare e t

declare :: String -> Declaration
declare str
  | null ds   = NullD
  | otherwise = head ds
  where ds = [ d | (d,"") <- matchDeclaration str ]

dec2expr :: Declaration -> Expr
dec2expr NullD = NullE
dec2expr (Declare e _) = e

dec2type :: Declaration -> Type
dec2type NullD = NullT
dec2type (Declare _ t) = t
```

9.6 Type evaluation

If we know the types of `f`, `g` and `h`, we should be able to work out for ourselves what the type is of the result of evaluating the expression `(f g h)`. This is routinely carried out as part of Haskell's *type evaluation* procedure, and can be mirrored in our system by the following code

```
doApplicationT :: Type -> Type -> Type
doApplicationT t1 t2
  | t2 == getFst t1  = getSnd t1
  | otherwise        = NullT
 where getFst (FuncT x _) = x
       getFst _ = NullT
       getSnd (FuncT _ y) = y
       getSnd _ = NullT
```

More generally, suppose someone gives us an expression, and asks us to determine it type. If the expression is simply a function name, we need to look it up in the type library; in all other cases we should be able to work out the type for ourselves. Here's my code for doing this:

```
getTypeFromLib :: FuncName -> Type
getTypeFromLib f
 | null ts = NullT
 | otherwise = head ts
 where ts = [ dec2type d | d <- typeLib, dec2expr d == (FuncE f) ]

-- Get the type of a specified expression
getType :: Expr -> Type
getType NullE = NullT
getType (FuncE f) = getTypeFromLib f
getType (IntE _) = UnaryT "Int"
getType (FloatE _) = UnaryT "Float"
getType (CharE _) = UnaryT "Char"
getType (StringE _) = UnaryT "String"
getType (BoolE _) = UnaryT "Bool"
getType (ListE []) = NullT
getType (ListE (e:es)) = ListT (getType e)
getType (TupleE []) = NullT
getType (TupleE [e]) = getType e
getType (TupleE es) = TupleT (map getType es)
getType (ApplyE []) = NullT
getType (ApplyE [e]) = getType e
getType (ApplyE (e:es))
  = foldl doApplicationT (getType e) (map getType es)
```

9.7 Simplifying the testing process

Finally, how can we simplify the procedures involved in testing the code? One easy approach is to write functions that perform the same sorts of function that we'd otherwise need to carry out by hand. I've introduced four such functions:

- `typeOf` takes a string representing an expression and returns its type. For example, `typeOf "5"` returns `UnaryT "Int"`

- `parse` does the same thing, but returns the result as a declaration. For example, `parse "5"` returns `IntE 5 :: UnaryT "Int"`
- `asType` takes a string and tries to interpret the *whole* of that string as a valid type, rather than just some initial substring. For example, `"Int->Bool"` will be parsed successfully, but `"Int Int"` won't be, even though it starts with the substring `"Int"`.
- `asExpr` takes a string and tries to interpret the *whole* of that string as a valid expression, rather than just some initial substring. For example, `"(5,True)"` will be parsed successfully, but `"24 7"` won't be, even though it starts with the substring `"24"`.

```
typeOf :: String -> Type
typeOf = getType . asExpr

asType :: String -> Type
asType str
  | null ts   = NullT
  | otherwise = head ts
  where ts = [ t | (t,"") <- matchType str ]

asExpr :: String -> Expr
asExpr str
  | null es   = NullE
  | otherwise = head es
  where es = [ e | (e,"") <- matchExpr str ]

parse :: String -> Declaration
parse str = Declare e t
  where e = asExpr str
        t = getType e
```

9.8 Summary

This week the focus has been on *parsing*, but we have also explored techniques for representing types and expressions, and for evaluating the types *of* expressions.

Appendix A

Sample Problem Sheets

COM2001 (Spring Semester): Problems, Week 2

These problems are optional; they do not contribute to your coursework assessment

Problem 2.1

Show in detail how Haskell evaluates `fact 3`, where `fact` is defined:

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

Problem 2.2

Show in detail how Haskell evaluates `summ [1,2,3]`, where `summ` is defined:

```
summ :: [Int] -> Int
summ [] = 0
summ (x:xs) = x + summ xs
```

Problem 2.3

Show in detail how Haskell evaluates `diff [1,2,3]`, where `diff` is defined:

```
diff :: [Int] -> Int
diff [] = 0
diff (x:xs) = (diff xs) - x
```

Problem 2.4

Is it true, for all lists **xs** of type **[Int]**, that

```
(summ xs) + (diff xs) == 0
```

If yes, prove it. If no, give a counter-example.

Problem 2.5

What is the type of `compn 2 (+3) (*2)`, where `compn` is defined:

```
compn :: Int -> (a -> a) -> (a -> a) -> a -> a
compn 0 f g x = f (g x)
compn n f g x = f (compn (n-1) f g x)
```

Problem 2.6

Show in detail how Haskell evaluates `alt (+) (-) [1,2,3,4]`, where `alt` is defined:

```
alt :: (a -> a -> a) -> (a -> a -> a) -> [a] -> [a]
alt f g (x:y:ys) = (f x y) : alt g f (y:ys)
alt f g _        = []
```

END OF PROBLEM SHEET

COM2001 (Spring Semester): Problems, Week 3

These problems are optional; they do not contribute to your coursework assessment

This week's problem sheet is designed to give you some practical experience designing an ADT in Haskell. We will assume that the implementation avoids the need to generate specific error values by using the Haskell **error** function wherever necessary. Wherever the problems below say to 'write something down', this means you should write down the relevant code in a Haskell program file. After writing stuff down, check that your file compiles properly, and run a few tests to make sure that your functions work properly.

Don't worry if you can't work out how to implement everything you're asked for. You can still gain valuable experience simply by *trying* to implement things. Some of the tasks suggested below are quite tricky!

Please note that I've included explanatory text between the various problems. The end of each problem is marked by the following symbol: ✓

Let's imagine that the *list* type, `[a]`, had never been defined in Haskell, and that we need to design our `List a` data type. To stop things being too simple, I'll suppose that the data structure is defined as follows:

```
data List a = EmptyList              -- the empty list
            | Singleton a            -- a list with just one element of type a
            | Join (List a) (List a) -- one list followed by another
     deriving (Show, Eq)
```

Problem 3.7

Write down two different ways of constructing the List containing the values 0, 1 and 2. Does HUGS regard these lists as equal? If not, can you explain what the problem is? ✓

As with all lists, there are certain basic functions that need to be made available (to avoid clashes with the standard list functions, I've changed the names of the functions slightly). For example, we need a function **pushL** which pushes an entry onto the front of a list (this is the equivalent of `x : xs` in the standard implementation of lists). This can be defined to have the type

```
pushL :: a -> List a -> List a
pushL x EmptyList = Singleton x
pushL x xs        = Join (Singleton x) xs
```

Problem 3.8

Write down the types of the following functions:

- `lengthL`, returns the length of the list
- `itemAtL`, returns the entry at the given index (the first index is 0)
- `nullL`, returns a boolean telling us if the list is empty

- `headL`, returns the entry at the head of the list
- `tailL`, returns the tail of the list
- `concatL`, joins two lists together (this is the equiavlent of `xs ++ ys`).
- `reverseL`, reverses a list

✓

The definition of **List a** says that the type has three constructors: `EmptyList`, `Singleton` and `Join`. We need to give the semantics for each function we define, by describing its effect on each of these constructors. For example

```
pushL x EmptyList     = Singleton x
pushL x (Singleton y) = Join (Singleton x) (Singleton y)
pushL x (Join xs ys)  = Join (Singleton x) (Join xs ys)
```

In this example, the last two cases can be written as one, *viz.*,

```
pushL x EmptyList = Singleton x
pushL x xs        = Join (Singleton x) xs
```

but this may not always happen – it depends on the function whose semantics you're defining.

Problem 3.9

Implement each of the functions above. ✓

We saw above that Haskell's default mechanism for defining equality doesn't work for this data type. One solution is to write a function

```
normalise :: List a -> List a
```

that takes any list and re-writes it in some standard format. To check whether two lists are equal, we first normalise them, and then check whether the normalised versions are equal; this looks like this:

```
equalsL :: Eq a => List a -> List a -> Bool
equalsL xs ys = ( (normalise xs) == (normalise ys) )
```

We'll say that a list `xs` is in *normal form* if it has one of the following formats:

- `EmptyList`;
- `Singleton x`;
- `Join (Singleton x) xs`, where `xs` is not `EmptyList`, and is itself in normal form.

For example, `Join (Join xs ys) zs` is not in normal form, whereas `Join (Singleton 1) (Singleton 2)` is.

Problem 3.10

Which of the following lists are in normal form? For those that aren't, write down an equivalent list that *is* in normal form.

(a) `Join (Singleton 0) EmptyList`

(b) `Join (Singleton 0) (Join (Singleton 1) (Singleton 2))`

(c) `Join (Join (Singleton 0) (Singleton 1)) (Singleton 2)`

(c) `Join (Join (Singleton 0) (Singleton 1)) (Join (Singleton 2) (Singleton 3))`

✓

Problem 3.11

Implement the function `normalise`. ✓

Problem 3.12

Test whether your `normalise` function works properly. That is, identify lists **xs** and **ys** which ought to be equal, but for which `xs == ys` returns *False*, and show that `equalsL xs ys` returns *True*. ✓

Problem 3.13

Justify your test strategy. Some questions you could address include: How many different pairs (**xs**,**ys**) do you need to test for your results to be significant? How do you choose the right pairs to test? How do you check that `equalsL` never returns *True* when it should return *False*? ✓

Problem 3.14

Suppose you were asked to prove that the functions you've defined for **List a** have essentially the same behaviours as the equivalent functions for the standard list type **[a]**. Sketch the method you would use. **Note.** You may find it helpful to consider `normalise` and `equalsL` separately from the other functions. ✓

END OF PROBLEM SHEET

COM2001 (Spring Semester): Problems, Week 4

These problems are optional; they do not contribute to your coursework assessment

You may find the following partial reminder of Haskell definitions useful:

```
class Eq a where
  (==) :: a -> a -> Bool
  -- and other stuff

class Show a where
  show :: a -> String
  -- and other stuff

data Stack a = Empty | Push a (Stack a)
```

Problem 4.15

Suppose the `Colour` data type is defined by

```
data Colour = RGB {
  red   :: Int,
  green :: Int,
  blue  :: Int
}
```

Suppose a graphical editor implements a feature which declares two colours with RGB values (r, g, b) and (r', g', b') to be equal provided these RGB values are within 10 of one another, i.e.

$$(r, g, b) \equiv (r', g', b') \qquad \Leftrightarrow \qquad (\mid r - r' \mid + \mid g - g' \mid + \mid b - b' \mid < 10)$$

Show how to make `Colour`, equipped with this definition of equality, a member of the `Eq` class by using the `instance` keyword.

Problem 4.16

A programmer wants to use the `Stack a` data type, but wants the entries in a stack to be displayed horizontally, like this:

`Empty`	*is shown as*	`[ >`
`Push 0 Empty`	"	`[ 0 >`
`Push 1 (Push 0 Empty)`	"	`[ 0 | 1 >`
`Push 2 (Push 1 (Push 0 Empty))`	"	`[ 0 | 1 | 2>`

and so on, where `[` shows the bottom of the stack and `>` the top. Show how to do this by making `Stack a` a member of `Show`, with a suitably defined `show` function.

END OF PROBLEM SHEET

COM2001 (Spring Semester): Problems, Week 5

These problems are optional; they do not contribute to your coursework assessment

A *heap* is a binary tree, t, in which the entries in any subtree, s, satify the following constraint:

the value at the root of s is greater than or equal to every other entry in s.

Writing $Heap\langle a\rangle$ for the type *heaps with entries of type a*, and $Msg\langle a\rangle$ for the type *messages to do with entities of type a*, a typical heap syntax might be:

- $createH : \rightarrow Heap\langle Entry\rangle$
  This function creates a new, empty, heap.
- $insertH : a \rightarrow Heap\langle Entry\rangle \rightarrow Heap\langle Entry\rangle$
  This function inserts an entry into a heap, and returns the new heap.
- $removeH : a \rightarrow Heap\langle Entry\rangle \rightarrow Heap\langle Entry\rangle$
  This function removes an entry from a heap if it is present, and returns the new heap. If the entry is not present, the function returns the original heap, unchanged.
- $mergeH : Heap\langle Entry\rangle \rightarrow Heap\langle Entry\rangle \rightarrow Heap\langle Entry\rangle$
  This function merges two heaps and returns the resulting new heap.
- $rootH : Heap\langle Entry\rangle \rightarrow Entry \cup Msg\langle Entry\rangle$
  This function returns the entry at the top of the heap.
- $flattenH : Heap\langle Entry\rangle \rightarrow List\langle Entry\rangle$
  This function returns the entries from the heap, sorted in decreasing order, in a list.

Problem 5.17

Which of the functions described in this heap syntax are *constructors*, and which are *observers*? [Hint. Only the function *removeH* is neither.]

Problem 5.18

For each of the *observers*, identify an attribute that needs to be included in any representation of heaps.

Problem 5.19

Implement an algebraic data type `Heap1 a`, using field label syntax, that includes a specific label for each of these attributes.

Problem 5.20

Implement the *constructors* relative to the `Heap1` representation.

Problem 5.21

Which of the entries and/or type constructors in your implementation of `Heap 1` are unnecessary, in the sense that the information or behaviour can be obtained by applying a suitably defined function instead?

Problem 5.22

Implement an algebraic data type `Heap2 a` using field label syntax, which contains no more components than necessary. For each of the labels that you identified as being unnecessary in your implementation of `Heap1`, implement a function that extracts the associated information from heaps of type `Heap2 a`.

END OF PROBLEM SHEET

COM2001 (Spring Semester): Problems, Week 6

These problems are optional; they do not contribute to your coursework assessment

Problem 6.23

Show in detail how Haskell evaluates

```
isort [1,3,2,4]
```

Problem 6.24

Show in detail how Haskell evaluates

```
treesort [1,3,2,4]
```

In particular, what does the tree look like that's constructed during the evaluation of this expression?

Problem 6.25

Show in detail how Haskell evaluates

```
heapsort [1,3,2,4]
```

In particular, what does the heap look like that's constructed during the evaluation of this expression?

Problem 6.26

(a) Is it *always* true that `treesort` sorts lists faster than `isort`?
(b) Is it *always* true that `heapsort` sorts lists faster than `treesort`?

END OF PROBLEM SHEET

```
-- Copyright (c) Mike Stannett 2005, 2006, 2007
-- CODE FOR WORKSHEET 6 (2006-07)
-- This file (sorting.hs) is available via the COM020 WebCT
-- pages, in the "Resources" folder.

-----------------------------------------------------------------
--                      Insertion Sorting                      --
-----------------------------------------------------------------
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insertL x (isort xs)

insertL :: Ord a => a -> [a] -> [a]
insertL x [] = [x]
insertL x l@(y:ys)
    | x <= y    = x:l
    | otherwise = y: insertL x ys

-----------------------------------------------------------------
--                      The TREE data type                     --
-----------------------------------------------------------------
data Tree a = EmptyT | Node (Tree a) a (Tree a)

instance (Eq a) => Eq (Tree a) where
    s == t  =  (flattenT s) == (flattenT t)

instance (Show a) => Show (Tree a) where
    show EmptyT = "*"
    show (Node EmptyT x EmptyT) = show x
    show (Node l x r) =  "{" ++ show l ++ "} "
                      ++ show x
                      ++ " {" ++ show r ++ "}"

-- constructors
createT :: Tree a
createT = EmptyT

insertT :: Ord a => a -> Tree a -> Tree a
insertT x EmptyT = Node EmptyT x EmptyT
insertT x (Node l y r)
    | x > y    = Node l y (insertT x r)
    | x <= y   = Node (insertT x l) y r
```

```
mergeT :: Ord a => Tree a -> Tree a -> Tree a
mergeT EmptyT t = t
mergeT s      t = foldr insertT t (flattenT s)

--observers
emptyT :: Tree a -> Bool
emptyT EmptyT = True
emptyT _      = False

leftT :: Tree a -> Tree a
leftT EmptyT = EmptyT
leftT (Node l _ _) = l

rightT :: Tree a -> Tree a
rightT EmptyT = EmptyT
rightT (Node _ _ r) = r

rootT :: Tree a -> a
rootT EmptyT = error "Tree.rootT: tree is empty"
rootT (Node _ x _) = x

flattenT :: Tree a -> [a]
flattenT EmptyT = []
flattenT (Node l x r) = (flattenT l) ++ [x] ++ (flattenT r)

-- mutator
removeT :: Ord a => a -> Tree a -> Tree a
removeT _ EmptyT = EmptyT
removeT x (Node l y r)
    | x < y     = Node (removeT x l) y r
    | x > y     = Node l y (removeT x r)
    | x == y    = mergeT l r

list2tree :: Ord a => [a] -> Tree a
list2tree = foldr insertT EmptyT

treesort :: Ord a => [a] -> [a]
treesort = flattenT . list2tree
```

```
------------------------------------------------------------------
--                    The HEAP data type                        --
------------------------------------------------------------------
type Heap a = Tree a

createH :: Heap a
createH = EmptyT

insertH :: Ord a => a -> Heap a -> Heap a
insertH x EmptyT = Node EmptyT x EmptyT
insertH x h@(Node l y r)
    | x > y     = Node h x EmptyT
    | otherwise = Node (insertH x l) y r

removeH :: Ord a => a -> Heap a -> Heap a
removeH _ EmptyT = EmptyT
removeH x h@(Node l y r)
    | x == y    = mergeH l r
    | otherwise = h

mergeH :: Ord a => Heap a -> Heap a -> Heap a
mergeH EmptyT h = h
mergeH h EmptyT = h
mergeH h@(Node l x r) h'@(Node l' x' r')
    | x <= x'   = Node (mergeH h l') x' r'
    | otherwise = Node l x (mergeH r h')

rootH :: Heap a -> a
rootH EmptyT = error "Heap.rootH: empty heap"
rootH (Node _ x _) = x

flattenH :: Ord a => Heap a -> [a]
flattenH EmptyT = []
flattenH (Node l x r) = x : flattenH (mergeH l r)

list2heap :: Ord a => [a] -> Heap a
list2heap = foldr insertH EmptyT

heapsort :: Ord a => [a] -> [a]
heapsort = flattenH . list2heap
```

Appendix B

Past Assignments

COM2001 (Spring Semester): Assignment 1

Deadline: 11.00am, Week n
This assignment is worth (this was worth 3 credits) of the total marks for this course

A programmer wants to implement a *deque* (double-ended queue). A deque consists of a sequence of values – you can insert values at either end of the deque, and (provided the deque isn't empty) you can query and remove values at either end. The programmer wants the ability to perform (at least) the following operations.

- **create** : Takes no parameters, and returns the empty deque.
- **addFront**, **addBack** : These add an entry to the relevant end of the deque and return the new deque.
- **empty** : Tests whether a deque is or is not empty.
- **removeFront**, **removeBack** : These remove an entry from the relevant end of the deque and return the resulting deque.
- **front** : Returns the entry currently at the front of the deque.
- **back** : Returns the entry currently at the back of the deque.

Question 1. [7%]
Design an ADT called **Deque** that satisfies the programmer's requirements; remember to include the relevant sorts, syntax and semantics.

Question 2. [3%]
Write down a Haskell implementation of your ADT.

END OF ASSIGNMENT

COM2001 (Spring Semester): Assignment 2

Deadline: 11.00am, Week n
This assignment is worth (this was worth 3 credits) of the total marks for this course

A Haskell programmer has been writing code for a *parser*. First she defines a couple of auxiliary functions and a general `ParserClass`. The functions defined in this class are supposed to tell us how to combine parsers together. For example, if `p` and `q` are parsers, then ``p `andThen` q`` is the parser that first behaves like `p` and then (if it was successful) like `q`.

```
module Com2020 where

import Hugs.Prelude

buildList ::  (a,[a]) -> [a]
buildList (x,xs) = x:xs

mergeLists ::  [[a]] -> [a]
mergeLists = foldr (++) []

class ParserClass parser where
    addToken ::  a -> parser a
    parseUsing ::  parser a -> String -> [(a, String)]
    buildWith ::  parser a -> ( a -> b ) -> parser b
    zeroOrMore ::  parser a -> parser [a]
    oneOrMore ::  parser a -> parser [a]
    orElse ::  parser a -> parser a -> parser a
    andThen ::  parser a -> parser b -> parser (a,b)
    andThenJoin ::  parser [a] -> parser [a] -> parser [a]
    alwaysOK ::  parser [a]

    alwaysOK = addToken []
    andThenJoin p q = (andThen p q) `buildWith` (uncurry (++))
```

Next she defines an algebaric data type called `Parser a`, and shows how to make it an instance of `ParserClass`. The idea she has in mind is this: an object of type `Parser a` is essentially just a function that reads a string of characters, and attempts to extract tokens of type `a` from the beginning of that string. For example, if you wanted to parse the string “`123hello`” so as to extract the integer `123`, you would use a parser of type `Parser Int`.

```
data Parser a = Parser {
    parser ::  String -> [(a,String)]
    }

instance ParserClass Parser where
    parseUsing (Parser f) str = f str
    addToken x = Parser (\str -> [(x,str)])
    (Parser f) `andThen` (Parser g) = Parser h
        where h str = [( (x,y), s) | (x,s') <- f str, (y,s ) <- g s']
    (Parser f) `orElse` (Parser g) = Parser h
        where h str = (f str) ++ (g str)
    buildWith (Parser g) f = Parser h
        where h str = [ (f p, s) | (p, s) <- g str ]
    zeroOrMore p@(Parser f) = Parser h
        where h str
            | null (f str) = [([], str)]
            | otherwise = parseUsing (oneOrMore p) str
    oneOrMore p@(Parser f) = Parser h
        where
            p' = oneOrMore p
            h str
                | null firstPass = []
                | otherwise = foldr (++) [] [
                    ([x],s) :  [ (x:xs,s') | (xs,s') <- parseUsing p' s ]
                        | (x,s) <- firstPass ]
                where firstPass = f str
```

Finally, she defines some functions to help her test if the program is working OK so far. She writes a number of functions for extracting characters from the beginning of a string, subject to some condition or other being satisfied.

```
matchIf, matchWhile ::  (Char -> Bool) -> Parser String
matchIf test = Parser t
  where
    t [] = []
    t (x:xs)
      | test x = [ ([x], xs) ]
      | otherwise = []
matchWhile test = (oneOrMore (matchIf test)) `buildWith` mergeLists

matchChar, matchChars ::  Char -> Parser String
matchChar c = matchIf (== c)
matchChars c = matchWhile (== c)

matchString ::  String -> Parser String
matchString [] = alwaysOK
matchString (x:xs) = ((matchChar x) `andThenJoin` (matchString xs))

test1 = parseUsing (matchChar 'x') "xxy"
test2 = parseUsing (matchChars 'x') "xxy"
test3 = parseUsing ( (matchChars 'x') `andThen` (matchString "xx") )
                    "xxxxy"
test4 = parseUsing ( (matchChars 'x') `andThen` (matchString "xx")
                    `andThen` (matchString "xy") ) "xxxxy"
```

Question 1. [10%]
The programmer has forgotten to add any comments to her code. Download the source code from the course website and add suitable comments throughout the program. Your comments should explain the basic purpose of each function that is defined. [Hint. You may it useful to play around with the functions first, to get a feel for the way they work. In particular, consider the values `test1`, `test2`, `test3` and `test4`.]

Question 2. [10%]
By adding code to the program file, define a general-purpose parser `matchInt` of type `Parser Int` that extracts initial integer tokens like `1`, `12` and `123` from strings like "`123.5`" and "`123 hello`". [Hint. Consider combining a suitable `buildWith` term with `matchWhile isDigit`.]

END OF ASSIGNMENT

COM2001 (Spring Semester): Assignment 3

Deadline: 11.00am, Week n
This assignment is worth (this was worth 4 credits) of the total marks for this course

YOUR WORK FOR THIS ASSIGNMENT SHOULD BE HANDED IN ELECTRONICALLY. FOLLOW THE INSTRUCTIONS ON THE COVER SHEET.

In this assignment you'll be implementing a Haskell program that mirrors the way Haskell evaluates types. To keep the project manageable, I've already developed code that can parse basic expressions and type declarations (see Appendix D). However, the types catered for are basic – there is absolutely no provision in the code for types that involve constraints, and no provision for polymorphic types.

For example, the code is currently able to deduce the following types:

Expression	Calculated Type	Represented As
`456`	`Int`	`UnaryT "Int"`
`double`	`Int -> Int`	`FuncT (UnaryT "Int") (UnaryT "Int")`
`double 456`	`Int`	`UnaryT "Int"`

IMPORTANT: If you find any part of this assignment too difficult, carry on with the next question. It is possible to complete later questions even if earlier questions are not completed correctly.

Question 1. [12%]
ADDING POLYMORPHISM

Extend the definition of `matchType` so that it becomes capable of parsing *polymorphic types* like `Either a b`. Don't forget to demonstrate, by including as comments the results of various tests, that your code accurately handles polymorphic types. [Optional Hint. Define a new type, `VarName`, capable of representing variables, like `a`, `b` and `entry`, as well as its associated parser `matchVarName`. Add a new constructor, `PolyT`, to the definition of `Type` to represent polymorphic types, and define its associated parser `matchPolyT`. Adjust any other functions that definitely need adjusting (and add comments to let us know that you've done so)]

Question 2. [8%]
ADDING ANONYMOUS FUNCTION EXPRESSIONS

Extend the definition of `typeOf` to include finding the type of *anonymous function expressions* of the form `(\x -> e)`. Don't forget to demonstrate, by including as comments the results of

various tests, that your code accurately handles type evaluations involving anonymous functions. [Optional Hint. Add a new constructor `AbstractE` to the declaration of `Expr` to represent anonymous functions, as well as an associated parser `matchAbstractE`. Add an extra case to `getType` corresponding to the new constructor. Adjust any other functions that definitely need adjusting (and add comments to let us know that you've done so).]

Question 3. [10%]
ADDING CONSTRAINTS

Define a function `matchCT`, capable of parsing *constrained types* like '`Num a => a`'. Don't forget to demonstrate, by including as comments the results of various tests, that your code accurately handles the parsing of constrained types. [Optional Hint. Define a new type, `ClassName`, capable of representing class names like `Eq`, `Show` and `Ord`, together with a variable `classLib` which the user is expected to populate with relevant classnames. Define a function `matchClassName` which is capable of parsing declared class names from a character stream. Then define a data type `CType` representing constrained types, and finally define `matchCT`.]

Question 4. [10%]
TYPE EVALUATION

Define a function `getCType` capable of determining the (possibly polymorphic, possibly constrained) type of any valid expression. If you need to amend or define any additional functions, state this clearly using comments. Don't forget to demonstrate, by including as comments the results of various tests, that your code accurately handles the evaluation of types for expressions of all kinds. [Optional Hint. Amend `typeLib` and `getTypeFromLib` to allow the declaration of polymorphic and constrained types, and define functions corresponding to Haskell's typing rules. For example, define a function `doApplicationCT` so that `doApplicationCT t1 t2` computes the type that results when a function of type `t1` is applied to a value of type `t2`.]

Appendix C

Haskell Typing Rules

Rule 1. Function Application

$$\begin{array}{rl} \text{Given} & f :: \mathit{Cons}_f \Rightarrow \sigma \rightarrow \tau \\ & e :: \mathit{Cons}_e \Rightarrow \sigma \\ \text{Deduce} & f\ e :: (\mathit{Cons}_f, \mathit{Cons}_e) \Rightarrow \tau \end{array}$$

Rule 2. Type Instantiation

$$\begin{array}{rl} \text{Given} & f :: \mathit{Cons}_f \Rightarrow \tau \\ \text{Deduce} & f :: \mathit{Cons}_f\{\sigma/a\} \Rightarrow \tau\{\sigma/a\} \end{array}$$

Rule 3. Abstraction

$$\begin{array}{rl} \text{Given} & (x :: \sigma)\ \mathit{implies}\ f :: \mathit{Cons}_f \Rightarrow \tau \\ \text{Deduce} & \backslash x \rightarrow f :: \mathit{Cons}_f \Rightarrow \sigma \rightarrow \tau \end{array}$$

Appendix D

Basic Definitions

```
{-
  COM2001 Advanced Programming Topics
  Assignment 3

  The attached code shows how to parse types and expressions, provided
  there are no constraints involved. Notice that the types of user-defined
  functions have to be declared explicitly as part of the declaration of
  the variable "typeLib".
-}


{--------------------------------------------------------------------------------
                             USEFUL AUXILIARY CONSTRUCTS
--------------------------------------------------------------------------------}

list2set :: Eq a => [a] -> [a]
list2set [] = []
list2set (x:xs)
  | x `elem` xs = list2set xs
  | otherwise   = x : list2set xs

inBrackets :: String -> String
inBrackets str = "(" ++ str ++ ")"

inBrackets2 :: String -> String -> String
inBrackets2 [] s2 = s2
inBrackets2 s1 [] = s1
inBrackets2 s1 s2 = inBrackets (s1 ++ " " ++ s2)

joinStrings :: String -> [String] -> String
joinStrings _ []  = ""
joinStrings _ [s] = s
joinStrings sep (s:ss) = s ++ sep ++ joinStrings sep ss

char2str :: Char -> String
char2str x = [x]

dig2int :: Char -> Int
dig2int d = fromEnum d - fromEnum '0'
```

```
str2int :: String -> Int
str2int s = str2intAux (reverse s)
  where str2intAux [x]    = dig2int x
        str2intAux (x:xs) = dig2int x + 10*(str2intAux xs)
```

```
{-----------------------------------------------------------------------------
                             PARSING / PATTERN MATCHING
                         Code based on Thompson, Section 17.5
-----------------------------------------------------------------------------}

infixr 5 >*>

type Parse a b = [a] -> [(b,[a])]

-- Don't match anything
matchNone :: Parse a b
matchNone inp = []

-- Add a symbol into the stream and assert match
matchAndAdd :: b -> Parse a b
matchAndAdd val inp = [(val,inp)]

-- Match if next input satisfies a given property
matchProperty :: (a -> Bool) -> Parse a a
matchProperty p (x:xs)
  | p x         = [(x,xs)]
  | otherwise   = []
matchProperty _ []       = []

-- Match the next input symbol
matchInput :: Eq a => a -> Parse a a
matchInput t = matchProperty (t==)

-- combine the results of two matchs
-- e.g. (matchLP `alt` matchDigit) checks for an LP or a digit
alt :: (Eq a, Eq b) => Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = list2set (p1 inp ++ p2 inp)

-- Chain matches together to recognise strings
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp = [((y,z),rem2) | (y,rem1) <- p1 inp, (z,rem2) <- p2 rem1 ]

-- Build values from matchd strings
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x, rem) <- p inp ]

-- Match a list of symbols all satisfying the same property
matchList :: (Eq a, Eq b) => Parse a b -> Parse a [b]
matchList p = (matchAndAdd []) `alt`
              ((p >*> matchList p) `build` (uncurry (:)))

-- Match a single symbols satisfying a property
lift :: (Eq a, Eq b) => Parse a b -> Parse a [b]
lift p = (p >*> matchAndAdd []) `build` (uncurry (:))

-- Match a non-empty list of symbols all satifying the same property
matchNEList :: (Eq a, Eq b) => Parse a b -> Parse a [b]
matchNEList p = (p >*> matchList p) `build` (uncurry (:))

-- Match an alphanumeric string of characters
matchAlphaNums :: Parse Char [Char]
matchAlphaNums = matchList matchAlphaNum
```

```
-- Match a non-empty string of inputs, one after the other
matchInputs :: Eq a => [a] -> Parse a [a]
matchInputs [] = error "matchInputs: Empty test string"
matchInputs [x] = matchInput x `build` buildMatch1
  where buildMatch1 x = [x]
matchInputs (x:xs) = (matchInput x >*> matchInputs xs) `build` (uncurry (:))
  where buildMatch2 (x,y) = [x,y]


{-----------------------------------------------------------------------------
                          PARTICULAR TOKENS OF INTEREST
-----------------------------------------------------------------------------}

matchLP = matchInput '('-- left parenthesis
matchRP = matchInput ')'-- right parenthesis
matchCO = matchInput ','-- comma
matchSP = matchInput ' '-- space
matchLB = matchInput '['-- left bracket
matchRB = matchInput ']'-- right bracket
matchPT = matchInput '.'-- point (dot)
matchSQ = matchInput '\''-- single quote
matchDQ = matchInput '"'-- double quote
matchSL = matchInput '\\'   -- slash

matchAny      = matchProperty (\_ -> True)
matchAlphaNum = matchProperty isAlphaNum
matchUpper    = matchProperty isUpper
matchLower    = matchProperty isLower
matchDigit    = matchProperty isDigit

matchWhiteSpace = matchList matchSP

matchArrow = (matchWhiteSpace >*> matchInputs "->" >*> matchWhiteSpace)
           `build` buildArrow
  where buildArrow _ = "->"

matchInt = matchNEList matchDigit

matchFloat = (matchInt >*> matchPT >*> matchInt) `build` buildFloat
  where buildFloat (x,(_,y)) = (x ++ "." ++ y)

matchChar = (matchSQ >*>
            ((matchInputs "\\'" ) `alt` ((matchProperty ('\'' /=)) `build` char2str))
            >*> matchSQ)
          `build` buildChar
  where buildChar (_,(s,_)) = "'" ++ s ++ "'"

matchString = (matchDQ >*>
               matchList ((matchInputs "\\\"" ) `alt` ((matchProperty ('"' /=)) `build` char2str))
               >*> matchDQ)
            `build` buildString
  where buildString (_,(ss,_)) = "\"" ++ (foldr (++) "" ss) ++ "\""

matchBool = (matchInputs "True") `alt` (matchInputs "False")
```

```
{-----------------------------------------------------------------------------
                                PARSING TYPES
-----------------------------------------------------------------------------}


type TypeName       = String
type TypeNameParser = Parse Char TypeName

matchTypeName :: TypeNameParser
matchTypeName = (matchUpper >*> matchAlphaNums) `build` (uncurry (:))


type TypeParser = Parse Char Type
data Type = NullT | UnaryT TypeName | ListT Type | TupleT [Type] | FuncT Type Type
  deriving (Show)

instance Eq Type where
  NullT     == NullT         = True
  UnaryT s  == UnaryT t      = s == t
  ListT s   == ListT t       = s == t
  TupleT s  == TupleT t      = s == t
  FuncT u v == FuncT x y     = (u == x) && (v == y)
  UnaryT s  == TupleT [t]    = UnaryT s == t
  UnaryT s  == FuncT NullT t = UnaryT s == t
  _         == _             = False

normalise :: Type -> Type
normalise (TupleT []) = NullT
normalise (TupleT [t]) = normalise t
normalise (FuncT _ NullT) = NullT
normalise (FuncT NullT t) = normalise t
normalise (ListT t) = ListT (normalise t)
normalise t = t

matchType :: TypeParser
matchType = matchUnaryT `alt` matchListT `alt` matchTupleT `alt` matchFuncT `alt` matchNullT

matchNullT :: TypeParser
matchNullT = (matchLP >*> matchWhiteSpace >*> matchRP) `build` (\_ -> NullT)

matchUnaryT :: TypeParser
matchUnaryT str = [ (UnaryT s, rem) | (s,rem) <- matchTypeName str ]

matchListT = (matchLB >*> matchWhiteSpace >*> matchType >*> matchWhiteSpace >*> matchRB)
            `build` buildListT
  where buildListT (_,(_,(t,(_,_)))) = ListT t

matchTupleT = (matchLP >*> matchType >*> matchRepeatingBlock >*> matchRP)
            `build` buildTupleT
 where
  matchBlock = matchCO >*> matchWhiteSpace >*> matchType
  matchRepeatingBlock = matchList matchBlock
  buildTupleT (_,(t,(blocks,_))) = normalise typ where
    typ = TupleT (t : (map getTypes blocks))
    getTypes (_,(_,x)) = x

matchFuncT :: TypeParser
matchFuncT = matchFuncNoBracketsT `alt` matchFuncInBracketsT
```

```
matchFuncInBracketsT :: TypeParser
matchFuncInBracketsT = (matchLP >*> matchFuncT >*> matchRP)
                       `build` buildFuncIBT
  where buildFuncIBT (_,(t,_)) = normalise t

matchFuncNoBracketsT :: TypeParser
matchFuncNoBracketsT = (
  (matchUnaryT `alt` matchTupleT `alt` matchFuncInBracketsT `alt` matchNullT)
  >*> matchArrow >*> matchType
  ) `build` buildFuncNBT
 where buildFuncNBT (t1,(_,t2)) = normalise (FuncT t1 t2)




{-------------------------------------------------------------------------------
                     TYPE GENERATED BY FUNCTION APPLICATION
-------------------------------------------------------------------------------}

doApplicationT :: Type -> Type -> Type
doApplicationT t1 t2
  | t2 == getFst t1  = getSnd t1
  | otherwise        = NullT
 where getFst (FuncT x _) = x
       getFst _ = NullT
       getSnd (FuncT _ y) = y
       getSnd _ = NullT


{-------------------------------------------------------------------------------
                            PARSING EXPRESSIONS
-------------------------------------------------------------------------------}

type FuncName = String
type ExprParser = Parse Char Expr
data Expr = NullE
          | FuncE FuncName
          | IntE Int
          | FloatE Float
          | CharE Char
          | StringE String
          | BoolE Bool
          | ListE [Expr]
          | TupleE [Expr]
          | ApplyE [Expr]
  deriving (Eq, Show)

matchExpr :: ExprParser
matchExpr = matchNotNullE `alt` matchNullE

matchNotNullE :: ExprParser
matchNotNullE = matchNotNullOrApplyE `alt` matchApplyE

matchNotNullOrApplyE :: ExprParser
matchNotNullOrApplyE = matchFuncE `alt` matchIntE `alt` matchFloatE `alt`
                       matchCharE `alt` matchStringE `alt` matchBoolE `alt`
                       matchListE `alt` matchTupleE
```

```
matchNullE :: ExprParser
matchNullE = (matchLP >*> matchWhiteSpace >*> matchRP)
           `build` (\_ -> NullE)

matchFuncE :: ExprParser
matchFuncE = ((matchLower >*> matchAlphaNums)
           `build` (FuncE . (uncurry (:))))

matchIntE :: ExprParser
matchIntE = matchInt
          `build` (IntE . read)

matchFloatE :: ExprParser
matchFloatE = matchFloat
            `build` (FloatE . read)

matchCharE :: ExprParser
matchCharE = matchChar
           `build` (CharE . read)

matchStringE :: ExprParser
matchStringE = matchString
             `build` (StringE . read)

matchBoolE :: ExprParser
matchBoolE = matchBool
           `build` (BoolE . read)

matchListE = (matchLB    >*> matchWhiteSpace
          >*> matchExpr  >*> matchWhiteSpace
          >*> matchBlock >*> matchRB)
          `build` buildListE
 where
  buildListE :: (Char,([Char],(Expr,([Char],([Expr],Char))))) -> Expr
  buildListE (_,(_,(e,(_,(es,_))))) = ListE (e:es)
  matchBlock = matchList matchComponent
  matchComponent = (matchCO   >*> matchWhiteSpace
                >*> matchExpr >*> matchWhiteSpace) `build` buildComponent
  buildComponent ::  (Char,(String,(Expr,String))) -> Expr
  buildComponent (_,(_,(e,_))) = e

matchTupleE = (matchLP    >*> matchWhiteSpace
          >*> matchExpr  >*> matchWhiteSpace
          >*> matchBlock >*> matchRP)
          `build` buildTupleE
 where
  buildTupleE :: (Char,([Char],(Expr,([Char],([Expr],Char))))) -> Expr
  buildTupleE (_,(_,(e,(_,(es,_))))) = TupleE (e:es)
  matchBlock = matchList matchComponent
  matchComponent = (matchCO   >*> matchWhiteSpace
                >*> matchExpr >*> matchWhiteSpace) `build` buildComponent
  buildComponent ::  (Char,(String,(Expr,String))) -> Expr
  buildComponent (_,(_,(e,_))) = e
```

```
matchApplyE = (matchNotNullOrApplyE >*> matchRepeatingBlock) `build` buildApplyE
 where
  matchRepeatingBlock = matchList (matchSP >*> matchWhiteSpace >*> matchExpr)
  buildApplyE :: (Expr,[(Char,(String,Expr))]) -> Expr
  buildApplyE (e,xs) = ApplyE (e : map third xs)
  third :: (Char,(String,Expr)) -> Expr
  third (_,(_,e)) = e

{-------------------------------------------------------------------------------
                       DECLARING TYPES OF USER-SPECIFIED FUNCTIONS
-------------------------------------------------------------------------------}

data Declaration = NullD | Declare Expr Type
  deriving (Eq, Show)

type DeclarationParser = Parse Char Declaration
matchDeclaration = (matchExpr >*> matchWhiteSpace >*>
                   matchInputs "::" >*> matchWhiteSpace >*>
                   matchType)
                 `build` buildDeclaration
 where buildDeclaration (e,(_,(_,(_,t)))) = Declare e t

declare :: String -> Declaration
declare str
  | null ds   = NullD
  | otherwise = head ds
  where ds = [ d | (d,"") <- matchDeclaration str ]

type TypeLib = [Declaration]
typeLib :: TypeLib

-- EXAMPLE: A system with 3 user-defined functions
typeLib = map declare [
  "double :: Int -> Int",
  "fst :: (Int, Int) -> Int",
  "snd :: (Int, Int) -> Int"
  ]

{-------------------------------------------------------------------------------
                       GETTING THE TYPE OF A GENERAL EXPRESSION
-------------------------------------------------------------------------------}

dec2expr :: Declaration -> Expr
dec2expr NullD = NullE
dec2expr (Declare e _) = e

dec2type :: Declaration -> Type
dec2type NullD = NullT
dec2type (Declare _ t) = t

getTypeFromLib :: FuncName -> Type
getTypeFromLib f
 | null ts = NullT
 | otherwise = head ts
 where ts = [ dec2type d | d <- typeLib, dec2expr d == (FuncE f) ]
```

```
-- Get the type of a specified expression
getType :: Expr -> Type
getType NullE = NullT
getType (FuncE f) = getTypeFromLib f
getType (IntE _) = UnaryT "Int"
getType (FloatE _) = UnaryT "Float"
getType (CharE _) = UnaryT "Char"
getType (StringE _) = UnaryT "String"
getType (BoolE _) = UnaryT "Bool"
getType (ListE []) = NullT
getType (ListE (e:es)) = ListT (getType e)
getType (TupleE []) = NullT
getType (TupleE [e]) = getType e
getType (TupleE es) = TupleT (map getType es)
getType (ApplyE []) = NullT
getType (ApplyE [e]) = getType e
getType (ApplyE (e:es)) = foldl doApplicationT (getType e) (map getType es)


{------------------------------------------------------------------------------
                           SIMPLIFIED INPUT AND OUTPUT
------------------------------------------------------------------------------}

-- Reads an expression and determines its type
typeOf :: String -> Type
typeOf = getType . asExpr

-- Reads a valid type and outputs it
asType :: String -> Type
asType str
  | null ts   = NullT
  | otherwise = head ts
  where ts = [ t | (t,"") <- matchType str ]

-- Reads a valid expression and outputs it
asExpr :: String -> Expr
asExpr str
  | null es   = NullE
  | otherwise = head es
  where es = [ e | (e,"") <- matchExpr str ]

-- Parse an expression and give its type
parse :: String -> Declaration
parse str = Declare e t
  where e = asExpr str
        t = getType e
```

END OF ASSIGNMENT

Appendix E

Solutions to Problem Sheets

COM2001 (Spring Semester): Solutions to Week 2's Problems

The most important thing you need to do, when asked to prove things about functions defined in Haskell, or when asked to show how Haskell evaluates those functions, is to *assign names to each of the various cases in the function definition.* Traditionally, if the function is called **foo**, the various cases will be called something like `[foo.1]`, `[foo.2]`, and so on. You need to assign these names because *you need to explain which rule is being used at each stage of your answer.* I've assigned names in each of the questions below (they appear as comments in the Haskell definition).

Solution to problem 2.1

We've been asked to show how Haskell evaluates `fact 3`, where `fact` is defined:

```
fact :: Int -> Int
fact n
  | (n <= 0) = 1                -- [fact.1]
  | otherwise = n * fact (n-1) -- [fact.2]
```

To answer this question,. we simply write down the steps taken by Haskell during the evaluation of the function. You have to use your own judgment as to how much detail to go into; the question is obviously concerned with applications of `[fact.1]` and `[fact.2]`, so that's what we'll focus on; I won't bother showing how Haskell evaluates other, lower-level, stuff. For example, rather than explain the Haskell definition of the function `(*)`, I'll simply say that the result follows by standard 'arithmetic'. On the other hand, the fact that Haskell evaluates brackets before other operations is important, so you may want show the evaluation of brackets, as it happens, one bracket at a time. I've done this in some of the answers below, to show you what it looks like; in other cases I've not bothered.

```
fact 3 ⇝ 3 * fact 2                    by [fact.2]
       ⇝ 3 * ( 2 * fact 1 )            by [fact.2]
       ⇝ 3 * ( 2 * ( 1 * fact 0 ) )    by [fact.2]
       ⇝ 3 * ( 2 * ( 1 * 1 ) )         by [fact.1]
       ⇝ 3 * ( 2 * 1 )                 arithmetic
       ⇝ 3 * 2                         arithmetic
       ⇝ 6                             arithmetic
```

Solution to problem 2.2

We've been asked to show how Haskell evaluates `summ [1,2,3]`, where `summ` is defined:

```
summ :: [Int] -> Int
summ [] = 0                  -- [summ.1]
summ (x:xs) = x + summ xs    -- [summ.2]
```

The evaluation proceeds as follows (as before, I'm taking low-level calculations for granted).

```
summm [1,2,3]  ⇝  1 + summm [2,3]                 by [summm.2]
               ⇝  1 + ( 2 + summm [3] )           by [summm.2]
               ⇝  1 + ( 2 + (3 + summm [] ) )     by [summm.2]
               ⇝  1 + ( 2 + (3 + 0 ) )            by [summm.1]
               ⇝  6                               arithmetic
```

Solution to problem 2.3

We've been asked to show how Haskell evaluates `diff [1,2,3]`, where `diff` is defined:

```
diff :: [Int] -> Int
diff [] = 0                    -- [diff.1]
diff (x:xs) = (diff xs) - x    -- [diff.2]
```

The evaluation proceeds as follows:

```
diff [1,2,3]  ⇝  (diff [2,3]) - 1                    by [diff.2]
              ⇝  ( (diff [3]) - 2 ) - 1              by [diff.2]
              ⇝  ( ( (diff []) - 3 ) - 2 ) - 1       by [diff.2]
              ⇝  ( ( 0 - 3 ) - 2 ) - 1               by [diff.1]
              ⇝  -6                                  arithmetic
```

Solution to problem 2.4

We asked to say whether it is true, for all lists **xs** of type **[Int]**, that

```
(summm xs) + (diff xs) == 0
```

If our answer is yes, we have to prove it. If our answer is no, we have to give a counter-example.

The answer I'm going to give for this problem is incomplete, because I'm going to assume that the list in question is *finite*, *i.e.*, that it only contains finitely many entries. We'll see later in the course how to handle proofs and evaluations that involve infinite lists (and other infinite constructs).

Given this condition, it's fairly obvious from the evaluations derived in Questions 2 and 3 that the answer is *YES*. But how do we prove it? We need to find a technique that will let us prove something to be true about every single finite list of integers - the answer is to use *induction*. Later in the course we'll come across a technique called *structural induction*, that will let us prove things about lists directly, but for now let's stick with the standard version of *proof by induction* that should be familiar from such topics as discrete mathematics. Before reading the following definition, you'll need to remember that $\mathbb{N}$ is the set of *natural numbers*, *i.e.*, $\{0, 1, 2, \dots\}$.

Proof by induction Suppose P is some predicate defined over the natural numbers. That is, given any $n \in \mathbb{N}$, the statement $P(n)$ is either *True* or *False*. In order to prove that $P(n)$ is true for every n (or in other words, in order to prove the statement $\forall n P(n)$), it is enough to prove the following two things:

- *Base Case for P.* We have to prove that $P(0)$ holds outright.
- *Induction Step for P.* We have to prove that $P(n) \Rightarrow P(n+1)$ holds for every $n \in \mathbb{N}$. That is, using the *assumption* that $P(n)$ is *True*, we have to prove that $P(n+1)$ is also *True*.

To use induction, we need to think of something to do with lists that can be expressed using natural numbers - the obvious property is *length*. The claim we'll prove (by induction) is the following

$$P(n) \equiv \texttt{[length xs == n]} \Rightarrow \texttt{[summ xs + diff xs == 0]}$$

In other words, $P(n)$ says that the required result holds whenever the list **xs** is of length n. We say that the proof is *by induction on the length of xs*. Using induction in this way, we'll be showing that the result holds if the length is 0, and hence if the length is 1, and hence if the length is 2, and hence .... Since every finite list has length n for some $n \in \mathbb{N}$, we will eventually have shown that the result holds for every finite list.

Base Case: $P(0)$ For the base case, we have to show that the result holds whenever `length xs == 0`. But if `length xs == 0`, the list `xs` must be empty, so in this case,

`summ xs + diff xs`	`==`	`summ [] + diff []`	since `xs == 0`
	`==`	`0 + 0`	by `[summ.1, diff.1]`
	`==`	`0`	arithmetic

Induction step: $P(n) \Rightarrow P(n+1)$ To prove the induction step, we first have to assume that $P(n)$ is *True*. In other words, we make the assumption that the result holds whenever `length xs == n`. Then we have to prove that the same result holds when **xs** has length $n+1$. So, let's suppose that **xs** is a list of length $n+1$. Clearly, if this is the case, then **xs** isn't empty, so we can write `xs == (h:t)`, where h is the integer at the head of **xs**, and t is the tail of **xs**. Notice that the length of **t** is precisely n, so because we're assuming $P(n)$ is *True*, we can say that `summ t + diff t == 0`. So, using the definitions of **summ** and **diff**, we find that

`summ xs + diff xs`	`==`	`summ (h:t) + diff (h:t)`	since `xs == (h:t)`
	`==`	`(h + summ t) + ( (diff t) - h)`	by `[summ.2, diff.2]`
	`==`	`summ t + diff t`	by cancellation of `h`
	`==`	`0`	by `[`$P(n)$` applied to t]`

as required. □

Solution to problem 2.5

What is the type of `compn 2 (+3) (*2)`, where `compn` is defined:

```
compn :: Int -> (a -> a) -> (a -> a) -> a -> a
compn 0 f g x = f (g x)
compn n f g x = f (compn (n-1) f g x)
```

To answer this question, we make repeated use of the following (obvious) fact:

If	`f`	`::`	`a ->`	`b`
and	`x`	`::`	`a`	
then	`f x`	`::`		`b`

We also use the fact that when a function has more than one argument, it is always applied to those arguments, one at a time, from left to right. So to work out the type of `compn 2 (+3)`

(*2), we first work out the type of compn 2, then of compn 2 (+3), and finally compn 2 (+3) (*2). But first, we need to know the types of the various parameters mentioned in this problem. We'll see later in the course that the types are a bit more complicated than suggested here, but for now we'll assume that they have the types

```
   2  :: Int
 (+3) :: Int -> Int
 (*2) :: Int -> Int
```

To start the process, let's determine the type of compn 2:

```
  compn :: Int -> (a -> a) -> (a -> a) -> a -> a
      2 :: Int
-------------------------------------------------
compn 2 ::        (a -> a) -> (a -> a) -> a -> a
```

Next, we need to apply compn 2 to (+3), which means we need to match up the types in the following table:

```
compn 2 ::   (a -> a)   -> (a -> a) -> a -> a
   (+3) :: Int -> Int
```

Clearly, we need to choose a equal to Int. Doing this we get

```
     compn 2 :: (Int -> Int)  ->  (Int -> Int) -> Int -> Int
        (+3) ::  Int -> Int
-------------------------------------------------------------
compn 2 (+3) ::                   (Int -> Int) -> Int -> Int
```

and now the rest is straightforward:

```
compn 2 (+3) :: (Int -> Int) ->  Int -> Int
           (*2) :: Int -> Int
--------------------------------------------
     compn 2 (+3) (*2) ::        Int -> Int
```

So the type of compn 2 (+3) (*2) is Int -> Int. What's more, this makes sense. According to its definition, compn is a function that expects to be given *four* arguments, but in the problem we've only supplied three of them. As soon as we specify the missing fourth parameter, the definition will return us an answer, so we can think of compn 2 (+3) (*2) as a function that returns a value whenever we provide it with a number to use as the fourth parameter. In other words, it's a function of type Int -> Int.

Solution to problem 2.6

We've been asked to show how Haskell evaluates alt (+) (-) [1,2,3,4], where alt is defined:

```
alt :: (a -> a -> a) -> (a -> a -> a) -> [a] -> [a]
alt f g (x:y:ys) = (f x y) : alt g f (y:ys)          -- [alt.1]
alt f g _        = []                                -- [alt.2]
```

The evaluation proceeds as follows:

	`alt (+) (-) [1,2,3,4]`	
$\leadsto$	`((+) 1 2) : alt (-) (+) [2,3,4]`	by `[alt.1]`
$\leadsto$	`3 : alt (-) (+) [2,3,4]`	arithmetic
$\leadsto$	`3 : (((-) 2 3) : alt (+) (-) [3,4])`	by `[alt.1]`
$\leadsto$	`3 : (-1 : alt (+) (-) [3,4])`	arithmetic
$\leadsto$	`3 : (-1 : (((+) 3 4) : alt (-) (+) [4]))`	by `[alt.1]`
$\leadsto$	`3 : (-1 : (7 : alt (-) (+) [4]))`	arithmetic
$\leadsto$	`3 : (-1 : (7 : []))`	by `[alt.2]`
$\leadsto$	`3 : (-1 : [7])`	list definitions
$\leadsto$	`3 : [-1, 7]`	list definitions
$\leadsto$	`[3, -1, 7]`	list definitions

END OF SOLUTIONS

COM2001 (Spring Semester): Solutions to Week 3's Problems

This week's questions asked you to writer a Haskell program with certain properties. The code at the end of this answer sheet gives a complete working solution.

Let's imagine that the *list* type, `[a]`, had never been defined in Haskell, and that we need to design our `List a` data type. To stop things being too simple, I'll suppose that the data structure is defined as follows:

```
data List a = EmptyList              -- the empty list
            | Singleton a            -- a list with just one element of type a
            | Join (List a) (List a) -- one list followed by another
     deriving (Show, Eq)
```

Solution to problem 3.1

We are asked to write down two different ways of constructing the List containing the values 0, 1 and 2. Here they are

- `Join (Singleton 0) (Join (Singleton 1) (Singleton 2))`
- `Join (Join (Singleton 0) (Singleton 1)) (Singleton 2)`

Does HUGS regard these lists as equal? No, it does not. The problem is that HUGS is essentially testing the two values to see if they are equal as *strings*, rather than focussing on the lists they encapsulate. This is a consequence of using the `deriving` keyword; while this allows us to use a default implementation of the equality function `(==)`, that default version isn't particularly intelligent.

Solution to problem 3.2

We are asked to write down the types of the following functions; since the error messages associated with not being able to return an object of type `a` are different from those associated with not being able to return an object of type `List a`, I've decided to treat `Msg` as a parametric type. Thus, `Msg a` means 'a message to do with type `a`', while `Msg (List a)` means 'a message to do with `List a`'. Note, this are the basic one-off types; I've not bothered to doing recursive analysis to make sure that we can always apply one function to the result of another - ultimately, I'll throw away the `Msg` types by using the `error` function instead.

- `lengthL`, returns the length of the list
  `lengthL :: List a -> Int`
- `itemAtL`, returns the entry at the given index (the first index is 0)
  `itemAtL :: List a -> Int -> (a ∪ Msg a)`
- `nullL`, returns a boolean telling us if the list is empty
  `nullL :: List a -> Bool`

- `headL`, returns the entry at the head of the list
  `headL :: List a -> (a ∪ Msg a)`
- `tailL`, returns the tail of the list
  `tailL :: List a -> (List a ∪ Msg (List a))`
- `concatL`, joins two lists together (this is the equiavalent of `xs ++ ys`)
  `concatL :: List a -> List a -> List a`
- `reverseL`, reverses a list
  `reverseL :: List a -> List a`

Solution to problem 3.3

We're asked to implement each of the functions above. To this end, I'm going to drop the `Msg` types, and use the `error` function instead. The effect this has on the type signatures is actually equivalent to choosing `Msg a` ≡ `a` and `Msg (List a)` ≡ `List a` – can you see why? The definition of **List a** says that the type has three constructors: `EmptyList`, `Singleton` and `Join`. We need to give the semantics for each function we define, by describing its effect on each of these constructors.

```
lengthL EmptyList       = 0
lengthL (Singleton x)   = 1
lengthL (Join xs ys)    = (lengthL xs) + (lengthL ys)

itemAtL EmptyList     _ = error "no entries"
itemAtL (Singleton x) 0 = x
itemAtL (Singleton x) _ = error "index out of range"
itemAtL (Join xs ys)  n
  | n < 0               = error "index out of range"
  | lenxs > n           = itemAtL xs n
  | lenxs + lenys > n   = itemAtL ys (n - lenxs)
  | otherwise           = error "index out of range"
  where lenxs = lengthL xs
        lenys = lengthL ys

nullL xs                = (lengthL xs == 0)
headL xs                = itemAtL xs 0

tailL EmptyList               = error "no entries"
tailL (Singleton x)           = EmptyList
tailL (Join (Singleton x) xs) = xs
tailL (Join xs ys)
  | lengthL xs == 0           = tailL ys
  | otherwise                 = Join (tailL xs) ys

concatL xs ys           = Join xs ys

reverseL EmptyList      = EmptyList
reverseL (Singleton x)  = Singleton x
reverseL (Join xs ys)   = Join (reverseL ys) (reverseL xs)
```

We saw above that Haskell's default mechanism for defining equality doesn't work for this data type. One solution is to write a function

```
normalise :: List a -> List a
```

that takes any list and re-writes it in some standard format. To check whether two lists are equal, we first normalise them, and then check whether the normalised versions are equal; this looks like this:

```
equalsL :: Eq a => List a -> List a -> Bool
equalsL xs ys = ( (normalise xs) == (normalise ys) )
```

We'll say that a list `xs` is in *normal form* if it has one of the following formats:

- `EmptyList`;
- `Singleton x`;
- `Join (Singleton x) xs`, where `xs` is not `EmptyList`, and is itself in normal form.

For example, `Join (Join xs ys) zs` is not in normal form, whereas `Join (Singleton 1) (Singleton 2)` is.

Solution to problem 3.4

Which of the following lists are in normal form? For those that aren't, write down an equivalent list that *is* in normal form.

(a) `Join (Singleton 0) EmptyList`
NO. Normalised version is `Singleton 0`.

(b) `Join (Singleton 0) (Join (Singleton 1) (Singleton 2))`
YES.

(c) `Join (Join (Singleton 0) (Singleton 1)) (Singleton 2)`
NO. Normalised version is `Join (Singleton 0) (Join (Singleton 1) (Singleton 2))`

(c) `Join (Join (Singleton 0) (Singleton 1)) (Join (Singleton 2) (Singleton 3))`
NO. Normalisded version is `Join (Singleton 0) (Join (Singleton 1) (Join (Singleton 2) (Singleton 3)))`

Solution to problem 3.5

Implement the function `normalise`.

```
normalise EmptyList                      = EmptyList
normalise (Singleton x)                  = Singleton x
normalise (Join  EmptyList    xs)        = normalise xs
normalise (Join  xs           EmptyList) = normalise xs
normalise (Join (Join xs ys) zs)         = normalise (Join xs (Join ys zs))
normalise (Join (Singleton x) xs)        = Join (Singleton x) (normalise xs)
normalise xs                             = xs
```

Solution to problem 3.6

Test whether your `normalise` function works properly. That is, identify lists **xs** and **ys** which ought to be equal, but for which `xs == ys` returns *False*, and show that `equalsL xs ys` returns *True*.

We can use the examples above by adding the following declarations to the program:

```
test1a = Join (Singleton 0) EmptyList
test1b = Singleton 0
test2a = Join (Join (Singleton 0) (Singleton 1)) (Singleton 2)
test2b = Join (Singleton 0) (Join (Singleton 1) (Singleton 2))
test3a = Join (Join (Singleton 0) (Singleton 1)) (Join (Singleton 2) (Singleton 3))
test3b = Join (Singleton 0) (Join (Singleton 1) (Join (Singleton 2) (Singleton 3)))
```

Here's the WinHugs output for my tests, showing that I get the right results.

```
Main> test1a == test1b
False :: Bool
Main> test2a == test2b
False :: Bool
Main> test3a == test3b
False :: Bool
Main> equalsL test1a test1b
True :: Bool
Main> equalsL test2a test2b
True :: Bool
Main> equalsL test3a test3b
True :: Bool
```

I also need to check that `equalsL` doesn't return `True` when it shouldn't. Here's some more tests. Note: I've defined `empty = EmptyList :: List Int`. If I don't state what type `EmptyList` is supposed to have, I can run into problems. For example, try evaluating `equalsL EmptyList EmptyList`.

```
Main> equalsL empty empty
True :: Bool
Main> equalsL empty (Singleton 0)
False :: Bool
Main> equalsL empty (Join empty empty)
True :: Bool
Main> equalsL empty (Join empty (Singleton 0))
False :: Bool
Main> equalsL empty (Join (Singleton 0) (Singleton 0))
False :: Bool
Main> equalsL (Singleton 0) (Singleton 0)
True :: Bool
Main> equalsL (Singleton 0) (Join empty empty)
False :: Bool
Main> equalsL (Singleton 0) (Join empty (Singleton 0))
True :: Bool
```

```
Main> equalsL (Singleton 0) (Join (Singleton 0) (Singleton 0))
False :: Bool
```

Solution to problem 3.7

Justify your test strategy. Some questions you could address include: How many different pairs (**xs**,**ys**) do you need to test for your results to be significant? How do you choose the right pairs to test? How do you check that `equalsL` never returns *True* when it should return *False*?

I've used the fact that there are three constructors. The tests run through the various forms that **xs** and **ys** can take, starting with both being `EmptyList`. Notice that some lists should be equal even though they look very different, e.g. `Join empty ys` is the same as `ys`.

Solution to problem 3.8

Suppose you were asked to prove that the functions you've defined for **List a** have essentially the same behaviours as the equivalent functions for the standard list type **[a]**. Sketch the method you would use. **Note**. You may find it helpful to consider `normalise` and `equalsL` separately from the other functions.

The first problem is proving that `normalise` works, which we can do by induction. We need to do this first, because we'll be doing proof in which we need to say whether or not two lists are *equal*, and as we've seen this requires the use of `normalise`. Assuming we've done this, we can start proving that the various function pairs have essentially the same behaviour. The first thing to do is write down the related definitions and name them. Then we'll use induction. For example, here's the proof that `concatL` is essentially equivalent to `(++)`. First I write down and name definitions of the two functions:

```
concatL xs ys = Join xs ys     -- [concatL.1]

[]     ++ ys  = ys             -- [++.1]
(x:xs) ++ ys  = x : (xs ++ ys) -- [++.2]
```

Now I use induction on the length, n, of the first list, `xs`. For the sake of illustration, here's the base case for this example.

Proof. If $n = 0$, the first list is empty, whence its normalised `List a` form is just `EmptyList`, while its usual Haskell list form is `[]`. So we have the following evaluations, as required.

```
concatL xs ys  == concatL EmptyList ys  since normalise xs == EmptyList
               == Join EmptyList ys     by [concatL.1]
               == ys                    since normalise (Join EmptyList ys) == ys

     xs ++ ys  == [] ++ ys              since xs == []
               == ys                    by [++.1]
```

□

END OF SOLUTIONS

Complete working solution

```
-- ADT: List a
--------------
data List a = EmptyList              -- the empty list
            | Singleton a            -- a list with just one element
            | Join (List a) (List a) -- one list followed by another
     deriving (Show, Eq)


-- Syntax (using error function, so no Msg entries to worry about)
---------
pushL     :: a -> List a -> List a
lengthL   :: List a -> Int
itemAtL   :: List a -> Int -> a
nullL     :: List a -> Bool
headL     :: List a -> a
tailL     :: List a -> List a
concatL   :: List a -> List a -> List a
reverseL  :: List a -> List a
normalise :: List a -> List a

equalsL   :: Eq a => List a -> List a -> Bool
equalsL xs ys = ((normalise xs) == (normalise ys))


-- Semantics
------------
pushL x EmptyList       = Singleton x
pushL x xs              = Join (Singleton x) xs

lengthL EmptyList       = 0
lengthL (Singleton x)   = 1
lengthL (Join xs ys)    = (lengthL xs) + (lengthL ys)

itemAtL EmptyList     _ = error "no entries"
itemAtL (Singleton x) 0 = x
itemAtL (Singleton x) _ = error "index out of range"
itemAtL (Join xs ys)  n
  | n < 0               = error "index out of range"
  | lenxs > n           = itemAtL xs n
  | lenxs + lenys > n   = itemAtL ys (n - lenxs)
  | otherwise           = error "index out of range"
  where lenxs = lengthL xs
        lenys = lengthL ys

nullL xs                = (lengthL xs == 0)
headL xs                = itemAtL xs 0

tailL EmptyList               = error "no entries"
tailL (Singleton x)           = EmptyList
tailL (Join (Singleton x) xs) = xs
```

```
tailL (Join xs ys)
  | lengthL xs == 0           = tailL ys
  | otherwise                 = Join (tailL xs) ys

concatL xs ys          = Join xs ys

reverseL EmptyList     = EmptyList
reverseL (Singleton x)  = Singleton x
reverseL (Join xs ys)   = Join (reverseL ys) (reverseL xs)

normalise EmptyList                     = EmptyList
normalise (Singleton x)                 = Singleton x
normalise (Join  EmptyList    xs)       = normalise xs
normalise (Join  xs           EmptyList) = normalise xs
normalise (Join (Join xs ys) zs)        = normalise (Join xs (Join ys zs))
normalise (Join (Singleton x) xs)       = Join (Singleton x) (normalise xs)
normalise xs                            = xs


-- Test-values
--------------
empty  = EmptyList :: List Int
test1a = Join (Singleton 0) empty
test1b = Singleton 0
test2a = Join (Join (Singleton 0) (Singleton 1)) (Singleton 2)
test2b = Join (Singleton 0) (Join (Singleton 1) (Singleton 2))
test3a = Join (Join (Singleton 0) (Singleton 1)) (Join (Singleton 2) (Singleton 3))
test3b = Join (Singleton 0) (Join (Singleton 1) (Join (Singleton 2) (Singleton 3)))
```

COM2001 (Spring Semester): Solutions to Week 4's Problems

You may find the following partial reminder of Haskell definitions useful:

```
foldr :: (a -> b -> b) -> b -> [a] -> b

class Eq a where
  (==) :: a -> a -> Bool
  -- and other stuff

class Show a where
  show :: a -> String
  -- and other stuff

class (Eq a) => Ord a where
  (<) :: a -> a -> Bool
  -- and other stuff
```

Solution to problem 4.1

Suppose the type RGB is defined by

```
data Colour = RGB{
  red   :: Int,
  green :: Int,
  blue  :: Int
}
```

(a) To make Colour a member of Eq, using the deriving keyword, is easy – we just add the required text at the end of the declaration:

```
data Colour = RGB{
  red   :: Int,
  green :: Int,
  blue  :: Int
} deriving (Eq)
```

(b) Suppose a graphical editor implements a feature which declares two colours with RGB values (r, g, b) and (r', g', b') to be equal provided these RGB values are within 10 of one another, i.e.

$$(r, g, b) \equiv (r', g', b') \qquad \Leftrightarrow \qquad (\mid r - r' \mid + \mid g - g' \mid + \mid b - b' \mid < 10)$$

To make Colour, equipped with this definition of equality, a member of Eq by using the instance keyword requires us to define our own version of equality:

```
data Colour = RGB{
  red   :: Int,
  green :: Int,
  blue  :: Int
}
```

```
instance Eq Colour where
  (RGB r g b) == (RGB r' g' b') =
     ( abs(r-r') + abs(g-g') + abs(b-b') < 10 )
```

Solution to problem 4.2

A programmer wants to use the `Stack a` type, but wants the entries in a stack to be displayed horizontally, like this:

Empty	[>
Push 0 Empty	[0 >
Push 1 (Push 0 Empty)	[0 | 1 >
Push 2 (Push 1 (Push 0 Empty))	[0 | 1 | 2>

and so on, where `[` shows the bottom of the stack and `>` the top. Here's how to do this by making `Stack a` a member of `Show`, with a suitably defined `show` function.

```
data Stack a = Empty | Push a (Stack a)

instance (Show a) => Show (Stack a) where
  show st = wrap (show' st) where
    wrap str = "[ " ++ str ++ " >"
    show' Empty          = ""
    show' (Push x Empty) = show x
    show' (Push x stack) = (show x) ++ " | " ++ (show' stack)
```

Note. If you type `Empty` into HUGS, you will get a message saying that `show` isn't defined. This is not an error! You get the same message if you type in `[]` as well, even though lists are a standard part of Haskell. The reason for the error message is that Haskell can't work out what type the expression `Empty` is supposed to have. To get the system to work, you have to tell it the type you're expecting (this is only necessary for the empty stack; if it contains entries, Haskell will use those entries to work out the type of the stack as a whole). If you type in `Empty :: Stack Int`, HUGS will reply, as required, with `[ > :: Stack Int`.

Solution to problem 4.3

According to HUGS, we have `3 :: Num a => a`. Given this information (and the reminder at the top of the sheet, if necessary) show how Haskell determines the type of the expression

```
\ x -> (\ y -> foldr y (x>3))
```

To answer this question, we need to remember the three rules Haskell uses to work out expression types. These rules are

Rule 1. Function Application

$$\begin{array}{rl} \text{Given} & f :: \mathit{Cons}_f \Rightarrow \sigma \to \tau \\ & e :: \mathit{Cons}_e \Rightarrow \sigma \\ \text{Deduce} & f\ e :: (\mathit{Cons}_f, \mathit{Cons}_e) \Rightarrow \tau \end{array}$$

Rule 2. Type Instantiation

Given	$f :: \mathit{Cons}_f \Rightarrow \tau$
Deduce	$f :: \mathit{Cons}_f\{\sigma/a\} \Rightarrow \tau\{\sigma/a\}$

Rule 3. Abstraction

Given	$(x :: \sigma)$ *implies* $f :: \mathit{Cons}_f \Rightarrow \tau$
Deduce	$\backslash x \rightarrow f :: \mathit{Cons}_f \Rightarrow \sigma \rightarrow \tau$

Clearly, we need to use Rule 3, so let's start by defining `f` ≡ `\y -> foldr y (x>3)`, so that we can write the target funbction in the form expected for the Rule:

```
\x -> (\y -> foldr y (x>3)) == \x -> f
```

The Rule tells us to assume that `x ::` σ, for some type σ that we've yet to identify, and see what we can deduce about `f`. Again, we're going to need Rule 3 to work out the type of `f`, so let's write `g` ≡ `foldr y (x>3)`, so that `f` can be written

```
f == \y -> g
```

As before, we'll assume that `y ::` τ, where τ is another type whose identity we've yet to determine, and see what we can determine about `g`.

Given our assumptions about `x` and `y`, we work out the type of `g`, using Rules 1 and 2. The first step is easy:

```
foldr ::  (a -> b -> b)  -> b -> [a] -> b
y ::            τ
```

which tells us that we need to take

$\tau \equiv$ `a -> b -> b`

If we do this, it follows immediately that `foldr y ::  b -> [a] -> b`.

Next, we need to work out the type of the argument `(x>3)`. As the reminder at the top of the page indicates, comparison operators are defined in the `Ord` class. So we have `(>) ::  Ord c => c -> c -> Bool`. Since we're also told that `3 ::  Num d => d`, and have assumed that `x ::` σ, we can work out:

```
(>)  ::  Ord c =>  c   -> c -> Bool
  x  ::            σ
```

so we need to take $\sigma \equiv$ `c`. Doing so gives

```
  (>)  ::  Ord c =>  c ->  c -> Bool
    x  ::            c
-----------------------------------
(>) x  ::  Ord c =>        c -> Bool
    3  ::  Num d =>        d
```

Taking `d` ⇝ `c` gives us

```
    (>) x  ::  Ord c =>              c ->  Bool
        3  ::  Num c =>              c
  --------------------------------------------
  (>) x 3  ::  (Num c, Ord c) =>           Bool
```

Now we know the types of `(x>3)` and `foldr y`, we can work out the type of `g == foldr y (x>3)`.

```
foldr y  ::                         b    -> [a] -> b
  (x>3)  ::  (Num c, Ord c) =>   Bool
```

Taking `b` ⇝ `Bool` allows us to match the types (and also forces us to update our definition of τ; see below), giving

```
      foldr y  ::                       Bool ->  [a] -> Bool
        (x>3)  ::  (Num c, Ord c) =>   Bool
  ----------------------------------------------------------
foldr y (x>3)  ::  (Num c, Ord c) =>             [a] -> Bool
```

Let's take stock of where we've got to. Having assumed that `x ::` σ and `y ::` τ, we've managed to show that

```
                σ ::  c
            τ ::  a -> Bool -> Bool
foldr y (x>3) ::  (Num c, Ord c) => [a] -> Bool
```

According to Rule 3, if the assumption that `y ::` τ allows us to deduce that `g ::` $\mathit{Cons}_{\mathtt{g}} \Rightarrow \omega$, we can deduce that `f` $\equiv$ `\y -> g ::` $\mathit{Cons}_{\mathtt{g}} \Rightarrow \tau \to \omega$. In this case, we have

$\mathit{Cons}_{\mathtt{g}}$ $\equiv$ `(Num c, Ord c)`
τ $\equiv$ `a -> Bool -> Bool`
ω $\equiv$ `[a] -> Bool`

so that

```
(\y -> foldr y (x>3)) :: (Ord c, Num c) => (a -> Bool -> Bool) -> [a] -> Bool
```

and hence

```
\x -> (\y -> foldr y (x>3))
    :: (Ord c, Num c) => c -> (a -> Bool -> Bool) -> [a] -> Bool
```

Haskell usually cites type variables in alphabetic order; this is equivalent to simulateously replacing `c` ⇝ `a`, `a` ⇝ `b`, which gives the HUGS answer

```
\x -> (\y -> foldr y (x>3))
    :: (Num a, Ord a) => a -> (b -> Bool -> Bool) -> [b] -> Bool
```

In other words, there are no constraints on `y`, but `x` has to be ordered (because we compare it with something), and numeric (because the thing we compare it with is a number).

END OF SOLUTIONS

COM2001 (Spring Semester): Solutions to Week 5's Problems

A *heap* is a binary tree, t, in which the entries in any subtree, s, satisfy the following constraint:

the value at the root of s is greater than or equal to every other entry in s.

Writing $Heap\langle a\rangle$ for the type *heaps with entries of type* a, and $Msg\langle a\rangle$ for the type *messages to do with entities of type* a, a typical heap syntax might be:

- $createH\colon\ \to Heap\langle Entry\rangle$
  This function creates a new, empty, heap.
- $insertH\colon a \to Heap\langle Entry\rangle \to Heap\langle Entry\rangle$
  This function inserts an entry into a heap, and returns the new heap.
- $removeH\colon a \to Heap\langle Entry\rangle \to Heap\langle Entry\rangle$
  This function removes an entry from a heap if it is present, and returns the new heap. If the entry is not present, the function returns the original heap, unchanged.
- $mergeH\colon Heap\langle Entry\rangle \to Heap\langle Entry\rangle \to Heap\langle Entry\rangle$
  This function merges two heaps and returns the resulting new heap.
- $rootH\colon Heap\langle Entry\rangle \to Entry \cup Msg\langle Entry\rangle$
  This function returns the entry at the top of the heap.
- $flattenH\colon Heap\langle Entry\rangle \to List\langle Entry\rangle$
  This function returns the entries from the heap, sorted in decreasing order, in a list.

NOTE. When I constructed this heap syntax, I forgot to include a function *emptyH* for testing whether a heap is empty. In fact, however, such a function is unnecessary, as it can be simulated using the functions provided. Can you see how?

Solution to problem 5.1

Which of the functions described in this heap syntax are *constructors*, and which are *observers*? [Hint. Only the function *removeH* is neither.]

Ans. The *constructors* are: *createH*, *insertH* and *mergeH*. The *observers* are: *rootH* and *flattenH*.

Solution to problem 5.2

For each of the *observers*, identify an attribute that needs to be included in any representation of heaps.

Ans. We need to include the following attributes

- *rootval*, the value at the root of the heap.
- *flattened*, the flattened 'list'-version of the heap.

Solution to problem 5.3

Implement an algebraic data type `Heap1 a`, using field label syntax, that includes a specific label for each of these attributes.

Ans. The following code includes the attributes explicitly:

```
data Heap1 a = Heap1 {
  rootval :: a,
  flattened :: [a]
} deriving (Eq)
```

Solution to problem 5.4

Implement the *constructors* relative to the `Heap1` representation.

Ans. Surprisingly, the type we define, based on this representation, could be little more than an ordered list. The only attributes specifically required by the given syntax are a value and a list; the tree structure is completely invisible. So you could quite reasonably define `Heap1` as follows:

```
data Heap1 a = Heap1 {
  rootval :: a,
  flattened :: [a]
} deriving (Eq, Show)

createH :: Heap1 a
createH = Heap1 {
  rootval = error "Heap1.root: the heap is empty",
  flattened = []
}

insertH :: Ord a => a -> Heap1 a -> Heap1 a
insertH x (Heap1 _ xs) = Heap1 y ys
  where ys = sortList (x:xs)
        y  = head ys

mergeH :: Ord a => Heap1 a -> Heap1 a -> Heap1 a
mergeH (Heap1 _ []) h2 = h2
mergeH h1 h2 = Heap1 x xs
  where xs = sortList ((flattened h1) ++ (flattened h2))
        x = head xs

-- AUXILIARY FUNCTION (insertion sort)
sortList :: Ord a => [a] -> [a]
sortList [] = []
sortList (x:xs) = ins x (sortList xs)
  where ins :: Ord a  => a -> [a] -> [a]
        ins x []      = [x]
        ins x l@(y:ys)
          | x > y     = x : l
          | otherwise = y : (ins x ys)
```

On the other hand, you might consider the fact that all heaps are binary trees to be relevant in its own right, and include this fact as an attribute that has to be included in the representation.

In that case, you might define something like

```
data Heap1 a = Heap1 {
  left      :: Heap1 a,
  rootval   :: a,
  right     :: Heap1 a,
  flattened :: [a]
} deriving (Show)

createH = Heap1 {
  leftH = createH,
  rootval = error "Heap1:root: the heap is empty",
  rightH = createH,
  flattened = []
}

insertH :: Ord a => a -> Heap1 a -> Heap1 a
insertH x h@(Heap1 l v r fs)
  | null fs   = Heap1 l x r [x]
  | x >= v    = Heap1 h x createH (x:fs)
  | otherwise = Heap1 (insertH x l) v r (sortList (x:fs))

-- ... and so on
```

Solution to problem 5.5

Which of the entries and/or type constructors in your implementation of `Heap 1` are unnecessary, in the sense that the information or behaviour can be obtained by applying a suitably defined function instead?

Ans. The answer to this problem will depend on the complexity of your answer for the last one. In the first solution presented above, `rootval` is redundant, as it can easily be identified as the head of `flattened`, but flattened is a necessary component – without it, we lose all our information as to the heap's contents. In the second solution, `flattened` is redundant (*or so it seems* - see below!), as it can be reconstructed given our knowledge of `rootval`, `left` and `right`.

Solution to problem 5.6

Implement an algebraic data type `Heap2 a` using field label syntax, which contains no more components than necessary. For each of the labels that you identified as being unnecessary in your implementation of `Heap1`, implement a function that extracts the associated information from heaps of type `Heap2 a`.

Ans. Simplifying the first solution gives

```
data Heap2 a = Heap2 {
  flattened :: [a]
}

rootval :: Heap2 a -> a
rootval (Heap2 []) = error "Heap2:root: the heap is empty"
rootval (Heap2 xs) = head xs
```

while simplifying the second solution would appear to give

```
data Heap2 a = Heap2 {
  leftH :: Heap2 a,
  rootH :: a,
  rightH :: Heap2 a
} deriving (Eq)

-- THIS IS WRONG - WHAT IF THE HEAP IS EMPTY!
flattened :: Ord a => Heap2 a -> [a]
flattened = sortList . heap2list
  where heap2list :: Ord a => Heap2 a -> [a]
        heap2list (Heap2 l x r) = (heap2list l) ++ [x] ++ (heap2list r)
```

But there's a problem – how do we identify the empty heap? Without `flattened` to hand, this is no longer possible. We *need* this capability, since we can't otherwise convert heaps to lists safely. This observation tells us that, while `flattened` itself is largely redundant, it possesses one attribute that is absolutely essential to our implementation – we can use `flattened` to determine whether or not the heap is empty. Clearly, if we're going to remove `flattened`, we need to introduce a Boolean flag of some sort to replace this particular capability. Doing so gives us the corrected solution:

```
data Heap2 a = Heap2 {
  leftH :: Heap2 a,
  rootH :: a,
  rightH :: Heap2 a,
  emptyH :: Bool
} deriving (Eq)

instance (Show a) => Show (Heap2 a) where
  show (Heap2 l v r b)
    | b         = "~"
    | otherwise = "{" ++ (show l) ++ "|" ++ (show v) ++ "|" ++ (show r) ++ "}"

flattened2 :: Ord a => Heap2 a -> [a]
flattened2 = sortList . heap2list
  where heap2list :: Ord a => Heap2 a -> [a]
        heap2list (Heap2 l x r b)
          | b         = []
          | otherwise = (heap2list l) ++ [x] ++ (heap2list r)

-- Included for illustration (see text below)
createH2 = Heap2 createH2 (error "oops") createH2 True
```

One final word. Notice that we have to give our own `instance` declaration, making `Heap2` a member of `Show`. If we leave it to Haskell, our definition of *createH* will cause problems, for when HUGS tries to show us *e.g.*, `(Heap2 createH 5 createH False)`, it will enter an infinite loop, recursively evaluating `createH` in an attempt to show us what it looks like.

END OF SOLUTIONS

COM2001 (Spring Semester): Solutions to Week 6's Problems

Solution to problem 6.1

Show in detail how Haskell evaluates

```
isort [1,3,2,4]
```

We are given the following code:

```
isort :: Ord a => [a] -> [a]
isort [] = []                           -- [isort.1]
isort (x:xs) = insertL x (isort xs) -- [isort.2]

insertL :: Ord a => a -> [a] -> [a]
insertL x [] = [x]                      -- [ins.1]
insertL x l@(y:ys)
  | x <= y    = x:l                     -- [ins.2]
  | otherwise = y: insertL x ys         -- [ins.3]
```

The evaluation of `isort [1,3,2,4]` therefore proceeds as follows:

```
 isort [1,3,2,4]
 ⇝ insertL 1 (isort [3,2,4])                              by [isort.2]
 ⇝ insertL 1 (insertL 3 (isort [2,4]))                    by [isort.2]
 ⇝ insertL 1 (insertL 3 (insertL 2 (isort [4])))          by [isort.2]
 ⇝ insertL 1 (insertL 3 (insertL 2 (insertL 4 (isort [])))) by [is.2]
 ⇝ insertL 1 (insertL 3 (insertL 2 (insertL 4 [])))       by [is.1]
 ⇝ insertL 1 (insertL 3 (insertL 2 [4]))                  by [ins.1]
 ⇝ insertL 1 (insertL 3 [2,4])                            by [ins.2]
 ⇝ insertL 1 (2 :  insertL 3 [4])                         by [ins.3]
 ⇝ insertL 1 (2 :  [3,4])                                 by [ins.2]
 ⇝ insertL 1 [2,3,4]                                      by [ins.2]
 ⇝ [1,2,3,4]                                              by [ins.2]
```

Solution to problem 6.2

Show in detail how Haskell evaluates

```
treesort [1,3,2,4]
```

In particular, what does the tree look like that's constructed during the evaluation of this expression?

The relevant code is:

```
data Tree a = EmptyT | Node (Tree a) a (Tree a)
```

```
treesort :: Ord a => [a] -> [a]
treesort = flattenT . list2tree                           -- [tsort]

list2tree :: Ord a => [a] -> Tree a
list2tree = foldr insertT EmptyT                          -- [ls2tr]

flattenT :: Tree a -> [a]
flattenT EmptyT = []                                      -- [flatT.1]
flattenT (Node l x r) = (flattenT l) ++ [x] ++ (flattenT r)  -- [flatT.2]

insertT :: Ord a => a -> Tree a -> Tree a
insertT x EmptyT = Node EmptyT x EmptyT                   -- [insT.1]
insertT x (Node l y r)
  | x > y    = Node l y (insertT x r)                     -- [insT.2]
  | x <= y   = Node (insertT x l) y r                     -- [insT.3]

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z                                     -- [foldr.1]
foldr f z (x:xs)  = f x (foldr f z xs)                    -- [foldr.2]

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)                                       -- [dot]
```

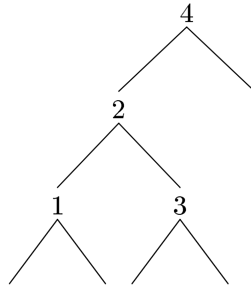
The evaluation begins as follows:

```
treesort [1,3,2,4]
== (flattenT . list2tree) [1,3,2,4]  by [tsort]
== flattenT (list2tree [1,3,2,4])    by [dot]
```

To simplify the evaluation, I'll start by evaluating the inner expression, `list2tree [1,3,2,4]`. We have

```
list2tree [1,3,2,4]
== foldr insertT EmptyT [1,3,2,4]                                          by [ls2tr]
== insertT 1 (foldr insertT EmptyT [3,2,4])                                by [foldr.2]
== insertT 1 (insertT 3 (foldr insertT EmptyT [2,4]))                      by [foldr.2]
== insertT 1 (insertT 3 (insertT 2 (foldr insertT EmptyT [4])))            by [foldr.2]
== insertT 1 (insertT 3 (insertT 2 (insertT 4 (foldr insertT EmptyT []))))  by [foldr.2]
== insertT 1 (insertT 3 (insertT 2 (insertT 4 EmptyT)))                    by [foldr.1]
== insertT 1 (insertT 3 (insertT 2 (Node EmptyT 4 EmptyT)))                by [insT.1]
== insertT 1 (insertT 3 (Node (insertT 2 EmptyT) 4 EmptyT))                by [insT.3]
== insertT 1 (insertT 3 (Node (Node EmptyT 2 EmptyT) 4 EmptyT))            by [insT.1]
== insertT 1 (Node (insertT 3 (Node EmptyT 2 EmptyT)) 4 EmptyT)            by [insT.3]
== insertT 1 (Node (Node EmptyT 2 (insertT 3 EmptyT)) 4 EmptyT)            by [insT.2]
== insertT 1 (Node (Node EmptyT 2 (Node EmptyT 3 EmptyT)) 4 EmptyT)        by [insT.1]
== Node (insertT 1 (Node EmptyT 2 (Node EmptyT 3 EmptyT))) 4 EmptyT        by [insT.3]
== Node (Node (insertT 1 EmptyT) 2 (Node EmptyT 3 EmptyT)) 4 EmptyT        by [insT.3]
== Node (Node (Node EmptyT 1 EmptyT) 2 (Node EmptyT 3 EmptyT)) 4 EmptyT    by [insT.3]
```

This tree looks like this:



Having evaluated `list2tree [1,3,2,4]`, we are in a position to show the complete evaluation; in the following evaluation, it is to be understood that the above steps are inserted at the appropriate point. The amount of detail you provide here can vary considerably; I've not bothered showing evaluations of the list function (++) — you may prefer to take the properties of (++) into account. In addition, I've sometimes performed multiple evaluations in parallel, as when I replace several instances of `flattenT emptyT` with `[]`.

```
treesort [1,3,2,4]
== (flattenT . list2tree) [1,3,2,4]                              by [tsort]
== flattenT (list2tree [1,3,2,4])                                by [dot]
== flattenT (Node (Node (Node EmptyT 1 EmptyT)
    2 (Node EmptyT 3 EmptyT)) 4 EmptyT)                          by [above]
== (flattenT (Node (Node EmptyT 1 EmptyT) 2 (Node EmptyT 3 EmptyT)))
    ++ [4] ++ (flattenT EmptyT)                                  by [flatT.2]
== (flattenT (Node EmptyT 1 EmptyT)) ++ [2] ++
    (flattenT (Node EmptyT 3 EmptyT)) ++ [4] ++
    (flattenT EmptyT)                                            by [flatT.2]
== (flattenT EmptyT) ++ [1] ++ (flattenT EmptyT) ++ [2] ++
    (flattenT EmptyT) ++ [3] ++ (flattenT EmptyT) ++ [4] ++
    (flattenT EmptyT)                                            by [flatT.2]
== [] ++ [1] ++ [] ++ [2] ++ [] ++ [3] ++ [] ++ [4] ++ []        by [flatT.1]
== [1,2,3,4]                                                     [list syntax]
```

Solution to problem 6.3

Show in detail how Haskell evaluates

```
heapsort [1,3,2,4]
```

In particular, what does the heap look like that's constructed during the evaluation of this expression?

As before, we start by showing the relevant code (we will also be referring back to some of the definitions given above):

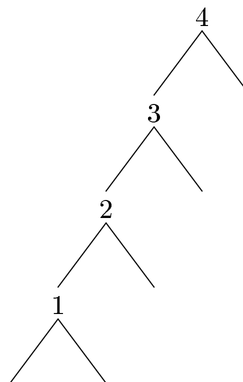
```
type Heap a = Tree a

heapsort :: Ord a => [a] -> [a]
heapsort = flattenH . list2heap                   -- [heapsort]

insertH :: Ord a => a -> Heap a -> Heap a
insertH x EmptyT = Node EmptyT x EmptyT           -- [insH.1]
insertH x h@(Node l y r)
```

```
  | x > y     = Node h x EmptyT                  -- [insH.2]
  | otherwise = Node (insertH x l) y r           -- [insH.3]

list2heap :: Ord a => [a] -> Heap a
list2heap = foldr insertH EmptyT                 -- [ls2hp]

flattenH :: Ord a => Heap a -> [a]
flattenH EmptyT = []                             -- [flatH.1]
flattenH (Node l x r) = x : flattenH (mergeH l r)  -- [flatH.2]

mergeH :: Ord a => Heap a -> Heap a -> Heap a
mergeH EmptyT h = h                              -- [mergeH.1]
mergeH h EmptyT = h                              -- [mergeH.2]
mergeH h@(Node l x r) h'@(Node l' x' r')
  | x <= x'   = Node (mergeH h l') x' r'         -- [mergeH.3]
  | otherwise = Node l x (mergeH r h')           -- [mergeH.4]
```

As before I'll simplify the evaluation by evaluating the inner expression, `list2heap [1,3,2,4]`. We have

```
list2heap [1,3,2,4]
== foldr insertH EmptyT [1,3,2,4]                                    by [ls2hp]
== insertH 1 (foldr insertH EmptyT [3,2,4])                          by [foldr.2]
== insertH 1 (insertH 3 (foldr insertH EmptyT [2,4]))                by [foldr.2]
== insertH 1 (insertH 3 (insertH 2 (foldr insertH EmptyT [4])))      by [foldr.2]
== insertH 1 (insertH 3 (insertH 2 (insertH 4 (foldr insertH EmptyT []))))  by [foldr.2]
== insertH 1 (insertH 3 (insertH 2 (insertH 4 EmptyT)))              by [foldr.1]
== insertH 1 (insertH 3 (insertH 2 (Node EmptyT 4 EmptyT)))          by [insH.1]
== insertH 1 (insertH 3 (Node (insertH 2 EmptyT) 4 EmptyT))          by [insH.3]
== insertH 1 (insertH 3 (Node (Node EmptyT 2 EmptyT) 4 EmptyT))      by [insH.1]
== insertH 1 (Node (insertH 3 (Node EmptyT 2 EmptyT)) 4 EmptyT)      by [insH.3]
== insertH 1 (Node (Node (Node EmptyT 2 EmptyT) 3 EmptyT) 4 EmptyT)  by [insH.2]
== Node (insertH 1 (Node (Node EmptyT 2 EmptyT) 3 EmptyT)) 4 EmptyT  by [insH.3]
== Node (Node (insertH 1 (Node EmptyT 2 EmptyT)) 3 EmptyT) 4 EmptyT  by [insH.3]
== Node (Node (Node (insertH 1 EmptyT) 2 EmptyT) 3 EmptyT) 4 EmptyT  by [insH.3]
== Node (Node (Node (Node EmptyT 1 EmptyT) 2 EmptyT) 3 EmptyT) 4 EmptyT  by [insH.3]
```

This tree looks like this:



The complete evaluation (subject to the same caveats as before) can be represented as follows:

```
heapsort [1,3,2,4]
== (flattenH . list2heap) [1,3,2,4]                                        by [heapsort]
== flattenH (list2heap [1,3,2,4])                                          by [dot]
== flattenH
    (Node (Node (Node (Node EmptyT 1 EmptyT) 2 EmptyT) 3 EmptyT) 4 EmptyT)  by [above]
== 4 : flattenH
    (mergeH (Node (Node (Node EmptyT 1 EmptyT) 2 EmptyT) 3 EmptyT) EmptyT)  by [flatH.2]
== 4 : flattenH (Node (Node (Node EmptyT 1 EmptyT) 2 EmptyT) 3 EmptyT)     by [mergeH.2]
== 4:3 : flattenH (mergeH (Node (Node EmptyT 1 EmptyT) 2 EmptyT) EmptyT)   by [flatH.2]
== 4:3 : flattenH (Node (Node EmptyT 1 EmptyT) 2 EmptyT)                   by [mergeH.2]
== 4:3:2 : flattenH (mergeH (Node EmptyT 1 EmptyT) EmptyT)                 by [flatH.2]
== 4:3:2 : flattenH (Node EmptyT 1 EmptyT)                                 by [mergeH.2]
== 4:3:2:1 : flattenH (mergeH EmptyT EmptyT)                               by [flatH.2]
== 4:3:2:1 : flattenH EmptyT                                               by [mergeH.1]
== 4:3:2:1 : []                                                            by [flatH.1]
== [4,3,2,1]                                                               [list syntax]
```

Solution to problem 6.4

(a) Is it *always* true that `treesort` sorts lists faster than `isort`?
(b) Is it *always* true that `heapsort` sorts lists faster than `treesort`?

The aim of this problem was to get you *thinking* about program complexity (we'll be covering it in more depth later in the course). You are not expected to have got into as much detail as I have below. In particular, I would not expect you to attempt a mathematical induction.

There are many ways to answer these questions. We'll see later in the course how to calculate the *time complexity* of the three functions, but for now I'll use the statistics provided by HUGS whenever it performs a computation. It is important to note that the statistics generated depend upon the implementation; I'm using the code cited above — other implementations may be more efficient or less efficient, and may give different answers to parts (a) and (b).

To start with, how fast do the three functions sort a list if we try the easiest possible case (taking the list to be empty). For practical reasons we need to tell Hugs what type the result should be; we do this by writing *e.g.*, `isort [] :: [Int]` instead of simply `isort []`.

Expression	Reductions	Cells
isort []	19	36
treesort []	21	35
heapsort []	21	36

The relevant statistic here is the number of reductions. Clearly, `isort` requires fewer reductions to sort the empty list than does `treesort`, so the answer to part (a) is *NO*. The number of reductions required by `treesprt` and `heapsort` are identical for the empty list; what about other short lists? The numbers shown in the table are the numbers of reductions required when the stated function is provided with the stated input.

Input / Function	treesort	heapsort
[ ]	21	21
[1]	35	32
[1,1]	69	53
[1,2]	69	53
[2,1]	59	52
[1,1,1]	120	81
[1,1,2]	120	81
[1,2,1]	93	80
[2,1,1]	93	73
[2,2,1]	100	79
[2,1,2]	110	80
[1,2,2]	120	81
[1,2,3]	120	81
[1,3,2]	93	80
[2,3,1]	100	79
[2,1,3]	110	80
[3,1,2]	93	73
[3,2,1]	90	72

A word of warning! These figures are not as straightforward as they might seem to be! We should not forget that `treesort` and `heapsort` sort their arguments in opposite orders; this means, for example, that we should compare the work performed in computing `treesort [1,2,3]` with that performed in computing `heapsort [3,2,1]`. Even so, it is clear that `heapsort` out-performs `heapsort` for every non-empty example in the table.

Suppose we want to show that `heapsort` can sometimes be slower than `treesort`. In this case all we need to do is to demonstrate the existence of a *counter-example* — a single example is all it takes to prove a claim wrong. Alternatively, we might try to prove that `heapsort` *is* always faster than `treesort`. One obvious idea would be to try induction, but it's not entirely obvious how an induction proof would work in these circumstances, because so much depends on the order of entries in the original unsorted list. Notice, however, that the workload comparisons between `treesort` and `heapsort` shown above are quite remarkable; if we focus on lists of length 2, for example, then *every* instance involving `heapsort` is faster than *every* instance involving `treesort`. The same is true for instances involving lists of length 1 and 3. Perhaps we could use induction to show that the same property holds true no matter how long the list is?

END OF SOLUTIONS

Bibliography

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, R.I., USA, 1967.

[HI98] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution.* Springer-Verlag, 1998.

[HM96] G. Hutton and E. Meijer. *Monadic Parser Combinators.* Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996. `http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps`.

[HM98] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *J. Functional Programming*, **8**(4):437–444, July 1998. `http://www.cs.nott.ac.uk/~gmh/pearl.pdf`.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, October 1969.

[HPF99] P. Hudak, J. Peterson, and J. Fasel. *A Gentle Introduction to Haskell Version 98*, September 1999. `http://haskell.org`. Also available at the course website.

[Nel95] M. Nelson. *(C++ Programmers Guide to the) Standard Template Library.* IDG Books Worldwide, Inc., Foster City, CA, 1995.

[Tho99] Simon Thompson. *Haskell, The Craft of Functional Programming.* Addison-Wesley, 2nd edition, 1999.

Index

>*>, 73

abstract data type, 5
alt - merging the results of parsing, 72
array, 49
average-case complexity, 101
axiom, 3
axiomatic semantics, 4

best-case complexity, 101
big-O, 99
big-O notation, 100
binary tree, 58
branch-point, 58
build - simplifying the results of parsing, 74

chaining parsers, 73
characteristic function, 57
completeness, 5
constraint, 39
constraint propagation, 42
constructor, 4
constructors, 36
correctness by construction, 21
currying, 75

data, 36
deque, 56
design-for-test, 21

eager, 101
Either, 8
Eq, 37
equational reasoning, 88

field label, 35, 37
Floyd-Hoare Logic, 21

Generalised principle of induction, 89

heap, 63
heap sort, 63
heap sort - statistics, 65

induction, generalised principle of, 89
induction, principle of, 89
information hiding, 51
insertion sort, 63
insertion sort - statistics, 65
instance, 16, 38
invariant, 11
invariant assertion, 11

lambda notation, 100
lazy, 101
loop invariant, 23

matchAlphaNum, 71
matchAndAdd, 71
matchAny, 71
matchArrow, 76
matchBool, 77
matchChar, 77
matchDigit, 71
matchInput, 72
matchInputs, 76
matchList, 75
matchLower, 71
matchNEList, 76
matchNone, 71
matchProperty, 71
matchString, 77
matchUpper, 71
matchWhiteSpace, 76
method, 50
monad, 70
multiplicity, 57
mutator, 4

nullary constructors, 36
nullT, 78

observer, 4
order, 99

parser, 70
parsers - simplifying results with build, 74
parsers, chaining, 73
parsers, merging results using alt, 72
parsing punctuation, 72
patterns, 4
polymorphic type, 36
post-condition, 22
pre-condition, 22
Principle of induction, 89
principle of structural induction, 92
priority queue, 56, 63
program complexity, 97
program proof, 20

projection, 57
proof by induction, 29

queue, 56

random access, 57
recursive type, 36
refinement, 21
representation, 48
root, 58

semantics, 3
Show, 38
show, 38
stacks, 1
state transition systems, 39
stateless, 53
step-counting function, 101
syntax, 1

token, 70
traverse, 56
tree sort, 63
tree sort - statistics, 65
type class, 16, 37
type determination, 42
types, 3
typing rules, 42
typing rules - abstraction, 43
typing rules - function application, 42
typing rules - type instantiation, 42

uncurrying, 75

well-formed, 1
where, 17
worst-case complexity, 101